University of Central Florida
STARS

Electronic Theses and Dissertations, 2020-

2023

"Do You Want to Build with Snowman?": Positioning Twine Story Formats Through Critical Code Study

Daniel Cox
University of Central Florida

Part of the Creative Writing Commons
Find similar works at: https://stars.library.ucf.edu/etd2020
University of Central Florida Libraries http://library.ucf.edu

This Doctoral Dissertation (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Electronic Theses and Dissertations, 2020- by an authorized administrator of STARS. For more information, please contact STARS@ucf.edu.

STARS Citation
Cox, Daniel, ""Do You Want to Build with Snowman?": Positioning Twine Story Formats Through Critical Code Study" (2023). *Electronic Theses and Dissertations, 2020-*. 1848.
https://stars.library.ucf.edu/etd2020/1848

“DO YOU WANT TO BUILD WITH SNOWMAN?”: POSITIONING TWINE STORY FORMATS THROUGH CRITICAL CODE STUDY

by

DANIEL COX
B.S. Old Dominion University, 2014
M.A. Old Dominion University, 2017

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the College of Arts and Humanities
at the University of Central Florida
Orlando, Florida

Summer
2023

Major Professor: Anastasia Salter

© 2023 Daniel Cox

ABSTRACT

Using critical code studies, this dissertation examines the Twine story format Snowman. Despite existing books on the authoring tool Twine, a central part of its functionality, what it names "story formats," is rarely covered. This study steps into this gap and, based on my own experiences through working on story formats and documenting examples using Twine, explores the greater social context of the story format Snowman through examining its source code. This dissertation consists of three chapters, each using a different set of research methods. First, the metaphor of a stack is used to better understand how software like Snowman is based on a past of other, older concepts and functionality. Second, the concept of a network is applied to better understand how software projects often rely on relationships of trust and hidden labor. Third, two other story formats, which are based on Snowman, are compared through first using a distant reading approach to find structures and then a closer reading to review how they are different. This research presents not only a greater emphasis on story formats missing from existing scholarship but also positions the story format Snowman as an important, but often overlooked, part of Twine's history.

ACKNOWLEDGEMENTS

This research would not have been possible without my PhD cohort and friends at UCF. Although the winds of fate, and, of course, a global pandemic, scattered us across the world, I am still thankful for your friendship and support through my first few years moving to Florida and becoming part of the Texts and Technology program at UCF. I truly hope we see each other again someday.

This work would also not be possible without my advisor, Anastasia Salter, who, let's be honest here, has really put up with some nonsense from me at times. As we have discussed at times, I hope both of our next decade of life is less stressful than the last.

I also want to acknowledge my committee and their feedback and support through this process. Much of this had to be finished in a short period and you all were onboard to help me get to the finish line on time. Thank you.

TABLE OF CONTENTS

LIST OF FIGURES vii

LIST OF TABLES ix

CHAPTER 1: YOU CAN UNDERSTAND THIS 1

What is Twine? 4

Why Focus on Twine? 7

Why Study Twine's Source Code? 10

Why Study the Story Format Snowman? 14

Research Questions 17

Chapter Summaries 18

Conclusion 21

CHAPTER 2: CALLING THE PAST 23

Software Foundations 26

Function Stack as Strata in Snowman 30

Starting with jQuery 31

Progressing a Story Using Underscore 35

Completing the Function 41

Conclusion 43

CHAPTER 3: TRUSTING THE PRESENT 46

Revealing Culture 48

Dependency Network of Snowman 1.4 52

Exploring the Network 58

Trust Switches 63

Conclusion 67

CHAPTER 4: REFLECTING ON THE FUTURE.. 71
Coding Crimes.. 73
Structures of Snowman 1.3 76
Adventures 1.0 80
Trialogue 0.0.8 85
Children of Snowman 89
Conclusion... 91
CHAPTER 5: END OF BEGINNING.. 95
Findings and Contributions 97
Limitations 105
Future Research.. 108
End of Beginning 113
REFERENCES 115

LIST OF FIGURES

Figure 1. Screenshot of Twine 1.4.2 showing “passages” 4

Figure 2. Screenshot of Twine 2.6.2 6

Figure 3. Line 205 of “Story.js” in Snowman 1.4 with “this” Keyword 33

Figure 4. Complete Line 205 from “src/Story.js” in Snowman 34

Figure 5. Example use of “data-passage” encoding in Snowman 1.4 38

Figure 6. Event target conversion and search in Snowman 1.4 38

Figure 7. Retrieval of data from event target in Snowman 1.4 39

Figure 8. Escaping of data passage result based on jQuery and Underscore 39

Figure 9. Completed Line 206 of Snowman 1.4. 40

Figure 10. Lines 205 - 207 in “src/Story.js” of Snowman 1.4 42

Figure 11. Abbreviated “package.json” file for Snowman 1.4 53

Figure 12. “dependencies” section of Snowman 1.4 “package.json” file 54

Figure 13. Development dependencies for Snowman 1.4 55

Figure 14. Full visualization of Snowman 1.4 dependency network 57

Figure 15. jQuery inclusion in Snowman 1.4 dependency network 58

Figure 16. Project “inherits” within Snowman 1.4 dependency network 59

Figure 17. Project “defined” within Snowman 1.4 dependency network 60

Figure 18. Project “browserify” within Snowman 1.4 dependency network 60

Figure 19. Project “cssnano” within Snowman 1.4 dependency network 61

Figure 20. Project “is-arrayish” within Snowman 1.4 dependency network 62

Figure 21. Mention of project “colors” within Snowman 1.4 dependency network 63

Figure 22. Color-coded Line-by-line Comparison between “passage.js” and “Passage.ts” 83

Figure 23. Color-coded Line-by-line Comparison between “story.js” and “Story.ts” files 84

Figure 24. Screenshot of Trialogue Visual Layout 86
Figure 25. Visualization of Changes Between Snowman and Trialogue “passage.js” Files 87
Figure 26. Visualization of Changes Between Snowman and Trialogue “story.js” Files 88

LIST OF TABLES

Table 1. Snowman 1.3 Passage Properties.. 78

Table 2. Snowman 1.3 Passage Methods.. 78

Table 3. Snowman 1.3 Story Properties.. 78

Table 4. Snowman 1.3 Story Methods.. 79

Table 5. Search Results from Adventures 1.0 Source Code 82

Table 6. Search Results for Story Keywords in Adventures Source Code.................................. 83

Table 7. Search results using “passage.js” keywords in Trialogue 0.0.8 source code................. 87

CHAPTER 1: YOU CAN UNDERSTAND THIS

Programmers often leave notes in their source code. These can take many forms of including reminders about changes they still need to perform or information to help others better understand certain aspects. Ignored by the computer when the code is run, these notes are often called "comments" and date back to some of the earliest programming languages in the 1950s (Wexelblat, 1981). In a chapter dedicated to one of the most infamous comments found in source code, Cassel (2022) describes some common patterns found in the 6th edition of the Unix operating system released in May 1975. These include obvious entries like descriptions of functionality and the occasional swear word. Then, Cassel (2022) notes the infamous comment: "You are not expected to understand this" (Cassel, 2022; p. 63). At the time, the comment was a private note to other programmers on the same project. It was left to warn off other developers from changing any of the code associated with the comment, explains Cassel (2022). Yet, unbeknownst to anyone at the time, the comment would soon become famous through the lecture notes of a professor. Requesting a copy of the new version of the Unix operating system to review for his students in his upcoming computer science courses, John Lions at the University of New South Wales began to make some new lecture notes. After reviewing all 9000 lines of code, Lions put together a line-by-line commentary for his students that also included all the source code as a reference named simply "A Commentary on the UNIX Operating System" in 1976. While originally only for students as part of a course on operating systems, the collected notes were packaged as book, photocopied, and shared around the computer science students at the University of New South Wales and then beyond the university to others. The existence of the comment quickly spread to students at other campuses and then hobbyists more generally

who had gained access to the notes. The comment was taken as a taunt and drove others to decipher why it was added to the code and what it could mean.

That comment was not written for a public audience. The source code for the 6th edition of the Unix operating system was not intended to be shared through its inclusion in teaching resources. By the time licensing around the Unix operating system source code was changed in 1979 and Western Electric Company, the company who had created the code, sought to remove access to the book, the damage had already been done. Not only was the source code spread widely, but so too was the comment. “You are not expected to understand this” had gained its infamy in many programmer circles with people quoting it to each other as a joke and the phrase appearing on t-shirts at programmer conventions (Cassel, 2017). While a somewhat silly example, Cassel (2022) includes an important aspect of its history. Yes, the comment was a private note, and an accidental book made it popular, but the act of leaving the note in the source code suggested a future, human audience. Every comment placed by a programmer “is an implicit acknowledgement,” begins Cassel (2022), “of all the careful caretakers who may someday be revisiting your code” (p. 68). While not expecting as many people to come across the note as eventually happened, its authors were still expecting others to read it through the warning to future programmers. While the comment seemingly stands alone as a short note in one version of many of an operating system, its history does not. “[I]t’s *people* who write programs,” writes Cassel (2022), adding “code is written collectively” (p. 68; original emphasis). The code of the 6th edition of the Unix operating system contains notes between fellow programmers. The sharing of the lecture notes as a packaged notes was a collective action taken by many people across multiple university campuses. The larger history of the two both point to the hidden labor behind the code and the spreading of the notes. Programmers may have written

the original code, but its longevity came about through the contributions of many others researching and explaining its history over many decades.

This dissertation explores the inherently collaborative and co-authored nature of source code. Following the example of the history presented around the spreading of the "You are not expected to understand this" comment, this dissertation explores a different work, focusing on the ways in which its source code is a text worth studying by itself. Rather than state its own warning, this introduction chapter takes as its motto "You can understand this", establishing foundational concepts and helping a reader try to make sense of how the source code of a particular software project connects to a deeper past not obvious at first glance. Centered on Twine, an open-source tool used to create nonlinear, interactive stories (Klimas*,* 2013), this dissertation explores a single aspect of its functionality, what it names "story formats" (Cox, 2021a). Each chapter of this dissertation explores the collective nature of the source code Snowman through a different metaphor per chapter. To contextualize this research, I will begin with a summary of Twine itself before moving through the important of studying its source code. Next, I review my own history with Twine and why it became a focus of this study. This is then followed by a discussion on the importance of studying code, linking it to a larger tradition of researching texts. I then explain the concept of a "story format" in Twine and how studying the story format Snowman can give greater insights into its history and connection to other software projects. The chapter ends with two sections found in a more traditional introduction to a dissertation: research questions and chapter summaries.

What is Twine?

What is known as Twine began first as another tool. In 2006, Chris Klimas created a program named Twee and a corresponding "markup language." When wanting to create a digital story, an author could use particular symbols to "markup" a story to create connections between sections (Klimas, 2006). After being processed by the program Twee, the result would be a HyperText Markup Language (HTML) file with internal hyperlinks. This followed in the footsteps of other programs such as StorySpace where authors could write fiction, apply certain symbols, and create interactive stories to be played in a web browser (Bolter & Joyce, 1987). When playing the HTML output of Twee, a reader could use these hyperlinks to navigate the story in a nonlinear way, moving between sections as long as they had links to each other (Klimas, 2008). The program Twee was later replaced with a graphical user interface to the same functionality named Twine in 2009 (Arnott, 2013). Instead of needing to write in a special format, and use a separate tool to process the symbols, authors could visually move around its sections, called "passages," showing how they connected to each other (Figure 1).

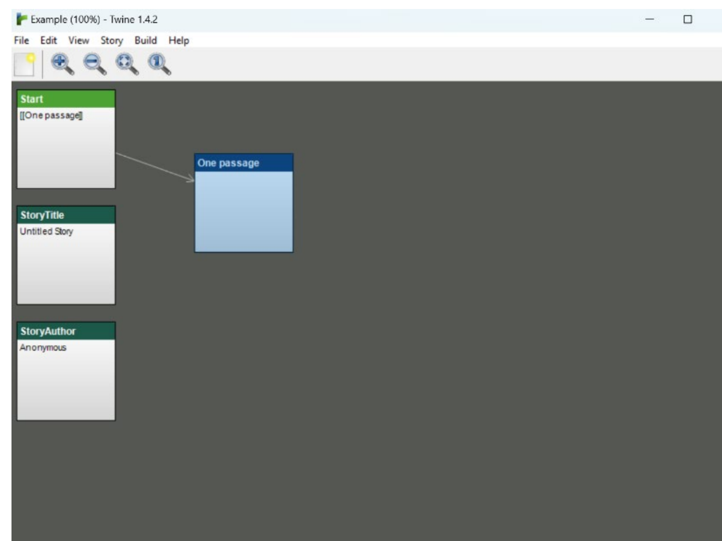

Figure 1. Screenshot of Twine 1.4.2 showing "passages"

Klimas continued work on Twine until it reached version 1.3.5 in later 2009, at which point it was set aside (Arnott, 2013). While Klimas had stopped actively working on Twine, it was slowly gaining in popularity across independent creators and programmers (Anthropy, 2009). It was featured in the book *Rise of the Videogame Zinesters* in 2012, saw its first academic coverage in 2014, and a dedicated book named *Writing Interactive Fiction with Twine* was published in 2016 (Anthropy, 2012; Friedhoff, 2014; Ford, 2016). At the same time, Twine was featured heavily in what was called a "Twine Revolution" as more diverse and minority voices used it and other tools to create more visible games including *Depression Quest* (2013), a game about experiencing the effects of depression (Quinn, 2013; Ellison, 2013). This gathering interest in Twine led to a new team of programmers working on a new version, fixing known bugs in the 1.3.5 version from 2009. This new 1.4 version was published in late 2013 (Arnott, 2013). In early 2014, Klimas returned to work on Twine, now with a new team behind the project and, on April 12, 2015, the first public access to a brand new and improved version, 2.0.4, was published (Cox, 2021b; Klimas, 2015). Unlike the solo work of Klimas on Twine up until its 1.3.5 release, the new 2.0 version included a team of people contributing fixes and working to help improve Twine. Throughout Twine 1.3.4 and into 1.4, the focus was on a desktop application. With 2.0.4, the first public version of what was called "Twine 2," this shifted to a web-based application existing both online, accessed using only a web browser, and as a desktop version (Klimas, 2015a). This new approach allowed the new web-based version of Twine to be available on more devices, no longer requiring special builds per operating system (Figure 2).

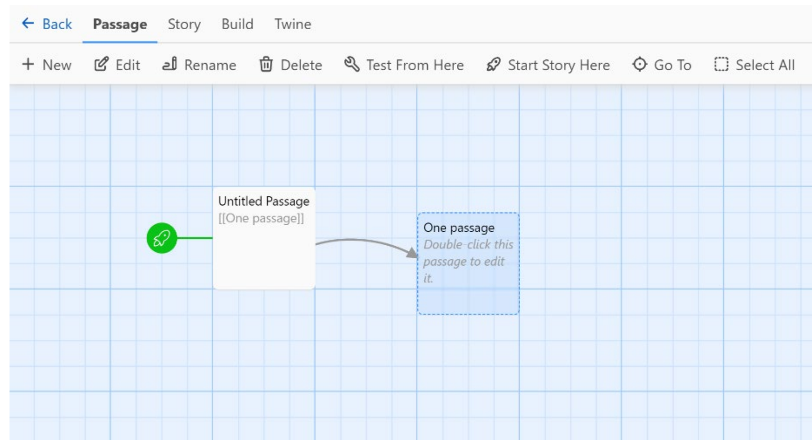

Figure 2. Screenshot of Twine 2.6.2

For many years, Twine 2 did not support the original Twee format. In this same period, other programs began to emerge as "Twine-compatible" where authors could use new programs to process the same data, producing HTML Twine could understand without needing to use the tool itself (Cox, 2019a). During this period, there was a section of the community who produced Twine works without using the tool itself. This eventually led to, nearly 15 years after its introduction in 2006, an official specification on Twee published in 2021, allowing other tools to use the format between themselves (Edwards & Cox, 2021). During and after the official specification development, pressure was placed on Chris Klimas to return its functionality to Twine itself (Cox, 2019a). Added as part of the 2.6.0 version in 2023, Twee support finally also returned to Twine, having been absent from 2015 to 2023 (Klimas, 2023a). As of this writing, authors can use the online or desktop versions of Twine 2.6 to create a story through visually arranging and creating links between passages. They can also, lost since the time of Twine 1.4, also use the Twee format again, allowing authors to write their stories, add symbols to create connections, and use a program, this time including Twine itself, to make HTML files playable in web browsers. Through community support over many years, this functionality finally returned to the graphical program inspired from the original format.

Why Focus on Twine?

My own connection to Twine and its community began in December 2012 during the time of Twine 1.3.5. While taking part in a casual competition for creating games with many other people, I encountered the work *CYBERQUEEN* (2012), created by Porpentine. *CYBERQUEEN* (2012) is a dark and claustrophobic work where the reader is faced with making decisions they have little context for and through which they are further drawn into a twisted world where the screen often shows the description of what might happen next through on-screen prompts such as choosing between "wet" and "sticky" (Porpentine, 2012). Being intrigued by both *CYBERQUEEN* (2012) and Twine as a tool for creating experiences like it, I began to try to learn it. Very quickly, I found its existing documentation frustrating and began to create my own. This led to, within a couple of weeks of playing *CYBERQUEEN* (2012), my first YouTube video on how to use Twine 1.3.5 in January 2013 (Cox, 2013b). The response to the video was very positive and I decided to begin to create more tutorials. This continued intermittently through mid-2015, with me writing a "Twine Tuesday" series most weeks (Cox, 2013a) and further creating dozens of videos covering examples of how to create different things with Twine into 2016 (Cox, 2016). In early 2017, I was invited to join the Twine Committee as part of a new non-profit named the Interactive Fiction Technology Foundation (IFTF) holding the copyrights for multiple digital storytelling tools for future safekeeping (*Our Mission and Goals*, n.d.). The Twine Committee was formed as a group of people governing "the maintenance, preservation, and improvement of the Twine project's intellectual property, infrastructure, and community" (*Twine Committee*, n.d.). I accepted the invitation with my first major project the creation of a dedicated teaching resource named the "Twine Cookbook" with examples based on my own videos and edited for a more public audience. While visiting the University of Central Florida

during a conference on November 7, 2017, I was interviewed by Anastasia Salter and Stuart Moulthrop for their book project on the theory and practice of creating with Twine that would eventually be published in 2021 (Salter & Moulthrop, 2021). At the time, I was coming to the end of my M.A. program at Old Dominion University and the interest in Twine as a research subject, something I had not seen in academia up to that point, pushed me into considering applying to UCF's Texts and Technology Ph.D. program in the months following the conference.

From 2017 into 2021, I acted as the primary contributor and author of the hundreds of pages of the Twine Cookbook while also continuing to create my own video series (Cox, 2016, 2017, 2019b, 2020b). Having a strong interest in acting as a historian and archivist for the Twine community, I also agreed to take over working on some functionality Chris Klimas had been maintaining on his own in 2018 (Cox, 2018). This led to my greater involvement in bringing a documentation focus to an aspect of Twine named "story formats", with me acting as a co-author on the collection of files named "Twine Specifications", technical documents explaining what Twine accepts as input and produces as output (Cox, et al., 2019). Through 2021 and into early 2023, I maintained an interest in Twine, but also focused on bringing my insider knowledge to new academic projects. I acted as a collaborator on research into studying how people understand their audiences when creating digital stories using Twine (Daiute et al., 2021), a history of the tool in connection to others (Cox, 2022b), and on bringing augmented reality to Twine (Berge et al., 2022). This study would not be possible without my insider knowledge, nor would aspects of its history be able to be studied by other scholars without my labor across many years in creating resources like the Twine Cookbook and the Twine Specifications.

The answer to the question "Why focus on Twine?" might be better stated as "How could I not write about Twine?" As the last two paragraphs explain, I have spent the better part of over

a decade documenting tools like Twine through creating not only hundreds of YouTube videos, but also contributing learning resources consisting of tens-of-thousands of words. There is no objective viewpoint for investigations into the archives and history around Twine for me. Most of the public-facing resources were written by me. My feelings around the labor behind the code will, at times, come through in my writing about its history, and this is not to be rejected nor to feel shame about during the process of trying to record it for others. As someone with a deep emotional stake in my own work with Twine over many years, I cannot always separate myself as a programmer who wrote the code from myself as a researcher presenting findings months to several years later. While not explicitly relying on its research methods, this study comes from recognizing, as the field of critical autoethnography calls for, "responsibility for our subjective lenses through reflexivity" (Boylorn & Orbe, 2021; p. 3). Describing the role of researchers when it comes to their emotional closeness to their work, D'Ignazio & Klein (2020) call this recognizing the "value [of] multiple forms of knowledge, including the knowledge that comes from people as living, feeling bodies in the world" (p. 104). Rather than pretend an objective stance is possible, my research presented in this study follows in the call of D'Ignazio and Klein (2020) to create research "informed by direct experience" (p. 21). It is through my knowledge and direct experience this research is possible, and I acknowledge many of the sources used in this study are those either I created or are possible through the work I have done in the past. This study is an investigation of the inherently co-authored nature of programming, and nothing could be more feminist than to explain my own positionality and passion as it comes to the work presented based on over a decade of personal labor.

Why Study Twine's Source Code?

To begin to understand why this study is focused on a single aspect of Twine means beginning with the importance of research into source code itself. This study is indebted to a deeper history around the studying of the relationships between texts and the technologies that produce them. Bolter (2001) traces the roots of the term 'technology' as based in Ancient Greek. The concept of *techne* includes in its definition the making of meaning or creation of some output based on a system of rules. Bolter (2001) describes how a "writer always needs a surface on which to make [their] marks and a tool with which to make them" (p. 15). The tools and methods used to make the marks is technology, based on a system of rules, and the process of writing is the creation of texts. These two concepts cannot be separated. Ong (2012), in *Orality and Literacy*, makes a similar claim, stating how "Writing, commitment of the word to space, enlarges the potentiality of language almost beyond measure" and "re-structures thought" (p. 7). The act of authoring, as presented by Ong (2012), is one of potentially changing how thinking works. Programming is, of course, another form of writing. Vee (2017) makes the immediate connection between being literate in coding, being able to read and write in a programming language, and the power this gives people within digital spaces. Vee (2017) writes how programming "is the act and practice of writing code" (p. 19) and how knowledge of coding influences how people "function in society" because of "the ways literacy is attached to power" (p. 27). By highlighting code in its textual form, Vee (2017) provides the gap between the importance of writing and the role of software in shaping how works are created. Vee (2017) also adds an aspect of power and its relationship to coding as well. Placed in a digital setting, the role of writing, the structuring of text using technology, is done using tools which are themselves

based on programming. The writing of code is an act of power enabled through access to technology and the literacy needed to use digital systems to create new works.

Through interacting with software, we are also interacting with the cultural forces behind its construction and its maintenance. In *Software Takes Command*, Manovich (2013) defines "cultural software" as "types of software that support actions we normally associate with 'culture'" (p. 20). Within the introduction to the book, Manovich (2013) categorizes software, with an important category being "[development] software tools and services that support all these [cultural] activities" (p. 23). By adopting a definition of cultural software as those providing functionality for creating other programs, the term "culture software" becomes a description of not only software supporting culture but also containing its own culture and influence over users through its systems and interfaces. How code is written becomes its own form of "culture" within the crafting of structures and connections to other, existing source code within these tools. While Bogost (2007) is primarily writing about games in his book *Persuasive Games*, an important term used in the book, "procedural rhetoric", also applies here. Bogost (2007) describes procedural rhetoric as "the practice of persuading through processes in general and computational processes in particular" (p. 2). Through the interfaces created using development tools, the ways in which people and digital systems relate to each other can be influenced. Not only does each programming language provide different functionality, the tools used to edit those languages, themselves products of programming languages, also contribute to their persuasive nature. If coding is an act of power enabled through technology, then the structures within code, created based on combinations of software tools and other humans, need to be studied for how they embed cultural values and hold persuasive influence over other systems and the people in which they interact. Code cannot be thought as only existing as

running on computers, as people read and write in programming languages as well, as the history around the comments in the Unix operating system remind us (Cassel, 2022). This duality of code positions it as inherently social. Other tools and people are needed to create new textual works to continue spreading past embedded cultural values into future software works.

Early in the book *Critical Code Studies*, Marino, (2020) defines code as a “social text” (p. 4). Depending on the reader, its meaning is changed. If a human is reading the code, its translation may have one meaning. If a computer is running it, the code may produce a different outcome. A mistranslation of the code by a computer is the same as a human attempting to read a language they do not understand. Meaning may be made, but not a complete or intended meaning by the original author. First and foremost, then, code is a text. It is read by different audiences and produces different meanings for them within a context. This collection of speaker, audience, and text places code as firmly within the tradition of rhetoric, as meaning making is contingent on who is producing the message, what the message is, and who is receiving it. At the same time, programming cannot be separated from the tools and services used to create with it. These embed cultural values and continue existing power structures influencing the programming language and tools created with it (Manovich, 2013). For humans writing code, they translate the symbols into meaning based on their own understanding and literacy with the digital tools used for editing (Vee, 2017). The necessity of studying code, then, is directly tied to the amount of power it exercises over the lives of the people who interact with it. As Marino (2020) explains, “If code governs so much of our lives, then to understand its operation is to get some sense of the systems that operate on us” (p. 3). While software has great potential for empowerment, its misuse can lead to greater oppression for minority groups, replicating the culture of its developers and editing tools in the applications created (Noble, 2018). Thus,

studying code becomes essential for understanding what additional values are entangled in software and how the structures found in source code affect other systems. This means examining not only the source code, but also its social relationships to other systems and people.

The importance of studying the source code of Twine is found in its connection to its social context. The history of Twine is one of being the output of many people working together and contributing new ideas and approaches. While Chris Klimas has had a major influence in its direction over its lifetime, Salter and Moulthrop (2021) document the history of Twine as influenced by the books created by Melissa Ford (Ford, 2016) and Anna Anthropy (Anthropy, 2012). There was also my work on the Twine Cookbook (Cox, 2021c) and dozens of tutorial videos across many years. Marino (2020) writes of how other texts should be examined in connection to source code and how this impacts its meaning in context. The use of the word "critical," explains Marino (2020), is intentional in the establishment of the field in connection to this idea: "[t]he *critical* in critical code studies encourages also exploring the social context through the entry point of code" (p. 28; original emphasis). This means examining code as more than a single file or collection, but in conversation with other works, tools, and development services in which it interacts (Manovich, 2013). As described by Marino (2020), "every piece of source code is only ever partial" (p. 48). To investigate the source code of some software means looking into what "missing" parts might not be evident in a file and supplied by other software libraries or packages on which it relies to run, test, or process its source in some way. Twine can be better understood through its social connections, digging into its source code and what structures might be present there.

Why Study the Story Format Snowman?

A strong case can be made for the study of Twine in its role in the "Twine Revolution" (Ellison, 2013). The same, too, for how Twine has enabled new forms of creativity and the importance of focusing on the theory and practices around it (Salter & Moulthrop, 2021). Twine could also be centered as a tool for creating interactive fiction (Ford, 2016). Yet, all these approaches have been covered previously. What has yet to be a focus in research is story formats themselves. When described by Ford (2016) in the book *Writing Interactive Fiction with Twine* (2016), each story format is summed up as "unique way[s] of writing scripts" (p. 125). Throughout most of the book, the emphasis is on only the default story format of Twine 2, Harlowe, with the first mention of using a different one coming significantly through most of its material with Ford (2016) writing "the [functionality] you love from Harlowe [is] available" in other story formats as well, but that it is "written with a different syntax" (p. 460). In fact, it is easy to think of story formats in Twine as little more than formatting differences. Throughout the period of Twine 1.3.5 and 1.4 from 2009 to 2014, they were described as "visual layouts" (Cox, 2021a). When creating a story in Twine 1.3.5 and 1.4, an author could choose between using the default story format, Sugarcane, and an alternative, Jonah. Both used the same programming functionality but produced different HTML presentations. When using the first, Sugarcane, each passage within the story was shown separately, replacing the contents of the current with the next. Jonah, on the other hand, appended content vertically, allowing a reader to scroll back through what sections of a story they had visited previously as one long journey shown on the screen (*Story Formats - Twine Wiki*, 2014). The role of story formats changed substantially with the introduction of Twine 2.0.4 in 2015. Instead of a single "language" with different presentations available, each story format became a different approach to using Twine. At first,

the story formats packaged with Twine were similar in what they offered to authors: Harlowe, the default option, uses parentheses to mark its functionality; SugarCube, another story format, uses less-than and greater-than symbols; and Snowman, provided a way to tap into the programming language JavaScript available in web browsers (Cox, 2021c). However, they quickly departed from each other in what they offered to authors and how it could be used.

Throughout working on the Twine Cookbook from 2017 to 2021, I was constantly put into a position of needing to translate between one story format and another. Many story formats for Twine 2 use a concept called "macros" carried over from earlier version of Twine, "a shortcut to performing a specific task" (Ford, 2016). These define functionality an author can use using one or more words. What might exist as functionality provided in one story might not exist in another. This meant an example of a particular pattern appearing in one story format becomes impossible to replicate in another. The same too with certain functionality around other media, with SugarCube supporting playing audio files, for example, with the same functionality not appearing in others. Salter and Moulthrop (2021) get the closest to understanding these differences in explaining how each story format has grown into "ways of thinking about making" in their introduction to their book (p. 15). It comes as no surprise the documentation for the default story format for Twine 2, Harlowe, moved to call itself as a "programming language" in 2022 (Arnott, 2022). Even SugarCube includes a nod in this direction, describing how to use its functionality as part of what it names "TwineScript" (Edwards, 2021). Yet, what ties both Harlowe, SugarCube, and all other story formats for Twine together is an aspect covered some in Salter and Moulthrop's (2021) work but rarely appearing in others: an author cannot write a story in Twine without also writing within the "programming language" defined by the story format they have chosen. It is impossible to use Twine to create a story and not engage with a story

format. The interface through which stories are written are crafted within the boundaries as established by the authors of the story format. Not only do they carry within themselves their own cultural values (Manovich, 2013), but how functionality is offered requires a level of literacy about what is available and how it can be used (Vee, 2017). All authors must write within the bounds of a language.

With all story formats carrying this importance to authoring in Twine, the choice of only one as a research subject may seem odd, but Snowman was picked for multiple reasons. First, as alluded to earlier in this chapter, I took over the maintenance of Snowman from Chris Klimas in 2018 (Cox, 2018). Much of this research comes from my "direct experience" (D'Ignazio & Klein, 2020) with the code through not only working on Snowman but also serving as a co-author on the Twine HTML Output specification (Cox et al., 2019). While there are others with some of my knowledge, I bring to this dissertation and its presented research detailed, insider experience with concepts and on coding structures no one else has. Beyond this, Snowman, as a story format, has not been covered in either Ford's (2016) or Salter and Moulthrop's (2021) books on Twine. It also does not appear in the independent collection *The Twine® Grimoire* (Baccaris, 2020, 2021). This gap in research and academic interest matches previous work on the "forgotten" aspects of many digital storytelling tools (Koenitz & Eladhari, 2019). Without researchers stepping in, like with the narrative around the "You are not expected to understand this" comment in the 6th edition of the Unix operating system, its history remains untold and potentially forgotten. As Cassel (2022) reminds, "code is written collectively" (p. 68). The way to better understand Snowman and its connection to other works is to investigate these relationships to past software libraries and possible future iterations by examining its "social context" (Marino, 2020; p. 28). With no one stepping up to tell the story of Snowman, I have

chosen to do so. As with my previous research into the history of Twine and other tools, they survive longer by others deciding to tell their stories and adding to the existing resources around them (Cox, 2022b). This dissertation is, in many ways, a narrative around Snowman as told through its relationships to other works, including those I created myself.

Research Questions

Each chapter in this dissertation was driven by research questions crafted to investigate a particular aspect of the relationships of the story format Snowman to other software libraries and projects. Arranged in order, these questions, and each chapter's associated research in attempts to answer, seek to position Snowman within a "geology" of sorts running from its relationships to past software, across its present coding dependencies, and into possible futures by reviewing projects based on the concepts and code from Snowman developed after a particular moment in its history. Each chapter following this introduction goes into greater detail of the theoretical basis with the questions included here for formatting purposes.

- Research Question 1: How can the metaphor of "strata" (Foucault, 1969) serve as a lens to understand the archaeology (Parikka, 2015) of function calls (Soloman, 2013) within the story format Snowman?
- Research Question 2: What can be learned from studying the "network" (Latour, 2007) of the code and testing dependencies of the story format Snowman through the ways in which some projects form "switches" affecting its arrangement and relationships? (Castells, 2010)?
- Research Question 3: How can applying a "macroanalysis" approach (Jockers, 2013) of using both "distant" (Moretti, 2013) and "close", comparative readings (Harvey & Pagel,

1991) of both “child” and “grandchild” source code be used to show a “legacy” of the story format Snowman in those projects based on it?

Chapter Summaries

The best place to begin to understand Snowman is to begin with the software libraries on which it is based. The next chapter examines three lines of code from the current version of Snowman, 1.4, connecting how parts of the code rely on other software libraries within their own histories and purposes. This chapter draws heavily from the concept of “strata” from the work of Foucault (1969) in understanding history as composed of intersecting “moving” and seemingly “unmoving histories” such as wars on one side and “crop rotations” on the other. Rather than take the obvious histories, scholars should seek to find where they intersect and work through what “system of relations” exist to be studied at these points (p. 3). Taken up within the field of media archaeology, Parikka (2015) calls for the studying of “nonlinear strata” where the digital present intersects with the material past (p. 6). In the edited collection *Media Archaeology*, Huhtamo and Parikka (2011) describe this approach as studying the moments where the “past is brought to the present, and the present to the past” of digital works (p. 15). Mapping the metaphor of strata over into programming, this chapter studies the moments of intersection between past and present when different software libraries communicate using the concept of a function. Within programming, larger programs are often broken down into smaller sections with their own input and output to solve smaller tasks named functions. These different sections interact with each other through the verb “calling” by sending input and waiting for a response. When one function calls another that calls another, they form into what is named a “stack” where the response of one is fed as input into another. Yet, in this arrangement, there is considerable

power concentrated at the function deepest in the past or upon which other codes rests. This chapter explores the metaphor of strata as applied to the function stack, exploring how three lines of code in Snowman highlight how the present version of the code "calls" back into a much deeper history of concepts and software libraries spanning years and even decades into the past through something seemingly as simple as responding to a reader clicking on a hyperlink in an HTML file.

While the first research chapter explores the past, the second examines the present. Software is often built on parts written by other people, and Snowman 1.4 is no exception. This chapter begins with the metaphor of a "network" as described by Latour (2007) and a complication to this understanding through what Castells (2010) names a "switch." Many systems are composed of multiple entries all contributing "work" toward some goal. Latour (2007) describes these collections as being part of "a vast array of entities swarming" of relative equality between them (p. 44). Yet, it is Castells (2010) who complicates networks as being inherently unequal in their constructions with certain "privileged instruments of power" among them. Within network structures, these "instruments of power" are named "switches" and are capable of "shaping social structure" around them (p. 510). Even a small change in a switch can ripple across a network and affect its edges. By being part of a network, all forces are affected by switches in some way, even if the direct relationship is not at first obvious. This chapter maps the metaphors of networks and switches onto programming by creating a visualization of all the other code projects needed for Snowman 1.4 to be prepared for use with Twine. Through creating a visualization of the hundreds of projects on which Snowman 1.4 is dependent, multiple examples are highlighted from all the relationships to examine the ways in which not only are code projects dependent on the most connected, but often rely heavily on the work of

the seemingly least connected, those on which large parts of an overall network would not otherwise work. As with the findings of the previous chapter around the structure of the stack, those on the edges of a dependency network often have greater power over its shape and arrangement than seem at first glance. This chapter ends with a reflection on the many smaller open-source projects many larger software collections are reliant on and what these switches can mean for issues like security.

The last research chapter looks to the future of Snowman through comparing its structures to those in other story formats not packaged with Twine. Between its introduction in 2015 along with Twine 2.0.4, Snowman's source code stayed relatively the same until a new maintainer took over the project in late 2018 (Cox, 2018). Between most of 2015 to 2019, other story formats were created based on the current version of Snowman at the time, 1.3. Through first doing a close analysis of the names of important structures in the Snowman 1.3 code, the text mining tool Orange was used to search through the source code of two other projects and identify files with the same names (Demšar et al., 2013). These structures were then compared back to the original, following the call of Tornhill (2015) to treat code as a "crime scene" where the text itself is examined and closely studied outside of running it through an approach named "static analysis" (Rival & Yi, 2020). By moving from a larger search to closer analysis, this chapter compares how two other story formats are different from the Snowman 1.3 code and how these changes manifest in what expectations they are fulfilling for the authors using them. This chapter, like its sister chapters, also examines the concepts of Twine manifest in Snowman, and how interpretations of these concepts, its own "cultural values" in the words of Manovich (2013), affect how the other story formats. While Snowman itself may have seen fewer updates

over the lifespan of Twine, its longevity lives on behind the multiple story formats based on its code structures and concepts affecting how authors use Twine to create new stories in the future.

Conclusion

The complexity of story formats is often erased from the history of Twine. In popular coverage of Twine, story formats are not mentioned nor does the differences in their functionality appear (Hudson, 2017; Robertson, 2021). Salter and Moulthrop's (2021) book is one of the very few sources who engage with the topic at any length. Story formats represent the labor of many others, but this fact is often completely overshadowed with a focus on Twine as a tool containing these other works. The story format Snowman, as an example of this labor, is the focus of this study, bringing greater attention to both its own positioning within Twine’s history and how it depends on a past and influences the future of other story formats. This introduction chapter opened with the example of the “You are not expected to understand this” comment as studied and examined by Cassel (2022). While perhaps popular to those in the know in the 1970s, the history of the comment contains an important seed of understanding of all programming projects and central to this study: even if working alone, a programmer is interacting with and through the labor of others. Programming is never truly an individual action. Like all texts, source code pulls from the work that came before it to establish concepts, procedures, and structures on which new works are based. As Cassel (2022) writes, “code is written collectively” (p. 68). This dissertation is itself built from the past of my own work across over a decade of written guides, video tutorials, and more public collections like the Twine Cookbook and Twine Specifications. Had I not encountered *CYBERQUEEN* (2012) in December 2012, I might not have created video tutorials on Twine in 2013, been interviewed in 2017, and ultimately moved to Florida to study at the University of Central Florida. Had I not decided to

take over the maintenance of the story format Snowman in 2018, this dissertation might have become something very different than the research presented after this introduction chapter. When Marino (2020) writes on how "visible code is always partial", the same could also be said about the past on which this study rests (p. 75). There is always more to discover using new metaphors and theoretical frameworks for studying code. As, hopefully, the following chapters reveal to readers, Snowman itself is indebted to a past stretching back many years even as its present is dependent on hundreds of other projects for it to be processed and included with Twine. What its future holds is still being written. While this study is one story told about Snowman positioned among other, existing narratives on other story formats and Twine itself, others will follow and, it can only be wished, built from this research labor as well.

CHAPTER 2: CALLING THE PAST

The history of programming languages begins with wanting to write code faster. After the unveiling of the first digital computer, Electronic Numerical Integrator and Computer (ENIAC), in 1945, researchers across businesses and universities immediately sought ways to “automate” its programming. At the time, each new computer program required re-inventing the same processes and structures. Through having a way to automate common parts, programmers could concentrate on only the new problems they were trying to solve rather than revisit those already known. During the continued development of the ENIAC, one of its lead researchers documented a need for common “routine” operations such as “logarithm, cosine, arctangent, or square root” be built into computers. Beyond these “routine” operations, there was also the need for other operations a programmer might need to add under the same category. Named “subroutines,” these were envisioned to help programmers with “the repetition of their coding,” as first described in an article appearing originally in 1947 (Mauchly, 1982). This concept was refined by other researchers with a more technical description of “subroutines” appearing in an academic paper in 1952 as “a self-contained part” and “an entity of its own” within a larger program designed to allow a programmer “to concentrate on one section of a programme [sic] at a time without the overall detailed programme [sic] continually intruding” (Wheeler, 1952; p. 235). Two years later, in 1954, the first general-purpose programming language, FORTRAN, introduced itself to a more public audience itself through its acronym and description. Within a 1956 programmer’s reference book for the language, FORTRAN is defined as a “FORmula TRANslating System” through which “automatic coding” (p. 1) could be accomplished through a series of symbols “closely resembling the ordinary language of mathematics" (Backus et al., 1956; p. 2). The programmer’s reference for FORTRAN also continued the use of the term

subroutines. In trying to match “the ordinary language of mathematics," the programmers and researchers behind FORTRAN borrowed a term from mathematics to help more general audiences understand how subroutines worked in their system: they named them “functions.” FORTRAN came with a set of functions it called a “library” with the ability to add more as needed (Backus et al., 1956, p. 40). When the new version of FORTRAN, named FORTRAN II, was published in 1958, it made the connection between existing functionality and those added by programmers even more explicit by adding a new keyword “CALL”. When a programmer wanted to use a function in the library, they needed to use the keyword “CALL” and the language would handle the “transfer of control to the subroutine" named (*Reference Manual: FORTRAN II for the IBM 704 Data Processing System*, 1958; p. 15). When “called,” the subroutine would gain direct control over the running code, making a connection to past labor form present work. Future programming languages copied from the concepts of FORTRAN as they slowly spread through additional sources like teaching materials.

Thought of as the first textbook on computer programming, *The Art of Computer Programming: Volume I* was started in 1962 and first published in 1968. Between the introduction of FORTRAN in 1956 and other programming languages in the later 1950s, there was a need for a comprehensive overview of programming concepts. In the third edition of the book, Knuth (1997) focuses on subroutines as a central topic important to all programmers. “When a certain task is to be performed at several different places in a program,” explains Knuth (1997), “it is usually undesirable to repeat the coding in each place” (p. 186). Knuth (1997) explains how “[m]ost computer installations have built up a large library of useful subroutines, and such a library greatly facilitates the programming of standard computer applications” (p. 186). Within this “large library of useful subroutines,” as explained by Knuth (1997), the

execution of the code moves through multiple layers. Using a metaphor becoming common by the time of the original publishing of the book *The Art of Computer Programming: Volume I* in 1968, Knuth (1997) writes how the subroutines interacting with each other create a "stack." Newer programs are built on previous ones. When run, the older code itself might be reaching deeper into the past of the subroutine library available. Such an approach is created when "[o]ne problem leads to another and this leads to another" where past solutions can be combined, explains Knuth (1997) (p. 241). Across a "function stack," each part solves a certain, smaller problem. They "call," borrowing from the term popularized in FORTRAN II, down to the past while passing data "up" to the previous as it finishes its own execution (*Reference Manual: FORTRAN II for the IBM 704 Data Processing System*, 1958). To solve a complex problem, a programmer might incorporate many subroutines. These are stacked on top of each other with code from one programmer contacting the programs of others which, in turn, potentially contacts others. In phrasing borrowed from the FORTRAN II programmer's reference, these series of relationships "may be indefinitely expanded" by one function calling the next (p. 2). This creates not only a "library" of functionality available to new programmers, but inherently constructs social relationships through which future programs are built on the labor of the past.

The history of a software project can be better understood by investigating the libraries it "calls." This chapter, like the tracing of the concept of "function," begins with earlier references and their own histories. As explained by Knuth (1997), when functions are used in connection to each other, they create a "function stack," a structure where each part influences others nearby within its organization. In this chapter, the latest version of Snowman, 1.4, is explored through three lines of its code as the point where humans interact with the story format. As explained in the next section, not only does source code carry with it an understanding of how it can be used,

what functions it exposes for others, but the arrangement of a function stack itself is always inherently one where choices made in the past of one project affect the future organization of others. As is explained in more depth later in this chapter on a section introducing the associated research, the three lines from Snowman 1.4 were chosen for their connection not only to other software libraries, but with how a user interacts with them as well. The primary form of interaction for HTML is clicking hyperlinks, and the lines examined serve as the bridge through which those interactions are translated into story progression by Snowman. This chapter wraps up by reflecting on the importance of studying code as connected to a deeper history on which functions "call" to the past.

Software Foundations

Software is built on software. The different layers of subroutines create a "geology" of layers across different libraries. To investigate these layers requires understanding how digital works often contain collisions of the present with the past. In their introduction to the edited collection *Media Archaeology*, Huhtamo and Parikka (2011) describe the work of investigating digital media as research through which the "past is brought to the present, and the present to the past" (p. 15). To better understand digital media, researchers must "take [the object's] material nature into consideration" (p. 8). All media have materiality. They are accessed, interfaced, and used in different ways. The contact between systems points to how they can be investigated by examining what their interaction reveal about their interfaces at each layer. Marino (2020) calls for this form of investigation when studying code, writing of how "[s]oftware studies, platform studies, and media archaeology [. . .] can now work to strengthen one another" (p. 22). This echoes the work of Wardrip-Fruin (2011) in the edited collection on media archaeology. In a chapter named "Digital Media Archaeology: Interpreting Computational Processes", Wardrip-

Fruin (2011) suggests researchers move beyond the “surface of their projects” to dig deeper into digital systems (p. 320). Wardrip-Fruin (2011) lays out a process for investigating “its surface output, the data it employs, and the processes it executes” (p. 307). While not discussing functions directly, Wardrip-Fruin (2011) puts forth a useful approach to understanding the relationship one function has to others through the input, output, and processes of each as part of navigating a stack from the present into the deeper past of discovering, in the words of Marino (2020), the “partial” code visible in any layer of meaning as found in some code.

Parikka (2015) puts a conceptual name to the different layers of meaning found in media: strata. Building from the work of Foucault (1969) in *The Archaeology of Knowledge*, Parikka (2015) pulls from the introduction of the work and the call to study the “various sedimentary strata” of history. According to Foucault (1969), the history of “governments, wars, and famines” can be connected to the “history of sea routes” and other, details beyond the obvious ones more traditionally studied (Foucault, 1969). Instead of looking at the obvious, an investigation of a cluster of historical events should begin by posing a different question: “which strata should be isolated from the others?” (p. 3). From this base, Parikka (2015) builds on Kittler’s (1999) provocation that humanities researchers should, in the paraphrased words of Parikka (2015), “have a proper understanding of the sciences and engineering realities that govern the highly fine-structured computer worlds in which we live – without ignoring the fact that technical media did not start with the digital” (p. 2). For Parikka (2015), combining the concepts of Foucault (1969) with the importance of the materiality of media as found in Kittler (1999), there is a need for constantly considering the stratification of remediation, as presented by Bolter and Grusin (2003). Each form of digital media presents ways in which it reveals and hides its own past to older media (Bolter & Grusin, 2003). In *The Geology of Media*, Parikka

(2015) calls for a study of stratification, the points of “double articulation” of present to past and past to present, how an object is made from other objects, by starting from the digital present and digging into the deeper material past (p. 36). Work should start at the point of interaction, where human meets system, and then dig deeper across the passage of time to understand where the technical media became digital from its own past. For each stratum, research should seek to find how it reveals and hides the previous layer on which it was built.

This chapter is not strictly concerned with digital media. However, it does use media archaeology and its emphasis on digging into the past as part of using the conception of strata as presented by Parikka (2015). As explained in the introduction to this chapter, functions “stack” on top of each other. Soloman (2013), in fact, makes this connection explicit in the essay “Last In, First Out”. Soloman (2013) writes how the “function stack” is a useful way to understand digital systems because of the connection between interfaces as “technological circumstances” as affecting “[other] media in turn” (Para. 25). There is no direct access to the data; every presented interface is itself mediated. Graphical user interfaces, as well as those between digital systems, reveal and hide aspects of themselves between layers. For Galloway (2006), how data is accessed is governed by rules put into place as part of the construction of how its interface is programmed. When using a word processing program, for example, the editing program presents an interface to the data on which it is working. A user cannot manipulate the data directly and can only edit in ways the interface itself allows. Galloway (2006) raises the importance aspect of digital interfaces as being centered on control over how data can be accessed and through what methods. This connects to how Brown (2015) describes data as being accepted or not by programs. Each interface creates rules for how it knows what to accept, and what it will not accept, created in part by the presences and absences within its interface creating these implicit

rules (Brown, 2015). Bratton (2015) brings this point home by pointing to the importance of communication and the metaphor of the stack specifically. As one digital system "sits" on another, its own control adds to the previous layers, creating a "stack" of power relationships between one structure and the next (Bratton, 2015). Soloman (2013), echoing the two forces of remediation from Bolter and Grusin (2003) for digital media, calls this the "paradoxical arrangement" of the function stack: each level in the stack "open[s] up possibilities for new types of free creative action" while also "are always already constrained and preconditioned by the lower-level systems and infrastructure upon which they are implemented" (Para. 25). This paradox is the point of "double articulation" mentioned by Parikka (2015), and as affected by what each accepts, from Brown (2015), and control from Galloway (2006) and Bratton (2015). The stack is a structure of power and control made by different functionality "stacked" on top of each other and through which one level accesses the values from another. Soloman (2013) describes this as each level "serv[ing] a nondiscursive role to those above, and yet, a discursive role in relation to those below" (Para. 26). As the introduction to this chapter relates, the structure of the functions stack is a foundational one to programming and the study of code.

The research presented in this chapter embraces the metaphor of strata from Parikka (2015) as a useful lens to understand the relationships between layers of a function stack as building from the work of Galloway (2006), understanding how control manifests within communication; Brown (2015), accepting and rejecting of data across interfaces; and Bratton (2015), how the implementation of a stack encodes power structures. In the next section, three lines for Snowman 1.4 are explored in-depth. Following the suggestion established by the field of media archaeology to begin where humans interact with a system, these three lines are highlighted because they control the loading of a section of a story, what Twine names a passage,

based on a reader clicking on a hyperlink in a web browser. Because of the importance of understanding the effects of remediation, and specifically how stratification can happen with an interface, the context of each software library is discussed first before proceeding into how each level of the stack interacts as part of their own functionality. Based on the order presented by Wardrip-Fruin (2011), each function is examined in terms of its input, processing, and its output. As first Knuth (1997) hints at and Soloman (2013) declares, each layer in a stack affects those above it through a "paradoxical arrangement" (Para. 26). This requires understanding, as with the calls from Parikka (2015), to move from the digital present into the material past. For Snowman, this means beginning where it does: the processing of HTML data.

Function Stack as Strata in Snowman

When a Twine HTML file is loaded in a web browser, the JavaScript code of the story format is run. This first loads its own values and then looks for special HTML elements within the webpage and starts to perform the work of loading different values and preparing for the first interaction with a reader (Cox et al., 2019). In the story format Snowman, the first moment where a reader can interact with a story occurs starting at Line 205 in a file named "src/Story.js." This contains the first of several uses of what are named "event listeners." In the programming language JavaScript, code can wait for someone or something to take an action and then react in some way. For example, when a user begins to type in a search field in a web browser, JavaScript code was written with an event listener for the interface element. As a user types, the event undergoes what is called a "trigger" in a web browser and a function is run as a result. This approach allows JavaScript code to only react when needed through programmers adding event listeners for possible interactions and only running when the associated event happens ("Introduction to Events", 2023). Snowman completes all its internal loading and then begins to

wait on an event named “click” triggered when a reader performs a left click on any hyperlink. In Twine works, the primary form of communication between reader and the story format is through the clicking of hyperlinks. In Snowman, this functionality is provided starting at Line 205. This section begins with reviewing the software library providing the functionality connected to this event listener, jQuery, before moving to a second library, Underscore. Finally, this section will explain how the last line of code, containing only a single function, connects to the two previous ones. It is only through the last line that the meaning of the three lines changes to become its own stratum within Snowman.

Starting with jQuery

Snowman does not work directly with functionality provided by a web browser. Instead, this is provided by the JavaScript library jQuery, an interface for working with the lowest level structure in a web browser called the document object model (DOM). When an HTML file is loaded by a web browser, it creates an in-memory object version of the document file. JavaScript can interact with this object, and a web browser will translate the interactions, along with any possible changes to its values, to manipulate the visual representation shown on a screen ("Document Object Model (DOM)", 2023). jQuery was created to solve a problem with accessing the DOM of a web browser. While the model is often the same across web browsers, the interfaces for accessing and changing are sometimes very different. In the early 2000s, there were multiple web browsers with each following their own approaches to using JavaScript functionality such as event listeners. jQuery sought to establish its own standard across them. In the words of an announcement by its creator to other programmers in 2004, jQuery will “revolutionize[] the way you can get Javascript[sic] to interact with HTML” (*Resig*, 2004). One of the earliest versions of its website declares its design to “change the way you write

Javascript[sic]" (*jQuery: New Wave Javascript*, 2006). By using jQuery, the library could act as an interface to the other web browser functions. A programmer could write to the standard of jQuery and know their own code would work the same regardless of what web browser ran the project. This approach was so successful, in fact, by nearly twenty years after its introduction, jQuery was reported to be used on 94.3% of all websites where it can be detected (*Usage Statistics and Market Share of JQuery for Websites, March 2023*, 2023). In an apt description of the role jQuery began to see itself in across web development, a blog post from 2014 names jQuery as a "repairman for browsers" (*The jQuery Foundation and Standards*, 2014). While each web browser provides their own functions for accessing the DOM, jQuery often "sits" on top of these interfaces, providing its own functions for working with and changing the visual representation of a loaded HTML file. In this role, it became and remains easier to detect in existing code based on how it presents: its functions are accessed using the dollar sign, $.

As early as the first public version, jQuery began to use the dollar sign, $, in JavaScript (*jQuery 1.0 – Alpha Release*, 2006). This allowed developers to simply reference its interface using the dollar sign and a name of one of its functions. In many cases, jQuery usage can be detected by looking for this pattern of the dollar sign, signaling a likelihood of using the library within some code. In Snowman 1.4, Line 205 of the src/Story.js file begins with a dollar sign and a function named "on()". To provide a standard across web browsers, jQuery established the use of functions like "on()" to work across different web browsers. A programmer can prepare code to react when a user left-clicks on the webpage, what is known more technically as a "click" event ("Element: click event", 2023). Written in English, the "on()" function describes the following situation: "When the specific event happens, do this" ("on()", 2023). In Snowman, the event is a user clicking. This then triggers the second part of the explanation, "do this." On Line

205, the second part is another function containing its own instructions for what should happen next.

To help programmers describe their own functions, the programming language JavaScript contains the keyword “function.” Whenever a programmer wants to define a new function, they can use the keyword (*ECMA-262, 15th edition, June 2023*, 2023). Because the contents of one function might contain the same name of existing values, JavaScript includes an approach a programmer can specify the current function rather than any other existing values names from nearby structures: the keyword “this.” When the keyword is used in JavaScript, a programmer is specifying the current function or structure. It can be thought of as translated into English as “this object right here” rather than another one ("this", 2023). On Line 205, both keywords are found. First, the line begins with the “this” keyword, referencing the larger structure in which the line is found. This is referring to the name of the file, “story.js”, within which the larger structure named “Story” is found. To help programmers quickly identify files, a common naming practice is to name the file based on the largest structure found within it. In Snowman 1.4, its “src/Story.js” file contains the Story structure. On Line 205, the keyword “this” appears first followed by the name of a value, “$el.” As explained, the use of the dollar sign signals the use of jQuery. Here, it refers to data retrieved from an earlier use of a jQuery function in the file. The naming convention helps to retain, at a quick glance, where the data came from it is using. Expanded to include the new keywords, Line 205 becomes slightly different (Figure 3).

```
this.$el.on('click','a[data-passage]',function (){});
```

Figure 3. Line 205 of “Story.js” in Snowman 1.4 with “this” Keyword

There is one last detail within the processing of Line 205. As mentioned as part of its input, the use of the “on()” function uses three values: what event to listen for, how to filter these results, and what function to call when the event happens. The first input is the event “click” by name. In the next section, the second input will be explained. This leaves the last input: the function to run when the event is triggered. To prepare for moving toward Line 206, there is one last item to mention: the input of the function reacting to the event. In JavaScript, as run in a web browser, every event generates data. This includes information on where the event occurred, what triggered it, such as a “click”, and other information ("Event", 2023). This is needed for the function to react to an event, telling the function important information. To complete Line 205, one last item is needed, the data passed to the internal function (Figure 4).

```
this.$el.on('click','a[data-passage]',function (e) {});
```

Figure 4. Complete Line 205 from “src/Story.js” in Snowman

Put all together, the stratum of jQuery acts as an interface to the DOM of the Twine HTML data. Moving from the “bottom” to the top, there is the “on()” function as part of a larger structure named Story in the src/Story.js file. To help developers have an standard interface for working with events, the “on()” function acts as an interface to the DOM presented by different web browsers. When an event happens, this function checks for what it is listening for, “click” on this line, and then filters the results before finally calling a function with data created by the web browser. This is passed to the “on()” function when the event happens, giving it access to data from the web browser on what happened and where within the DOM. In the next section, Line 206 is examined as another point of intersection with a different software library,

Underscore. First, however, understanding Underscore and its positioning in Snowman begins with its intersections with the functionality surrounding how a story progresses in Twine.

Progressing a Story Using Underscore

When an author uses the publish functionality in Twine, their current work is packaged as a collection of HTML elements along with the currently selected story format. In HTML, each part of a webpage is divided into smaller unit called “elements.” These describe how the webpage is presented through the data each unit, element, it contains ("HTML: HyperText Markup Language", 2023). Twine publishes a story by encoding its data into HTML elements as part of a webpage where the story is a single element with each passage its own HTML element containing its data (Cox et al., 2021). When the resulting HTML file is opened in a web browser, the story format JavaScript code runs and then attempts to read the packaged Twine HTML as data values stored in the same file. After it prepares its own values, Snowman processes these HTML elements as a collection of values matching what Twine names parts of a story: passages. The Story structure in Snowman has a value named “passages” containing each passage within the story with its own data. This collection becomes very important, as it serves as an internal database of the story as divided across different parts. When story progression is about to happen, it is this database where Snowman turns to determine what should be shown next to a reader. This makes the accessing and changing of story data, both as part of this collection, and as perceived by a reader, an important part of how Snowman works. It is also central to the next line of code covered in this section. Outside of some special situations, passage content is not shown to a reader unless they click on a link, functionality explained in the last section as starting with Line 205 in the “src/Story.js” file in Snowman. In this section, the next stratum, the JavaScript utility library Underscore, is positioned between jQuery and Snowman functions.

Understanding Underscore, like with jQuery, begins with knowing why it was created and what problems it is attempting to solve. Much like jQuery, Underscore was created to supply its own standard set of functions to perform different tasks in JavaScript. However, while jQuery is focused on working with the DOM more directly as an interface across web browser differences, Underscore is a set of utility functions augmenting the existing data structures and functionality of the programming language JavaScript (Ashkenas & Gonggrijp, 2022). Because of its specific focus, Underscore is often paired with other software libraries. To prevent confusion, like with jQuery using a particular symbol, the dollar sign, Underscore matches its own name, the underscore symbol, "_." Underscore functions use the underscore symbol and then the name of the function. Because of its emphasis on utility functionality, Underscore also supplies an interface where the output of one function can more easily fed into the input of another (Ashkenas & Gonggrijp, 2022). When used within a project, it is not uncommon to see multiple functions connected as they send data to each other as part of a single line of code.

Like with jQuery, the introduction of Underscore also represents a particular point in web development and changes to the language of JavaScript. After the first version of Underscore appeared in 2009, usage of the library slowly grew in popularity (Ashkenas, 2009). In 2012, some of the developers working on the code split off a new project named Lodash as a "drop-in replacement" for Underscore (Dalton, Cambridge & Bynens, 2012). This created two parallel branches moving forward with each claiming a smaller part of the overall usage by 2023. Lodash at approximately 3.4% and Underscore at 8.8% (*Historical Yearly Trends in the Usage Statistics of Javascript Libraries for Websites, March 2023, 2023*). There is one more event Underscore and its sister project Lodash faced jQuery did not: changes to the JavaScript programming language. Underscore, like jQuery, has its own standard set of utility functions. However, the

introduction of the sixth version of JavaScript in 2015 also established a new historical precedent: JavaScript would be updated every year instead of every few years (*ECMA-262, 6th Edition, June 2015*, 2015). This meant utility libraries like Underscore and its sister project Lodash became less needed over time as parts of what they provided to programmers began to slowly be added to the JavaScript language every year. This places the greater importance of utility libraries like Underscore as part of a pre-2015 historical pattern of development in JavaScript projects.

Having established the history of Underscore and its placement within the larger narrative around changes in the programming language JavaScript, it is time to understand how it impacts Snowman 1.4. Underscore becomes involved as part of Line 206. In the previous line of code, data was passed to a new function. Line 206 uses this data with information on what happened and where it happened within the DOM to help figure out what to load next for a reader during a Twine story. How it does this is connected with the second input to the previously reviewed jQuery "on()" function. In the previous section, this input was ignored because it becomes more important in understanding how Underscore and jQuery work together. In the previous section, this input was described as filtering of the results. This is true, as the "on()" function in jQuery might create many event listeners depending on the amount of content and its organization. However, the second input part filters down all possible results into only those matching a certain pattern within the DOM. In Line 205, this filtering is used is to look for first hyperlinks, defined by the anchor HTML element, and then specifically to those who also contain an internal part labelled "data-passage." This appears as part of the combined pattern of "a[data-passage]." This filtering is important because of how Snowman works: it encodes the story destination of a hyperlink when showing a new passage to a reader (Figure 5).

```html
<a data-passage="Connection">Connection</a>
```

Figure 5. Example use of "data-passage" encoding in Snowman 1.4

Line 206 begins with event data passed to it. When an event happens in a web browser, it is often connected to a location in the DOM called its "target" ("event.target", 2023). This the element, unit within the HTML, where the event happened as determined by the web browser. For example, a user might click on a button or a text field. In each case, the specific element would be the target of the event. When a reader clicks a link, the target of the event is the hyperlink element, the HTML anchor, itself. As passed to the inner function, this element is part of innermost part of Line 206. Based on the passed element, the closest one containing "data-passage" is found. In jQuery, the "closest()" function performs this search ("closest()", 2023). This usage on Line 206 finds the name of the destination passage from the target HTML element (Figure 6).

```javascript
$(e.target).closest('[data-passage]')
```

Figure 6. Event target conversion and search in Snowman 1.4

Because data is stored as part of the anchor element, the "data()" function in jQuery is used to retrieve it ("data()", 2023). On this line, the value is based on the closest element with "data-passage" starting from the target of the event element. Put together so far, the processes in English can be described as "Using jQuery, based on the HTML element within the larger

document-object, search both it and any neighbor elements for one containing ‘data-passage.’ Once found, retrieve its current value (as placed there by Snowman)” (Figure 7).

```
$(e.target).closest('[data-passage]').data('passage')
```

Figure 7. Retrieval of data from event target in Snowman 1.4

There may be occasions where an author has tried to use certain symbols in English in the names of passages that may have certain undesired meanings in HTML. To protect against these combinations of symbols causing problems, the result of the jQuery function is then fed into the use of the Underscore “unescape()” function. This function performs the action of searching for and then providing what it known as an “escaping sequence” by converting certain sequences of symbols into something web browsers can understand ("unescape()", 2022). This last processing is done on the result of the jQuery “data()” function. Explained in English, these new functions add to the description as the following: “Using jQuery, based on the HTML element within the larger document-object, search both it and any neighbor elements for one containing ‘data-passage.’ Once found, retrieve its current value (as placed there by Snowman). Using Underscore, process the name of the passage found” (Figure 8).

```
_.unescape($(e.target).closest('[data-passage]').data('passage'))
```

Figure 8. Escaping of data passage result based on jQuery and Underscore

The processing of Line 206 begins with data given to the internal function. Among its data is a value named “target” with the name of the HTML element where the reader clicked.

This is processed by jQuery using its "closest()" function, finding an element nearby to the "target" containing "data-passage" data. Next, the "data()" function is used to retrieve this data from the "data-passage" value within the HTML element. Finally, this resulting value is given to the Underscore "unescaped()" function where it processes the value and makes sure it is safe for use with Snowman. After all this happens, one more function is used: "Story.show()." However, the name "Story" does not appear on this line. Instead, the special keyword "this" is used again. As previously explained, the "this" keyword in the programming language JavaScript allows a data structure to refer to itself. Like with Line 205, Line 206 uses the same keyword to refer to the larger structure of Story. The more complete Line 206 shows these changes (Figure 9).

```
this.show( _.unescape( $(e.target).closest('[data-passage]').data('passage') ) );
```

Figure 9. Completed Line 206 of Snowman 1.4.

In the completed line, when a reader clicks a hyperlink, Snowman processes HTML data, checks its internal database of passage data, and then attempts to show a reader the next section of the story. Within this processing is a mix of jQuery, Underscore, and Snowman functions, each connected to and passing data to the next. Within a single line of code are multiple functions stacked on top of each other to receive the data of an event and use its values to find the next part of a Twine story to show to a reader. In the next section of this chapter, the final line is examined. It is both separate and key to understanding how the three lines work together, providing a completion of the function beginning with Line 205 while also changing how all the lines are understood together.

Completing the Function

The previous two lines of code examined in this chapter added to the understanding of the Snowman story format. The first established the use of jQuery and the role event listeners play within the code by waiting for the reader to “click” and then reacting. The second line was more complicated with the intersection of jQuery and Underscore through looking for an HTML element with a particular value, parsing it, and then using the value to search for the next passage to show a reader using the “Story.show()” function in Snowman. Finally, we have reached the last line: Line 207. This last section re-connects the function loop started from the first line through the last line of this code. Beginning on Line 207, the use of an important function is included: “bind().” In the programming language JavaScript, as has been previously examined, the keyword “this” refers to the data structure in which it is used. This is always the rule except when it is broken using a special function, “bind().” When used within a data structure, its internal reference can be overwritten with another, allowing an internal structure to “see” an outer one that would otherwise be impossible ("scope", 2023). In Snowman, this is used to allow the inner function to be able to “see” the larger data structure Story and its functions, something not allowed by default in JavaScript to prevent potential problems like values having the same names and creating confusion. Like with jQuery and Underscore, this creates an internal stratum as a source of functionality. This also enables functionality used in the previous line, access to the “Story.show()” function. As part of Line 206, the exact usage is “this.show(),” but the “show()” function is not part of the inner structure. It is part of the larger Story function. As with untangling the connections, understanding this line of code begins with isolating its historical context before moving through its input, processing, and output (Wardrip-Fruin, 2011). Without

understanding this last line, the entire structure carries a different and potentially conflicting meaning.

Line 207 includes only a single function, “bind().” However, this function has profound implications for both Line 206 and Line 205. The internal function’s reference to itself is overwritten, connecting it to the larger Story structure (Figure 10).

```
this.$el.on('click', 'a[data-passage]', function (e) {
    this.show(_.unescape($(e.target).closest('[data-passage]').data('passage')));
}.bind(this));
```

Figure 10. Lines 205 - 207 in “src/Story.js” of Snowman 1.4

It may seem strange to have an entire section of this chapter dedicated to a single function, but without the use of “bind()” function, Line 206 does not work. Line 207 cannot be disentangled from Line 206 without significantly re-writing the code. The “Story.show()” function cannot be accessed inside another function without being able to “see” the other structure. There is no clear-cut path between one line and the next within programming as these three lines demonstrate. There is always a hidden nature to the code within its own stack of meaning, and methodologies like media archaeology provide ways to see into how the code relates to each other by revealing its different layers in pursuit of the materialist end point. A complete understanding of Lines 205 – 207 is not possible without each line in turn. Without the use of “bind()” function, the use of Story function would not be “visible.” The same is also true of the use of value “$el” on Line 205 caught up in the ways the functions and data structures wrap around each other in JavaScript. The entire function stack originating within these three lines of code crisscrosses with other stratum through interlocking meanings as code from the present calls to the deeper past and back again. Echoing Parikka (2015) and a need explore

points of “double articulation,” the three lines of code examined in this chapter contain intersections of past and present at the same time as jQuery, Underscore, and Snowman functions all work together across their own different histories (p. 36). Each stratum explored creates a completed function placement in the larger “stack,” but also one matching how Soloman (2013) describes the “paradoxical arrangement” of one layer to the next. The structures of jQuery affect the use of functions of Underscore and how JavaScript itself works affects both software libraries through the use of the “bind()” function, changing how one structure can “see” another.

Conclusion

In the book *The Archaeology of Knowledge*, Foucault (1969) poses an important question at the center of this chapter: “which strata should be isolated from the others?” (p. 3). There is a considerable amount missing when viewed only “on the surface,” in the words of Wardrip-Fruin (2011), connected to only three lines of code. Beginning from the point where a reader interacts with the code, the use of the event listener in Snowman via the programming language JavaScript, the layers of meaning quickly twist into each other. Line 205 cannot work without Line 206 and Line 206 cannot work without the single use of the “bind()” function on Line 207, which connects back to Line 205. Yet, at the same time, the strata of jQuery and Underscore are entangled within the media archaeological examination of how a “click” event is processed by code written years apart from each other. When clicking on a hyperlink in a Twine story, a reader is not actively considering the decades of concepts and years of code translating their action into story progression for them. Yet, present code calls to a deeper past. Putting the central question another way, it might be better phrased as “Can programming stratum be easily isolated from each other?” The answer to this question, as this chapter hopefully points toward, is a definitive negative. An ever-important note comes from Marino (2020) on this: “visible code is

always partial" (p. 75). Despite this partially "visible" past of all code, the investigation of its relationships is vital to understanding not only the archaeology of a project in the form of digging into its layers but also the connections between structures. By beginning with how stratum relate to each other, and the ways in which they cannot be easily isolated, the more important questions become why they cannot be isolated and in which ways an examination of the remediated stratification patterns can help us understand the ways in which software build on each other. The present of Snowman is affected by choices made in the past and which make up the history of the software libraries on which it depends to work.

It can be no mistake Parikka (2015) and Soloman (2013) arrive independently at the importance of, in the words of Huhtamo and Parikka (2011), how an investigation of structures helps to see how "[t]he past is brought to the present, and the present to the past" (p. 15). Within media archaeology, the work of Foucault (1969), examination of strata, and Bolter and Grusin (2003), remediation, point to how digital media have a material past in which they, in often both equal measures, point to the partially visible past and hide evidence of it at the same time. By investigating these strata, a connection can be found through how the material past is echoed in the technical present; nothing is truly lost if references to it remain. Through the digital interfaces to the media, these references echo from the past to the present. For Soloman (2013), in considering the structure of the function stack, this same relationship becomes the use of the term "paradoxical arrangement," connecting to the work of Bolter and Grusin (2003) while also being in conversation with Galloway (2006), Brown (2015), and Bratton (2015) on how control and power become encoded in how the structure of a stack is created and maintained. One level of a stack influences the next. When it comes to programming, these interconnections between them often "hide" what Soloman (2013) calls "creative action" between one layer and another (Para.

25). Depending on how the access is created, it might not be possible for one layer to even communicate with another's values, as in the case of how this can happen in JavaScript. Yet, source code can always be studied through its input, processing, and output to begin to better understand its relationships to the past on which it was built (Wardrip-Fruin, 2011). Through these investigations, the greater interconnectivity of the past can be found entangled in the present.

Software is built on software in the same way media is built on media. The fields of critical code studies and media archaeology encourage us to investigate how the materialist past connects to the technical media present. In the case of Snowman 1.4 more specifically and JavaScript more generally, the project and language is in conversation with past languages and their own concepts. If there can be said to be a "materialist" past of programming languages, it is connected to the history around the early work to help create "automatic coding" in FORTRAN and other languages in the 1950s. Actions taken decades before in naming conventions and concept organization still affect the present. This chapter highlights the need for greater research into the history of software libraries projects rely on and how their own functions influence each other. This requires investigating the software libraries and their own histories for how patterns emerge, tracking how each layer of usage across a project connects to larger trends across past and present.

CHAPTER 3: TRUSTING THE PRESENT

On January 7, 2022, Marak Squires made a handful of changes in the source code of a project named *colors* that allows programmers to use different colors when creating text output for their programs. Through a few lines of new code, a programmer could use red text for warnings or green for when a program is operating normally. For projects like *colors*, it is very common to see small changes happen occasionally such as bug fixes or scheduled updates. When these happen, the version of the program is usually updated as well, signaling others they should use the new code in their own projects. In this one case, however, the changes were neither scheduled nor normal. Squires was protesting. Angered over how the project was being included in larger ones used by Fortune 500 companies to make millions of dollars while neither the project nor anyone who worked on it received any support or money, Squires crafted a targeted update to *colors* (Squires, 2022). Knowing most people would accept the new changes to *colors* into their own projects without question, Squires introduced an infinite loop into its code, updated the version, and tens of thousands of projects accepted the new changes immediately. Within hours of the update, those running the changes saw everything from minor problems to outright crashing across everything from small hobby projects to systems supporting millions of customers. It took most organizations multiple hours to revert the changes and clean up the mess (Roth, 2022). Rather than garner support for the *colors* project through this protest, however, the response from most was annoyance and disgust with Squires for causing the problem in the first place (Sharma, 2022). Many reported removing *colors* from their projects and anything else Squires might have worked on as well (Ropek, 2022). In place of gaining better support, Squires

had triggered a major reaction against all projects connected to their work. Trust had been permanently broken.

This chapter returns to two themes presented in chapter one: software carries cultural values (Manovich, 2013) and it can be best understood in a greater social context (Marino, 2020). As discussed in the last chapter, software projects "call" to the past in the current functionality visible to systems and people interacting with it. This chapter moves from the past to the present of Snowman 1.4 through making more visible two aspects highlighted in its introduction. First, applications of all sizes are built on the often-hidden labor of others. The labor behind *colors* is frequently included in everything from commercial software running websites to student projects. Yet, as the protest from Squires showed, many people either do not know of this hidden labor or do not care about the work on which their software relies on to run (Roth, 2022). Second, these smaller projects strongly affect other software to which they are connected. If changes in *colors* can crash software for millions of people, its influence is far greater than seems at first glance (Sharma, 2022). The dependency of one project on another can be described by the term "trust," but also goes beyond it as well. By choosing to become dependent on code written by another person, a programmer is putting their work into an implicit social relationship of control as well. As reviewed in the last chapter, other software libraries can strongly affect how code based on its functions can operate. This culture of trust and control is then echoed across all projects within its relationships. To help make sense of these relationships between software libraries, this chapter uses the metaphor of a "network" as defined by Latour (2007) as one in which the "work" of many forces all contributes toward each other. Building from this metaphor, a second one, "switch" is pulled from the work of Castells (2010). While Latour (2007) presents social relationships in which each party might have equal parity to each

other, this is rarely the case when examining expanded networks with their own sub-collection of forces. From Castells (2010) comes the metaphor of the "switch" within a network on which things depend and without which they cannot function, like the example of the project *colors* and how it was able to affect so many other projects. In examining the network of projects on which Snowman 1.4 depends, this chapter seeks to draw attention to how projects exist in connection to each other. This is then followed by an analysis of how the structure of the network as built on code affects the social relationships between programmers at the same time. The cultural values of smaller projects become part of the larger implicit contract of the collective itself.

Revealing Culture

In a chapter dedicated to analyzing a single file from The Transborder Immigrant Tool in *Critical Code Studies*, Marino (2020) describes the multiple software libraries being used by the tool. To cut down on the complexity of the chapter, Marino (2020) explains several lines of code as "[p]ackages and imports, libraries" (p. 74). These lines include references to software libraries as part of the Java programming language and other details not examined. Despite including all 597 lines of code from the "TBMIDlet.java" file being analyzed in the chapter, there are even more lines not included in the book that would, if included, inflate it by thousands or potentially tens-of-thousands of pages by themselves. Yet, without this other code not included in the chapter, the 597 lines of "TBMIDlet.java" file would not run. The single file is dependent on things like definitions of how data is stored in memory for different operating systems, accessing location information, and accessing media files on a computer defined across multiple other files and projects. Some of these are software libraries are part of the programming language Java used in the file, but many others are written by the authors of the project and others beyond even their work. Software is often dependent on other works in this way. There is a web of

connections between one project to others producing a series of social relationships between aspects of one to another.

The metaphor of a network is a powerful one for understanding how different relationships affect each other. In the book *Reassembling the Social*, Latour (2007) explains the different forces in a network through the concept of an "actor." These are, in the words of Latour (2007), "the moving target of a vast array of entities swarming toward it" (p. 44). To understand an actor, an analysis needs to begin with the other entities "swarming" it. At the same time, each different actor is performing "work," leaving a trace, on the others around it, establishing a relationship as the total, "net", of all these relationships. In a programming sense, the "work" in Latour's (2007) terms is seen in the effects generated by one project manifesting in another. In the example of the project *colors*, this is the addition of the infinite loop crashing many projects. Yet, in being affected by the "work" of one actor on another, there is a connection between them, creating a relationship. At one end of the network is a small change rippling out across the relationships and affecting seemingly disconnected parts at the other end. Within this pattern, there is obviously some actor within the network who can create major changes with small differences in how they operate. Yet, in how Latour (2007) explains networks, such actors cannot exist. To understand the importance of one actor gaining a higher degree of influence within a network of others requires adding another metaphor to the existing one provided by Latour (2007): switches.

Within any societal structure, there will always be some entities with more power than others. In identifying this aspect of all human endeavors, Castells (2010) writes in *Rise of the Network Society* of how there are networks of influences across power structures like economies. Yet, unlike how Latour (2007) explains networks as filled with actors affecting each other in

equal measure, Castells (2010) defines certain entities as “switches” or what is described as “privileged instruments of power” (p. 510). There are places within networks where power gathers and begins to affect the relationships around them. Any small change in the switches ripple outwards as they act as the gateway to other, more distant relationships of the network. Castells (2010) makes this point explicit in articulating switches as “shaping social structure” (p. 510). Not only are the switches gathering points of power within the network but can also have a profound influence on social relationships of people affected by the digital structures. By controlling what happens with a switch, both the virtual structure and the more physical relationships are affected. As with the project *colors* opening example, Squires brought attention to how powerful the package had become by disrupting many others with only a few changes. This was, up till that moment, only a potential power. While Castells (2010) was describing the perceived influence of power as being concentrated within governments trying to control access to information within the digital networks of countries, there are others who have made the connection between this same effect as the project *colors* example and within the realm of how software projects depend on each other.

In their research on open-source projects, de Souza et al. (2005) make the connection between the work of programmers and how its code is used with other projects. Not only is the software dependent on each other, the people involved are as well. de Souza et al. (2005) make the point of how the “social and organizational structure” of projects is “achieved through the technological organization of the underlying artifact, the software source code” (p. 204 - 205). The social organization of the project manifests in its code, echoing the label of “cultural software” from Manovich (2013), in how the culture of a project manifests in its code and influences the community around it. Such an observation also matches the work of Ofoeda et al.

(2019), who call for more study of the ways in which one software library can affect others. Ofoeda et al. (2019) suggest the connections between projects are mediated through the organization of their programming interfaces. By controlling access to and availability of its functions, for example, one party can influence another. A social network, for example, can control who can access its data by limiting who can use its functions or asking for money to provide certain levels of trust or access to select parties. Following the explanation of functions and their default ability to hide data from other data structures explained in chapter two, projects can restrict what is available for certain audiences. How programmers feel about other people accessing the data of a project can also produce a socialization effect on both the developers and what they produce (Ducheneaut, 2005). If, like Squires, programmers feel their work is underappreciated, they could stop working on it or even stage a protest.

This chapter uses the concepts of "network" from Latour (2007) and "switch" from Castells (2010). In the next section, these are applied to the dependency network of the current version of Snowman, 1.4. Through the application of these concepts, two aspects of the revealed relationships are examined: first, as both Manovich (2013) and Marino (2020) confirm, code is cultural and social. Code carries more than instructions for computers, it embeds the values of those who work on it. Second, the code structures found in programming are echoes of the social structures of the people who work on it. Through researching code, not only can glimpses of these relationships be discovered, so too can its cultural values and how they are translated into and out of the structures found in its source code. From de Souza et al. (2005) and Ofoeda et al. (2019) come an understanding of how programming interfaces affect not only other projects, but also the social relationships of projects existing in dependency to each other. As the example of the project *colors* points to, a change in the social understanding of a small project can be deeply

disruptive to the larger collective network surrounding its usage by others. Even smaller projects depend on the silent labor of many others to support it, as the next section examines. Snowman 1.4 is not immune to this fact.

Dependency Network of Snowman 1.4

To begin to understand Snowman, we must start with the programming language it is written in, JavaScript. As I examined in chapter two, early computers were defined with a need for "routine" operations. Programmers could then add more under the same category, "subroutines" (Mauchly, 1982). In the 2023 specification describing how the programming language JavaScript works, the term "function" is defined in connection to the term "subroutine" in the same way they appeared together in the reference manual for FORTRAN II in 1958: a function is a structure "that may be invoked as a subroutine" (*ECMA-262, 15th edition, June 2023*, 2023; Section 4.4.34). Just like the FORTRAN II reference manual, the programming language JavaScript provides some functions, but more can also be added (*Reference Manual: FORTRAN II for the IBM 704 Data Processing System*, 1958). For much of its life, JavaScript was closely associated with web browsers. JavaScript was first introduced as part of the web browser Netscape Navigator in 1995 to enable more interactive webpages (Krill, 2009). Using JavaScript meant also needing a web browser for much of its lifetime. In early 2009, however, this began to change. Ryan Dahl introduced what is known as a "runtime" for JavaScript. Instead of needing a web browser to interpret the code, a program named Node.js could provide "run" support for specific operating systems like Windows and macOS when it became "time" to interpret the language. This allowed programmers to use JavaScript outside of web browsers to create other types of programs with the runtime translating the language for them (*About - Node.js*, 2023). As many people at the time were familiar with JavaScript, Node.js became very

popular as a way to apply knowledge of JavaScript in web browsers to more programming-related tasks outside of web development (Dahl, 2010). Within months, many people had written JavaScript code using Node.js and soon others were using these new software libraries for their own projects. In 2010, a new companion program to Node.js was included with it, the Node Package Manager (NPM). Through using a website, developers could upload their code, named "packages", and others could download and use them. To speed up this process of finding existing code, NPM also embraced a special file for each project in Node.js named "package.json" with information about the project encoded in the JavaScript Object Notation (JSON) format for easier processing (*npm*, 2023a). When creating a new project, a programmer could create their own "package.json" file, list what other packages they wanted, and the companion program would retrieve them from the website and install them for usage.

As a project based in JavaScript, Snowman 1.4 has its own "package.json" file with its name, version, and other details. This information not only identifies the project as a package by itself, but also helps other JavaScript projects understand its information quickly (Figure 11).

```
{
    "name": "snowman",
    "version": "1.4.0",
    "license": "MIT",
    "repository": "https://github.com/videlais/snowman/tree/1.X"
}
```

Figure 11. Abbreviated "package.json" file for Snowman 1.4

Within most "package.json" files are two important sections, "dependencies" and "devDependencies." For most projects, the first section is usually short with a handful of to up to a dozen other packages needed. The second, however, might be considerably longer and require additional tools to compile and test the code to help other developers be able to reproduce the

same output (Goswami et al., 2020). The reason for this difference is in accounting for what is needed to run compared to what is needed to develop the code. Generally, what is needed for it to run is smaller because the second list handles the important tasks such as accounting for differences in web browsers and differences across hardware such as running on a mobile device or desktop computer. This is solved on the development side because, depending on where the code is designed to run, there might be a need to add more code for one operating system or another added or removed. For Snowman 1.4, this pattern is also used. The "dependencies" section of the Snowman 1.4 "package.json" file only lists four packages it needs to run (Figure 12).

```
"dependencies": {
    "jquery": "^3.2.1",
    "lz-string": "^1.4.4",
    "marked": "^0.7.0",
    "underscore": "^1.8.3"
}
```

Figure 12. "dependencies" section of Snowman 1.4 "package.json" file

While there are only four packages needed for Snowman 1.4 to run, the list needed for its development and preparing the code for usage with Twine stories is much longer. There are 20 packages listed as part of its development dependencies. These include packages needed to test its code, enforce certain standards, and make sure it can run the same across a wide range of web browsers and their different versions (Figure 13).

```
"devDependencies": {
    "browserify": "^14.1.0",
    "browserify-istanbul": "^2.0.0",
    "child-process-promise": "^2.2.1",
    "cpx": "^1.5.0",
    "cssnano-cli": "^1.0.5",
    "ejs": "^2.7.1",
    "eslint": "^6.3.0",
    "jasmine-core": "^2.5.2",
    "jsdom": "^15.1.1",
    "karma": "^4.3.0",
    "karma-browserify": "^5.1.1",
    "karma-coverage": "^1.1.2",
    "karma-jasmine": "^1.1.0",
    "karma-jsdom-launcher": "^7.1.1",
    "karma-source-map-support": "^1.2.0",
    "karma-spec-reporter": "0.0.32",
    "onchange": "^6.0.0",
    "rimraf": "^2.6.1",
    "shelljs": "^0.8.2",
    "uglifyify": "^5.0.2",
    "watchify": "^3.9.0"
  }
```

Figure 13. Development dependencies for Snowman 1.4

All the required packages are needed to work on the code for Snowman 1.4. This means not only the four packages included in the "dependencies" section of its package.json file, but also the 20 listed in its "devDependencies" section as well. Based on the addition of these two numbers, the assumption would be that only 24 packages are needed. However, all these packages also depend on others. As computed by NPM, the final count is 627 unique packages to download before work on Snowman can begin. While the total number may seem high, it is within the expectations of the number of dependencies for a Twine story format. The current versions of Harlowe, SugarCube, and Chapbook, other story formats also packaged in the current version of Twine along with Snowman 1.4, are dependent on 316 (Harlowe, 2023), 405 (Edwards, 2019), and 1573 other packages (Klimas, 2023a). To better understand these relationships, the tool NPM Graph is used. This online tool accepts a "package.json file" or name of a project hosted on NPM's website. From left to right, it then draws an arrow from the package listing the dependency to the package and then graphs its own dependencies, if any, as a

full visualization of all the relationships (Kieffer & Brigante, 2022). This creates, for even a smaller project like Snowman 1.4, a large and hard-to-read graph as some of the 627 packages are themselves dependent on inter-network connections.

To help make sense of, as well as highlight specific projects within it, the full visualization is included next and then three configurations within the network are explored. For the first, many arrows "swarm" on projects supported by a small group of people. It is these on which many other projects depend and see fewer changes as a result. The second configuration is larger projects with many changes but who are dependent on many others in turn. These projects often serve as smaller networks with projects depending on it and projects on which it depends relationships. The third configuration are those packages on the edges of the network. These hide a powerful aspect of the network. While seemingly disconnected from many, they hold more potential influence than seem at a surface level (Figure 14).

Figure 14. Full visualization of Snowman 1.4 dependency network

Exploring the Network

Within the defining of the term “actor” by Latour (2007), the words “swarming toward” appear. This helps describe the process through which connections emerge because of the “work” of one part “swarming” toward others, creating a relationship in the process (p. 44). Within the relationships of the dependencies in the visualization created by NPM Graph, an arrow indicates a dependent relationship from the left-hand side to the named package on the right-hand. In chapter two, the software libraries jQuery and Underscore were discussed in connection to the Snowman source code. These also appear within the network of packages for Snowman 1.4. However, while important for the internal working of Snowman, they do not have any projects on which they themselves depend. The relationship between Snowman to jQuery is a single line moving from one to another (Figure 15).

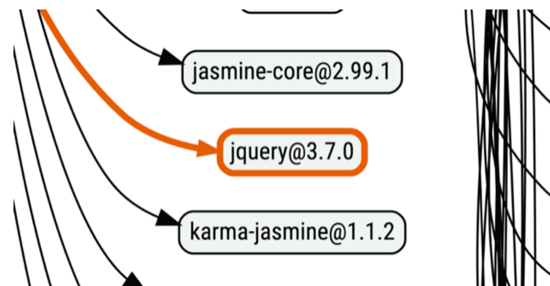

Figure 15. jQuery inclusion in Snowman 1.4 dependency network

Within the visualization, the more arrows pointing toward a package, the greater its seeming importance. With each arrow representing dependence, the more arrows, the more other projects depend on it. One such example in the Snowman 1.4 dependency network is the project *inherits* (Figure 16).

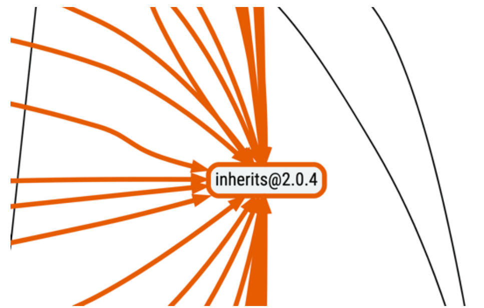

Figure 16. Project "inherits" within Snowman 1.4 dependency network

Over the lifetime of Snowman from 2009 to 2022, there have been 491 changes to its source code (Cox, 2017a). This is notable in relation to the project *inherits*, which reports only 32 changes across its 12 years of existence with nearly all these changes made by a single person, Isaac Schlueter (Schlueter, 2011). Yet, for seemingly unremarkable at first glance, the visualization of the Snowman 1.4 dependency network shows a very remarkable aspect of the project *inherits*: many other packages depend on its code. Its importance can only be seen through other projects "swarming" toward it, in the words of Latour (2007). Yet, the "work" of the collection of relationships is not distributed evenly. The labor of the project *inherits* is centered around primarily one person while other projects use this work indirectly as being dependent on other projects through which the project *inherits* is a dependency.

The project *defined* matches this same configuration within the network. An examination of the project shows only 28 changes across its 11 years with two people supplying most of these changes (Halliday & Harband, 2012). Yet, like the project *inherits*, an examination of the project by itself does not show its importance. Only through viewing the network of forces can the relationship between it and other packages most clearly be seen (Figure 17).

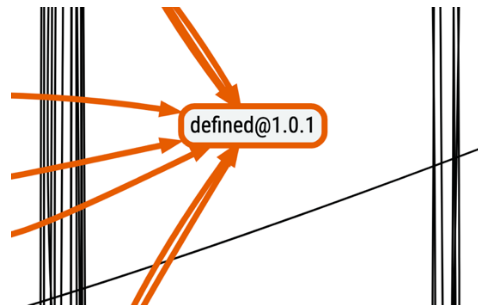

Figure 17. Project "defined" within Snowman 1.4 dependency network

The projects *inherits* and *defined* are examples of the "end" of relationships with no other packages on which they depend. They are also both examples of how the labor of only a handful is connected to the work of many others. Yet, this configuration is not only one to be found within the Snowman 1.4 dependency network. A second one also appears in places where a single larger project spreads outward to many others. For example, the project *browserify* is connected to Snowman through a single arrow from the left-hand side of the visualization but also shows many arrows pointing away from it (Figure 18).

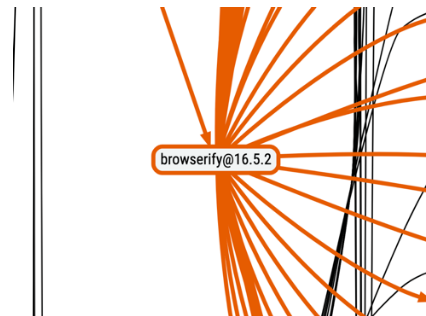

Figure 18. Project "browserify" within Snowman 1.4 dependency network

Unlike the 32 changes of the project *inherits* (Schlueter, 2011) and 28 of the project *defined* (Halliday & Harband, 2012), the project *browserify* shows a total of 2,290 changes across contributions by 187 different people. Part of this greater interest might account for its description of defining a way "to organize your browser code and load modules installed by

npm” (*browserify*, 2010). As explained as part of the introduction of the Node.js runtime, the program allows developers to separate JavaScript from web browsers. Yet, as the popularity of this package shows, many developers have used this separation to develop for web browsers even when not needing them to run JavaScript code. Within Snowman, the project *browserify* is used for this exact same purpose. While the project is designed for Node.js, it is used with other code within Twine as run in a web browser.

A second project matching the same configuration of *browserify* is the project *cssnano*. It too has a single connection from Snowman but is dependent on many other projects. However, unlike the project *browserify*, the project *cssnano* is, as described by its programmers, is “built on top of the PostCSS ecosystem” (*cssnano/cssnano*, 2015). Examining its own dependencies shows this same relationship with the project *cssnano*: other projects on which it depends use the “postcss” prefix, including the project *postcss* itself (Figure 19).

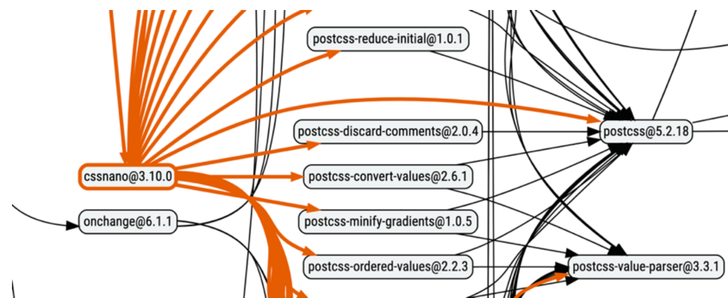

Figure 19. Project “cssnano” within Snowman 1.4 dependency network

There is one last configuration present in the Snowman 1.4 dependency network. So far, the projects *inherits* and *defined* represent those on which many depend but have seen fewer changes. On the other hand, there are those projects with a larger number of changes, but which are dependent on many others in turn, the projects *browserify* and *cssnano.* The last configuration represents those found on the right-most edge of the visualization. These are, like

the first configuration, those at the "end" of a network of forces, yet, at the same time, seemingly have a smaller influence at first glance. The first highlighted package matching this configuration is the project *is-arrayish*, found at the top-most of the visualization with only a single project dependent on it (Figure 20).

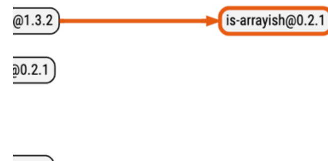

Figure 20. Project "is-arrayish" within Snowman 1.4 dependency network

Examining the project *is-arrayish* shows a configuration within the network like the projects *inherits* and *defined*. With only 28 changes across eight years, it only has four people who have contributed to its code (Qix, 2015). Yet, within the arrangement of the Snowman 1.4 dependency network, it does not have the same placement as the projects *inherits* and *defined* with many arrows pointing toward it. Regardless, its importance is not to be mistaken. Moving backward from *is-arrayish* shows a chain of connections leading back to Snowman itself across multiple projects. Even if not a point where many connections converge, Snowman still requires it because it is a dependency many relationships away as an outward edge of the network.

There is one more package to be examined, the project *colors*. The inclusion of the project *colors* in the opening of this chapter was not a coincidence. The package also appears within the Snowman 1.4 dependency network itself. It, like the project *is-arrayish*, at the end of a series of relationships (Figure 21).

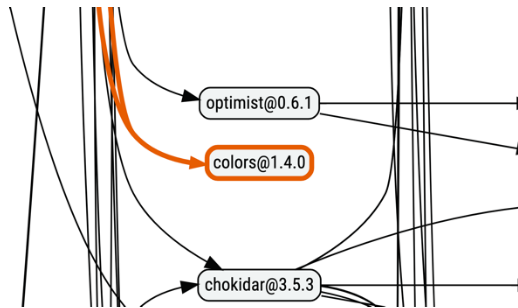

Figure 21. Mention of project "colors" within Snowman 1.4 dependency network

Unlike the projects *inherits*, *defined*, and *is-arrayish*, however, the project *colors* has seen 259 changes with the latest being the protest code added by Squires. No other changes have been made since January 7, 2022 (Squires, 2022). It remains frozen as it was when the protest by Squires occurred. Yet, despite this, it appears in the Snowman 1.4 dependency network and is a part of other code. Even if, as reported on, some programmers began to remove it from their own projects, the "work" continues (Ropek, 2022). As a comment on the code over a year later shows, some people remain completely unaware of the earlier protest (Chouhan, 2023). It remains part of the network.

Trust Switches

This section reviewed six different packages across three different configurations within a single Snowman 1.4 dependency network. While only brief details such as the people behind them and the number of changes over time were noted, none were singled out as more important than others. This was a purposeful move. The tool NPM Graph helps to show the relationships between projects such as *inherits* and *defined* being a focus of many others with the projects *is-arrayish* and *colors* only with one other project each dependent on them. Yet, a question remains: which of these might be thought of within the metaphor of a switch as defined by Castells (2010)? At first glance, a case could be made for the larger ones. The projects

browserify and *cssnano* could seemingly approach the potential for, in the words of Castells (2010), "shaping social structure" (p. 510). Dozens of people have contributed to both, and they are dependent on many others. Yet, the chapter opened on the project *colors* for a reason. Despite a lack of connections within the Snowman 1.4 dependency network, the protest by Squires was still able to affect many others. It clearly held great power over the networks in which, at least before January 7, 2022, it was connected. The same could be explained about the project *is-arrayish* and the series of connections leading back to Snowman. Should a person want, they could stage a protest using the project *is-arrayish* and achieve a similar result to the one done with the project *colors*. Given this potential, this chapter argues something different: all six packages are switches. The visualization hides an important aspect of their relationships as reminded by Ofoeda et al. (2019) and Ducheneaut (2005): code dependency is social dependency. In any case where one project is dependent on another, it inherits some of its social values through being dependent on its code and programmers for the project to run. Entangled among the relationships are aspects of control, trust, and labor.

The protest by Squires points to the inherently social context of the relationships between code packages. The Snowman 1.4 dependency network is not strictly one of programming dependency, it is also one of social relationships. Through studying the socialization of open-source projects, Ducheneaut (2005) presents research on the ways in which members of programming communities become more involved. Over time, the "progressive integration of new members" is achieved through engagement with the social and technical aspects of a single project (p. 328). It is through these contributions that they become part of the socialization of the community and the other packages on which it depends. The longer people participate at "higher" levels of involvement, the more likely they are to replicate its social structures and

reinforce the implicit social contract around its work. de Souza et al. (2005) makes this same point. The social organization of a project becomes mirrored in the "underlying artifact, the software source code" (p. 205). This is then echoed back at those contributing to the project, completing the circle of the social appearing in the technical and the technical, in turn, influencing the encoded values of the project, as Manovich (2013) explains. Social values are inscribed into a project, and in turn, the project influences the social values of other systems and people who interact with it. For as much as the Snowman 1.4 dependency network is a collection of one programmer trusting another, it is also built, at least to some degree, on the labor of the few. The higher number of connections yet fewer people contributing to projects like *inherits* and *defined* show a large part of the Snowman 1.4 dependency network relies on less than a dozen people despite being hundreds of different packages. At the same time, should any of these fewer people, like Squires, become disillusioned with their unrecognized labor, they have a much higher degree of control created through the implied trust social value layered onto it from those depending on it for their own projects. Those on the "edges" of the network must trust certain projects because the software they use trust others, propagating trust as a key social value spread by the network of software dependencies.

Along with encoded trust, invisible labor is also shared across the network. The projects *inherits* and *defined* are part of a larger trend in open-source communities. In their survey work across over 10,000 coding projects, Ghosh and Prakash (2000) report of how the "top 10 authors alone (0.08% of the total) are credited for 19.8% of the code base" (Para. 9). They continue, writing on how the "top 250 authors were credited with participation in over five projects, and the vast majority (over 77%) of authors were only involved in a single project" (Para. 10). Clearly, as shown in the Snowman 1.4 dependency network much of the labor rests on a smaller

number of people. This also matches with the research of Maass (2004) on a single programming community with "only 6% report frequently and 20.6% offer patch contributions periodically" (p. 67). This is also confirmed by previous research of Hertel et al. (2003) where greater involvement was based, in part, on a higher degree of trust given to both the work of the person and their standing in the community. Hertel et al. (2003) write of how those trusted "indicated higher concerns for reactions of significant others as well as higher pragmatic interests in improving the quality" of the project yet were a smaller percentage of the overall population (p. 1171). The greater the trust, the more labor was incorporated into projects. The Snowman 1.4 dependency network showed this as well. Many projects sit either on the far edges of the visualization within chains of relationships back to Snowman or, like the projects *inherits* and *defined*, as major points of convergence of many others. Yet, the number of people contributing to code on which many other packages depend is quite small compared to the larger projects. This same few-supporting-many pattern is also backed up by cultural studies research into fandom and fan-related labor such as the research of Stanfill (2019) and Salter and Stanfill (2020) where "fans" of projects contribute to and build on its influence through unpaid and frequently exploited labor through which larger organizations, like larger coding projects, are supported. As pointed out by Hertel et al. (2003), trust and labor are deeply encoded in these networks. The greater likelihood of a person to follow the social contract of a community, the more welcome they are. At the same time, the greater levels of trust granted also come with more unrecognized labor for the single project and the larger network of relationships to other projects to which it is connected.

Conclusion

This chapter opened with the story of the project *colors* and the political act of Squires. On January 7, 2022, Squires created a major incident for thousands of projects by changing their code to create an infinite loop (Squires, 2022). This action was a protest. As demonstrated through the act, there is no process in place to force or even incentivize large companies to support open-source code (Roth, 2022). In doing this, Squires drew attention to the ways in which changes to small packages can have major impacts on the larger network of relationships when considering the dependencies of different projects across time and space. By crashing everything from hobby to large, sprawling projects, Squires was able to not only point to the importance of the network of forces composing the "work" of their relationships as the dependent of distant dependents, but, to build from the work of Castells (2010), also pointed to the ways in which not only do the seemingly smallest projects have the most "weight", but how changes to their code can affect distant software packages through their acting as a switch for massive networks of nodes with no seeming connection to others at first, human glance. By drawing attention to ways in which major companies were profiting from the free labor of small projects, Squires was able to show how the switches of the network of dependencies are not always larger projects, but nearly always based on the often uncompensated work of fewer people (Ghosh & Prakash, 2000). These are often, as the examples of this chapter pointed out, the smallest projects with a handful of maintainers whose code have existed for a decade or more and act as foundational forces within the often-hidden network of dependencies of code within the space. Building from how Latour (2007) suggests networks work, the creation of relationships point inward to the small packages from the larger ones, creating the relationships

while also, from Castells (2010), having profound social impacts beyond the networks themselves when a disruption happens.

Brought into the examination of the Snowman 1.4 network of dependencies, the often-hidden nature of these relationships became clearer. When Marino (2020) writes, "the visible code does not reveal all," it describes these relationships (p. 75). For Snowman 1.4, the invisible code was able to be seen when visualized as a network of actors starting from the project and extending out multiple levels until evidence of the switches within it became more obvious based on the traces of "work" between the larger to smaller packages within the network. Now "visible," these sprawling relationships create dependencies where, as was noted in the chapter for projects like the projects *inherits* and *defined*, there are only a single maintainer per project and code that has seen very little changes over a decade of updates despite their increased importance and implicit trust (Hertel et al., 2003). Yet, more work needs to be done to understand the distant history and dependency of projects as an investigation of its code and social relationships. Not only, in the work of Manovich (2013) and de Souza et al. (2005), does the dependency of software connect to the social values of a network, ignoring the effects of virtual software networks can have profound issues. While the example of the project *colors* was a wakeup call for many developers within the JavaScript community in early 2022, it was not the first nor the last time a small package had or would make a major ripple (Alfadel et al., 2022). Others have used the same method to attack the larger networks to try to impose their own control by breaking the trust of the network.

In 2019, a group of researchers studied eight years of packages on NPM and reported on major vulnerabilities, noting previous examples in 2016 and 2018 where it was possible to "directly or indirectly influence" up to 100,000 other packages with small changes in targeted

code many other projects depended on to work (Zimmermann et al., 2019; p. 996). In 2021, multiple parties reported working on what has become known as “dependency confusion” or “NPM substitution attacks” where small changes are introduced to packages during search attempts to its database or during transit from the service to developers when downloading files in attempts to steal information, break into systems, or otherwise cause damage (Birsan, 2021; Goodin, 2021). This problem has become serious enough the NPM documentation specifically mentions it and how to combat it (*npm: Threats and Mitigations*, 2023b). What is not “visible” is increasingly not only of historical importance for the running code, telling a greater context of what the code is dependent on and needs for its development and running, but also increasingly the source of attacks as the network of dependencies comes under influence by the small switches within the larger, sprawling networks of projects dependent on each other and operating, for the most part, outside of human conception with tools like visualizations of the dependency graphs needed to understand the code and how it relates to each other.

Code dependency is both textual, how the code is written and what parts of another application programming interface it may use, but also shaped by the social forces and implicit trust of the dependency of one project on another. Snowman 1.4 is but one of many existing projects with a connection to over 600 others. It is strongly affected by changes in these other packages and especially within the most highly connected points of its own dependency graph. At the same time, Snowman has also been used by many people to create their own Twine stories, passing on this social values from the network into the interface and metaphors it presents to authors (Cox, 2020a). In the same way Manovich (2013) and de Souza et al. (2005) recognize the influence of the digital on the social, it can be no mistake that Castells (2010) defines the concept of “switches” as “shaping social structure” (p. 510). As the next chapter will

explore, other story formats have been based on Snowman, putting it within a different network of influences and spreading its own social values into the future.

CHAPTER 4: REFLECTING ON THE FUTURE

In early 1993, programmers at the National Center for Supercomputing Applications (NCSA) created a program whose impact is still being felt decades later. Named Mosaic, this graphical web browser would go on to have a profound effect on how people accessed webpages (Andreessen & Bina, 1994; *The Future Frontier: Computing on NCSA Mosaic's 10th Anniversary*, 2009). Having seen a prototype of one of the first graphical web browsers, Erwise, created by graduate students at Helsinki Technical University in 1992, the programmers at NCSA began to create their own (Berners-Lee, 1992). By early 1993, they had named their application Mosaic. Seeing the commercial potential of a graphical web browser for an upcoming version of their operating system, Microsoft bought the rights to the name and source code of Mosaic in 1994 to help create a new project they named Internet Explorer to be released the next year. Rather than, in the words of McCullough (2018), try to improve on the Mosaic code, the team behind Internet Explorer was told to "follow the traditional Microsoft game plan: the first version would be a copycat product that didn't have to be great; it just had to be good enough" (p. 49). At the same time, some of the programmers behind Mosaic, now without the rights to their own code, began working for a company initially named Mosaic Communications Corporation. This naming led to legal trouble with the rights to the name "Mosaic" having been sold, so the new company changed its name to Netscape and pushed to beat Microsoft to having a new commercial graphical web browser out on the market. In 1994, the re-named Netscape released their web browser, Navigator (McCullough, 2018; Ryan, 2010). By 1995, what was later labelled the "browser wars" had begun as Microsoft's Internet Explorer and Netscape's Navigator added different features to constantly compete with each other (Ryan, 2010). It was during these "browser wars" in which web technologies such as JavaScript, Cascading Style

Sheets (CSS), and the use of software platform Flash began to flourish as the two companies embraced new approaches to deliver interactive content using web browsers (Sink, 2003; Salter & Murray, 2014). Every war has its end, however. Nearly a decade later, the commercial company Netscape had folded, giving birth to a new open source web browser built on the same code named Firefox as part of a new organization named Mozilla managing its rights in 2002 (*History of the Mozilla Project*, 2023). In 2022, 27 years after its introduction in 1995, Internet Explorer would also see its own end (Woods, 2022). Nearly 30 years after the introduction of Mosaic, its "children" would end their "wars" as each had been replaced by other, newer applications serving the same purpose.

Drawing on the same historical trajectory of how the work behind the web browser Mosaic became the much more well-known applications Netscape Navigator and Internet Explorer, this chapter presents work around Snowman and two of its much more well-known "children": the story formats Adventures and Trialogue. In much the same ways the previous chapters examined the relationships between Snowman and other software as part of its past and present, this chapter moves into its future beyond its own source code. Despite previous research showing Snowman as a story format with fewer usages than others like Harlowe and SugarCube (Cox, 2020a), a catalog of story formats for Twine names it directly as the influence of many others (DeMarco, 2018b). Although not updated since 2018, a second collection of over 30 story formats for Twine include the tag "basedOn" naming Snowman for many of them, representing some 20% of all known story formats at the time (DeMarco, 2018a). Yet, as noted in the introduction to this dissertation, Snowman is not a focus of either Ford's (2016) or Salter and Moulthrop's (2021) books. Nor does it appear in Baccaris' *The Twine® Grimoire* collections (Baccaris, 2020, 2021). To help explain why Snowman is, like Mosaic, the less-known "root" of

more popular “branches,” this chapter draws on two concepts: from literary studies the concept of distant reading with a second, named “macroanalysis,” from the work of Moretti (2013) and Jockers (2013). From software studies, the concept of “static analysis” is used to explain the basis of looking at code as a “static” text in which its history and changes over time become important considerations to study (Rival & Yi, 2020). Like with distant reading, static analysis seeks trends and patterns in source code specifically to find the differences between versions of code across time within a project (Tornhill, 2015). Both approaches share a focus on text as a site of study where documents can be processed into quantitative patterns around particular usages, allowing for an easier comparison of multiple sources of input. In this chapter, distant reading is paired with a close examination of, first, the naming and code structures found in Snowman before moving to using a text mining tool to help drive a later closer examination of how different instances compare to one another across projects (Jockers, 2013). By moving from larger patterns to smaller examples, this chapter seeks to explore how the concepts Snowman appear in its “children,” how these changes match the expectation of the other story formats, and what they might also show about the future of Snowman outside of the original project.

Coding Crimes

In the forward to the book *Your Code as a Crime Scene*, Michael Feathers writes how it is “easy it is to look at code and think there is nothing more than what we see” (p. i). For the author of the book, Tornhill (2015), this is repeated on the very first page. In the opening paragraphs of *Your Code as a Crime Scene*, Feathers (2015) relates how when people only look at “what’s visible in the code,” they will “miss a lot of valuable information” (Tornhill 2015; p. 1). To better understand a project, explains Tornhill (2015), its code must be treated as a text with a history containing “evidence” of its authors and their changes. This focus on the code as a

point of investigation has a long history. Sallis et al. (1996) connect some of the earliest work on what is named “software forensics” to researchers studying security issues. Sallis et al. (1996) explain how software forensics experts sees as their role the determination of “authorships” based on “the premise that authors develop a style and approach that is identifiable” (p. 481). By examining the “layout”, “style”, and “structure” of code in a project, Sallis et al. (1996) suggest it can become possible to classify code as originating from the same person or project based on the textual patterns present in each. In the work of examining projects as “crime scenes,” Tornhill (2015) describes this method as “static analysis” based on existing security practices. Rather than treat the code as “dynamic,” running on a computer, the project is considered a complete text in and of itself where the static analysis is dedicated to understanding “the impact [. . .] code has on the machine” because of its mixed audiences of “machines and humans” (p. 5 - 6). In their book *Introduction to Static Analysis*, Rival and Yi (2020) use the same explanation as Tornhill (2015), noting the use of “dynamic” as “[taking] place while the program computes, typically over several executions” whereas static analysis is done “independently from any execution” (p. 26). Based on the differences between static analysis and trying to understand the code while running, both Tornhill (2015) and Rival and Yi (2020) note that while static analysis cannot give a complete accounting of a program without it running, examining the source code of a project can give insights impossible to gleam any other way. Rival and Yi (2020), like with Tornhill (2015), also note the importance of intention when using static analysis. As with any textual investigation, as Rival and Yi (2020) explain, the goal of the analysis is as important as the “choice of abstraction”; the relationships studied will affect what can be found from such an analysis (p. 444). While static analysis of a large collection of code can reveal much, it can also hide the finer details with an emphasis on major trends across many files. There must be some

balance between the patterns found and smaller instances within the work. Any larger trends in the programming studied must be balanced, in part, with an understanding of how they appear in practice within the code of a particular file or usage in a larger project.

In the opening chapter of the book *Macroanalysis*, Jockers (2013) writes of how the history of digital humanities has its roots in "humanities computing," the analysis of texts using digital tools (p. 1). Jockers (2013) suggests how there is room in literary studies to borrow from scientists in adopting the use of repeatable methods as part of quantitative analysis. When it comes to trying to read and comprehend thousands of works, a single human would struggle to remember all the details. The use of what literary studies names "close reading" cannot handle hundreds of works or trying to understand trends across many decades of publications. Jockers (2013) explains this as a major issue between wanting to understand large collections of texts, "big data," and the use of the close reading approach in literary studies: "big data renders it totally inappropriate as a method of studying literary history" (p. 7) Because of the time-costs in terms of years or decades for a single person to attempt to read thousands of works, the sheer impossibility of a one person both reading and retaining many connections between the works becomes increasingly impossible. Instead, building from the work of Moretti (2007) in the book *Graphs, Maps, Trees*, Jockers (2013) proposes the use of the term "macroanalysis" to define the use of larger-scale textual studies to understand not only the patterns across texts, but also the "systematic examination of data" beyond the interpretative act of reading, connecting into the importance of digital tools for understanding major trends (p. 25). Yet, Jockers (2013) does not propose to get rid of the use of the close reading method, suggesting the importance of a blended approach where digital tools can help identify "macro" trends while researchers can use instances found by data searches to understand the "micro" usages within them (p. 25). In the

chapter “Graphs, Trees, Materialism, Fishing” in an edited collection of responses to the book *Maps, Graphs, Trees*, Shalizi (2011) suggests a similar approach. Rather than taking on the larger-scale parsing of texts as a single point of research, Shalizi (2011) proposes the borrowing of another approach, “comparative method,” to understand texts highlighted by searches and text mining tools. Built on a foundation of comparative biology, Shalizi (2011) relates how instances found by searches can be compared in pairs or groups to better understand their relationships to some proposed origin or parent source (p. 140). When working on sources from a common source, the use of comparisons can show how each work has changed (Harvey & Pagel, 1991). By combining these two methods, larger-scale patterns can be found and then examined in a closer way.

Structures of Snowman 1.3

In the closing pages of the first chapter of the book *Your Code as a Crime Scene*, Tornhill (2015) concludes on an important point this section uses as a foundation: “when it comes to software design, there’s no tool that replaces human expertise” (p. 8). As Tornhill (2015) explains in a section named “Forget Tools”, the use of textual searching and pattern matching by themselves is not enough to gain a full insight into how code relates to each other. Because of naming changes and one file needing another to run “dynamically,” it can become difficult for text searching to understand the relationships between similar data structures across programming languages and formatting conventions (Tornhill, 2015; p. 8-10). This does not mean it is impossible to compare projects, but there is an inherit difficulty in reviewing data structures and programming patterns using exclusively distant reading tools. To help with this process, this section first covers the names of code structures found in Snowman 1.3 before then using these names to search against the first official version of the other story formats:

Adventures 1.0 and Trialogue 0.0.8. While the two previous chapters in this dissertation focused on Snowman 1.4, this chapter moves back a version to 1.3. The reason for this is because both of the other projects examined in this chapter are based on that version, not the work of the maintainer who followed after Klimas in 2019 (Cox, 2017a). As explained in chapter two, the programming language JavaScript also changed in 2015. With the move to Snowman 1.4, these changes were incorporated into Snowman and introduced new structures to the code. I selected Snowman 1.3 to avoid comparisons across versions of JavaScript.

As established in chapter one, Snowman was originally created by Chris Klimas and released alongside Twine 2.0.4 in 2015 (Cox, 2021b). While the project contains many files, three contain its core programming with multiple others serving as testing and packaging roles for the project, a common pattern found in JavaScript projects using the Node.js runtime (see chapter two). The three core files are found in the "src" folder: "story.js," "passage.js," and "index.js." For each file, the file ending, ".js", indicates they contain JavaScript code. Two of these files, "passage.js" and "story.js," represent its two core concepts: Passage and Story. The last file, "index.js," combines the two when the story format runs. In Snowman 1.3, the concepts of Passage and Story are represented as a collection of values and a set of functions. In the programming language JavaScript, the combination of values and functions is named an "object." Any values with names within an object are called its "properties" with its functions named "methods" (*ECMA-262, 15th edition, June 2023*, 2023). In Snowman 1.3, each file, "passage.js" and "story.js," contains their own properties and methods. For "passage.js," these are the following properties (Table 1).

Table 1. Snowman 1.3 Passage Properties.

Name	Description
id	Number representing passage creation order.
name	Label of passage.
tags	Passage metadata.
source	Content.

Each Passage object also has three methods: "render()," the processing of author content into HTML; "renderEl()," the parsing of special symbols into HTML elements; and "readyFunc()," the processing of JavaScript code within a passage. Each passage handles its own "rendering," a process through which HTML output is created (Table 2).

Table 2. Snowman 1.3 Passage Methods

Name	Description
render()	Translation of author symbols into HTML.
renderEl()	Converts markup symbols into HTML.
readyFunc()	jQuery helper function.

Similar, the "story.js" file has its own properties for its Story object (Table 3).

Table 3. Snowman 1.3 Story Properties

Name	Description
el	HTML element to show content.
startPassage	"id" number of passage to begin story.
creator	Name of program that created HTML.
creatorVersion	Version of program that created HTML.
history	Passages visited in story.
state	Collection of author-added data.
checkpointName	Name of current checkpoint.
errorMessage	What to show if an error occurs.
atCheckpoint	If story is currently at a checkpoint.
passages	Collection of Passage objects.
userScripts	Collection of Story JavaScript content.
userStyles	Collection of Story Style Sheets content.

Each Story object also has its own methods: "start()," the finding and showing of the starting passage for the story; "show()," the retrieval and replacing of currently shown content with a passage from the passages collection; and "render()," the calling of a passage's own

"render()" function by using its name. Snowman 1.3 also supports saving and returning to particular values saved to the URL using the "save()" function, the recording of values for possible URL encoding; "saveHash()," the creation of a URL with saved data; "checkpoint()," the saving of data in the web browser, and "restore()," the use of a encoded URL to re-create story data once saved (Table 4).

Table 4. Snowman 1.3 Story Methods

Name	Description
start()	Begins story.
passage()	Searches for name in collection of passages.
show()	Replaces current with new story content.
render()	Returns a passage's render() result.
checkpoint()	Records data to web browser's storage.
saveHash()	Converts author data into URL.
save()	Adds saveHash() value to URL.
restore()	Restores author data.

In the next two sections, each story format and its history is explained. This is followed by the results of my use of the names of the properties and methods from the "passage.js" and then "story.js" files in Snowman 1.3 as input for a text mining tool, Orange (Demšar et al., 2013). I performed a distant reading through arranging what Orange names a "workflow" through which the source code of each story format was used input and the names of different properties and methods used as part of a "keyword-based text document scoring" (*Text Mining - Keyword-Based Text Document Scoring*, n.d.). In this approach, each individual document is "scored" against each other, producing a table of results with the word count (frequency of term) within the document and as compared against the others entered. Because I was unfamiliar with the source code of each project before this work, this approach allowed me, as will be explained per story format, to quickly narrow down which files to more closely investigate by producing a list of possible files to study. As with the warning given by Tornhill (2015), "when it comes to

software design, there's no tool that replaces human expertise," these results were then verified and compared using a close reading approach (p. 8). While I have "direct experience" working with Snowman, I used the descriptions as a shorthand reference (D'Ignazio & Klein, 2020). They are also provided in this chapter to give the reader a better understanding of how each story format departed from the original functionality. As the story format with the largest obvious differences, Adventures 1.0 is explored first.

Adventures 1.0

The Twine story format Adventures reached its 1.0.0 version on September 15, 2017 (Kaczynski, 2017b). From the initial work beginning on August 7, 2017, the story format Adventures lasted until it was archived on May 13, 2020 (Kaczynski, 2017a). Snowman 1.3, on which Adventures 1.0 was based, did not provide an easy way to manage data outside of those functions available in the programming language JavaScript. To create a role-playing game, authors would need to program all the rules and structures they wanted by themselves. To make the management of character statistics and other expected aspects easier, Adventures 1.0 provided functions and data structures to carry out common calculations such as accounting for damage to a player character and managing in-game currencies. Later versions of Adventures also came packaged with icons and visual elements audiences might expect to see in a story mimicking common aesthetic trends from role-playing games for the 8- and 16-bit era of video game consoles (Kaczynski, n.d.). These helped authors present a unified visual appearance to their stories in Adventures.

As previously explored in chapter two and three, Snowman is written using the JavaScript programming language. Adventures, however, is not. It is written in a sister

programming language named TypeScript created by Microsoft in 2012 (Foley, 2012). Before the major change to JavaScript in 2015, programmers at Microsoft created a special version of JavaScript with the programming concept of “types.” When writing code, a programmer can specify the exact kind of data they want to use, its “type.” Each kind of data used in programming requires a different, specific amount of memory to use. For example, whole numbers such as five use a certain amount and are different than decimal numbers such as 4.15, which require other amounts of memory. In JavaScript, each value is dedicated the same, larger amount and then adjusted while the code is running if more memory is needed (*ECMA-262, 15th edition, June 2023*, 2023; Section 29.1). In cases where the larger, fuller amount was not needed, this can also lead to small amounts of wasteful memory cleaned up by the runtime. In TypeScript, the exact “type” is specified by the programmer, allowing the runtime to not need to adjust memory amounts as often while running. In general, the use of types can create faster programs, as a computer knows exactly how much memory it needs to reserve for the values based on the type specified by programmers in the language itself ("Everyday Types", 2023). Using TypeScript, programmers can write in one language and have it converted into JavaScript through a process named “transpiling” based on the term “compiled” describing the compiling of a file with software library code in order to run (Foley, 2012). This allows programmers to write in a different, sister language and still produce valid and frequently even more optimized JavaScript because of the original use of types. While JavaScript does not have types for its values, the “transpiled” code from TypeScript can often take advantage of ways to optimize or reduce the total number of values in ways a human doing the same work would not normally be able to do so ("Performance", 2023). To help differentiate the files from those using JavaScript, TypeScript files use a “.ts” file ending.

Based on my searching for the Passage-associated keywords of property and method names from Snowman 1.3 (see Table 1 and Table 2), Orange identified multiple possible files to investigate in the Adventures 1.0 source code (Table 5).

Table 5. Search Results from Adventures 1.0 Source Code

File Name	Word Count	Word Presence
defaultItems.ts	8.285714285714290	0.2857142857142860
Story.ts	3.142857142857140	0.2857142857142860
Character.ts	2.857142857142860	0.2857142857142860
Passage.ts	0.42857142857142900	0.2857142857142860
Item.ts	0.2857142857142860	0.2857142857142860
Choice.ts	0.14285714285714300	0.14285714285714300
Stat.ts	0.14285714285714300	0.14285714285714300

Reviewing the most likely files showed mixed results. Within the Adventures 1.0 source code, the words “tag” and “name,” properties found in the “passage.js” file of Snowman 1.3, appear many times, skewing the initial results. When I opened the first result, the “defaultItems.ts” file, I found a listing of many entries for many “items” with their default values. This file was ruled out immediately with the much more likely “Passage.ts” file chosen next. Comparing the two files showed some major differences. In Snowman 1.3, there are 223 lines of code in its “passage.js” file (Klimas, 2018). In the “Passage.ts” file for Adventures 1.0, there are only 18 lines of code (Kaczynski, 2017b). While the “Passage.ts” file has the same general property names, with “name” and “tags” appearing, for example, it seems to lack the same methods. As explained in chapter two, functions (methods) are the ways in which one structure communicates with another. The lack of functions in the “Passage.ts” file show an important change: its corresponding Passage object does not communicate with other objects within the code in the same way. A visual comparison using the software editing program Visual Studio Code shows the differences between the two files with large blank spaces on the right-hand side where the code lines do not match (Figure 22).

Figure 22. Color-coded Line-by-line Comparison between "passage.js" and "Passage.ts"

Following the same pattern applied to searching for property and method names from the "passage.js" file in the Adventures 1.0 source code, I performed a second search for keywords matching the "story.js" file from Snowman 1.3 (Table 7).

Table 6. Search Results for Story Keywords in Adventures Source Code

Name	Word Count	Word Presence
Story.ts	3.4285714285714300	0.2857142857142860
defaultItems.ts	2.0714285714285700	0.07142857142857140
Character.ts	0.7142857142857140	0.07142857142857140
Choice.ts	0.35714285714285700	0.07142857142857140
Passage.ts	0.14285714285714300	0.14285714285714300
Item.ts	0.07142857142857140	0.07142857142857140
Stat.ts	0.07142857142857140	0.07142857142857140

In the second usage, the file with the highest connection to the use of the property and method names from the "story.js" file in the Snowman 1.3 source code was the "Story.ts" file in Adventures 1.0. Opening this file showed some of the expected functionality using similar naming conventions from the Snowman 1.3 code such as the "show()" function, as mentioned in chapter two. However, beyond some obvious connections, the "Story.ts" file shows some major changes as well. In a similar way to the comparison between the "passage.js" and "Passage.ts" files, the "Story.ts" file is much shorter that its original counterpart "story.js" with the original at

506 lines of code and “Story.ts” at only 199 lines of code. A second visual comparison of the two files shows some of these changes (Figure 23).

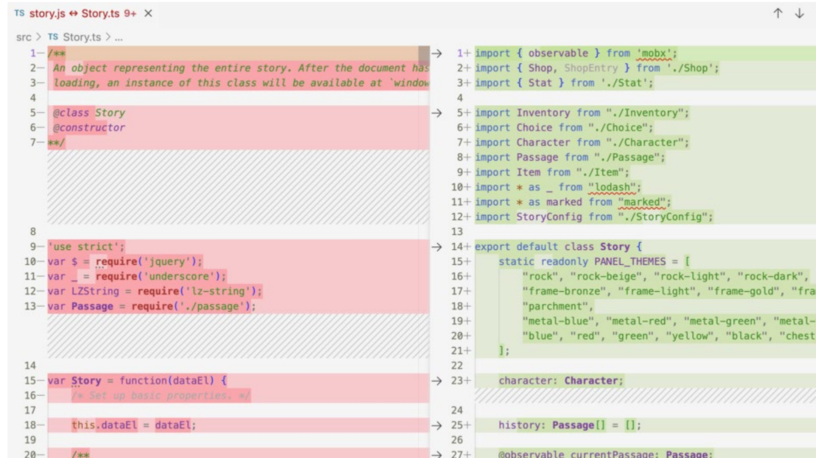

Figure 23. Color-coded Line-by-line Comparison between “story.js” and “Story.ts” files

While there are some similarly named properties from the “story.js” file in Snowman 1.3 appearing in the “Story.ts” file of the Adventures 1.0 source code, the introduction of new properties shows the shift in audience. Particularly, “lootableInventory” and “shop” names demonstrate expectations around the role-playing audience of the story format. These are connected to earlier lines in the “Story.ts” file where other files define core concepts of “Shop”, “Inventory”, “Character”, and “Item” referenced using special keywords in the programming language TypeScript. These show additional functionality contained in other files and combined, using the keyword “import” in TypeScript, with the original file when used together ("Handbook", 2012). Rather than all the functions related to the story format defined in one file, it is split between objects with their own properties and methods in other files.

The search of the names of the properties and functions of the “passage.js” and “story.js” files from Snowman 1.3 pointed to the obviously named counterparts of “Passage.ts” and “Story.ts” files in the Adventures 1.0 source code. Yet, the similar filenames hid the significant

changes in the Adventures 1.0 code from its “parent,” Snowman 1.3. As was outlined, there are two major shifts from the organization of the original project. First, the concept of a passage is changed through the move from Snowman 1.3, where each object has their own functions, to, in Adventure 1.0, as a collection of data affected by other functionality; it does have as many functions and thus does not communicate as often or in the same way. The second major change arrives from the use of additional concepts such as “Shop” and “Inventory,” pointing to the much stronger emphasis on functionality related to role-playing games not found in Snowman 1.3. Even with the change from JavaScript to TypeScript, it was still possible to find and, with some investigation of the “evidence,” in the words of Tornhill (2015), how the same concepts and named functionality appears in both projects in slightly different forms. While Adventures 1.0 showed a seemingly major departure from the Snowman 1.3 source code, the next story format, Trialogue 0.0.8, has much more subtle changes for its own audience.

Trialogue 0.0.8

The documentation of Trialogue explains it as a story format for changing “a branching narrative into an interactive chat story” (van Kemenade, 2022a). Unlike many story formats where a new passage completely replaces the current one, Trialogue appends the content. This echoes the layout of one of the early story formats for Twine 1, Jonah (see chapter one). In Trialogue, this creates a chat-like appearance as each new passage loads at the bottom and adds to the top. By allowing choices between different options at the bottom of the screen, the visual presentation of routes through a story replicates a “chat” layout between different parties (Figure 24).

Figure 24. Screenshot of Trialogue Visual Layout

The story format Trialogue began as work funded by the Slovak National Gallery as part of the Filla Fulla Chat project in connection to the centennial anniversary of the founding of Czechoslovakia (*Chat with Emil Filla & Ľudovít Fulla*, 2018). It was then improved through funding connected to the Tolerant Futures project as a collaboration with the University of Edinburgh and Durham University (van Kemenade, 2022c). For much of Trialogue's history and development, there were fewer versions with the first official one occurring with 0.0.8 on February 6, 2022 (van Kemenade, 2022b). Like Snowman, Trialogue is also written in JavaScript.

Like the search of the Adventures 1.0 source code, I used the same keywords from the Snowman 1.3 code from its "passage.js" (Table 1 and Table 2) and "story.js" files (Table 3 and Table 4) in the same workflow in Orange (Demšar et al., 2013). Using the Passage-related keywords quickly showed a familiar naming convention from Snowman 1.3 manifesting in Trialogue 0.0.8 (Table 8).

Table 7. Search results using "passage.js" keywords in Trialogue 0.0.8 source code

File Name	Word Count	Word Presence
story.js	11.714285714285700	0.7142857142857140
passage.js	10.285714285714300	1.0
index.html	3.142857142857140	0.2857142857142860
trialogue.css	0.2857142857142860	0.14285714285714300
index.js	0.0	0.0

Investigating the presence of a "passage.js" file in the Trialogue 0.0.8 source code and comparing its contents in the same file from Snowman 1.3 showed some spacing and the introduction of many new lines. Beyond these editing and comment changes, one notable addition was found in the introduction of a new property, "links," in the "passage.js" file of Trialogue 0.0.8 not present in the original file. Using a line-by-line comparison, the use of green lines in a file comparison visualization shows the new content in the Trialogue version and its placement in reference Snowman one (Figure 25).

Figure 25. Visualization of Changes Between Snowman and Trialogue "passage.js" Files

Investigating the "story.js" file of Trialogue 0.0.8 also showed some significant differences between the projects. The "story.js" file in the Trialogue 0.0.8 source code is a total of 848 lines of code whereas the original is 506 lines of code. The additional 342 lines of code seem to come from, in part, a major departure from the recording of changes to the story and how they are tracked over time. These additional lines are most obvious from the "start()" function shared by both projects with the use of a red color showing the changes to spacing and

the use of green additional lines in a line-by-line comparison of each file. In multiple instances, many new lines of code exist between the original formatting in the same file from Snowman 1.3 (Figure 26).

Figure 26. Visualization of Changes Between Snowman and Trialogue "story.js" Files

Unlike the more dramatic departure found from the movement from JavaScript to TypeScript found in comparing Snowman 1.3 to Adventures 1.0, the changes in Trialogue 0.0.8 are more subtle. While the "passage.js" files are named the same and seem very similar on the surface, the change of one property in the Trialogue 0.0.8 version points to a shift in the processing of authorial content. Rather than create internal links, connections between passages a player can click on to progress a story, as part of the processing of its content, the visual layout itself is changed, connecting back to how the story format handles is visualization presentation as different from Snowman 1.3. This appears first in the addition of a new property in the "passage.js" file and then in the processing of this property in the "story.js" file. Searching for the new property "links" shows it as part of functionality used to make the content appear after a short delay to mimic a person responding after a moment of consideration to compose a reply.

Each time a reader interacts with the story, there is a short delay before the next part is shown to them as if another person was chatting with them in real-time.

Children of Snowman

While the text mining tool Orange is useful for finding files matching specific keywords, the study of both the Adventures 1.0 and Trialogue 0.0.8 source code in this chapter found files matching the same names as the original Snowman 1.3. Such a legacy points to two important aspects of research shown in this chapter. First, using text mining tools can be helpful in isolating files for potential reviewing, but it must always be paired with “human expertise,” in the words of Tornhill (2015). In multiple cases, the files Orange predicted as most useful to search were not ones matching the same structures even if the keywords appeared there most often, as was the case with the “defaultItems.ts” file from the Adventures 1.0 source code. For these, a close examination of the code showed similar structures in other files matching those from the original project. Second, the use of static analysis and comparing the files showed changes between them that would most likely not seem as obvious in seeing the code running. It was only through a line-by-line comparison was it possible to see, for example, the changes between the “passage.js” in Snowman 1.3 and “passage.js” in Trialogue 0.0.8 files and then carry this change into additional research into other files to determine how the new change affected the visual representation of hyperlinks internally in Trialogue 0.0.8. Through the combined research methods, this helped show the differences between the story formats from Snowman 1.3 and how these changes reflect adjustments in their functionality toward different audiences and expectations.

Despite the difference in programming languages between Adventures 1.0 and Snowman 1.3, it was still possible to study the code structures in each. Notably, the number of lines of code between the similarly named files helps provide "evidence," in the words of Tornhill (2015), of how its internal objects acted in different ways. The greater number of methods found in the Snowman 1.3 code point to how its Passage object communicates to other code. The lack of them in the Adventures 1.0 code shows the opposite. Its own Passage object does not communicate as much in the same way. At the same time, the increase in lines of code and greater number of core concepts, represented by the use of the "import" keyword in TypeScript in its "Story.ts" file ("Handbook″, 2012), show the ways in which Adventures 1.0 contains much more code across other files named with functionality matching its role-playing audience and description for authors considering using it (Kaczynski, n.d.). This points to more functionality associated with its core concept of Story than connected to its Passage object. With more methods found in its "Story.ts" file than "Passage.ts" file, this also demonstrates greater communication between other objects and processes of data in one concept than the other such as the "rendering" found in Snowman 1.3's Passage object moved into the Story object of Adventures 1.0's source code. The lack of the same methods with the prefix "render", as found in the Snowman 1.3, illustrates how this functionality had been moved to be a part of a different object internally in Adventures 1.0. In Snowman 1.3, two of its three methods perform the action of "rendering" in its Passage object. The lack of these point to how the same object is acted upon rather than perform the actions itself.

While the core files of Trialogue 0.0.8 matched those found in Snowman 1.3, its changes also showed different expectations. The introduction of a new property, "links", revealed how story content appears to readers, reflecting a display layout echoing one of the earliest story

formats for Twine, Jonah (see chapter one). At the same time, the greater number of code lines also points to, in the same way it did in the Adventures 1.0 code, an increased focus on the Story rather than Passage object with more methods added in Trialogue 0.0.8 than exist in the same object in the Snowman 1.3 source code. As was explored in this section, this additional functionality is connected to the change in how the story format displays story content, introducing a delay in showing the result of a reader clicking on a hyperlink in a story and how the content is displayed to them.

Interestingly, neither Adventures 1.0 nor Trialogue 0.0.8 seem to contain two properties found in Snowman 1.3 in the form of metadata of the story being displayed to the reader. While the method names are similar across all three projects, the properties “creator” and “creatorVersion”, data containing what program, usually but not always Twine, that created the file and its own version, do not appear in Adventures 1.0 and Trialogue 0.0.8. Because these properties are not used in Snowman 1.3 beyond their recording from the HTML data, this might explain why they are missing while other structures, like methods of the same name, are retained. Such changes also speak to the ways in which some parts of the original project are removed or improved upon across new projects based on the same concepts, refining how the same general structures are understood and re-created by other programmers.

Conclusion

While centered on trying to improve a single project, the metaphor of code as a “crime scene” is helpful for studying source code (Tornhill, 2015). Through applying a static analysis approach, the text of code can be studied through examining how it connects to other projects by comparing their naming and code structures without needing to run it (Rival & Yi, 2020).

Borrowing from literary studies, applying distant reading or macroanalysis can help in looking for specific patterns in the code as a text (Jockers, 2013; Moretti, 2007). However, text mining is not enough to fully understand the code. As with the results shown from Orange, what it highlights from the source code of a project is not always the file needed or matching functionality from the original. This can then be paired with close reading and a comparative method to understand both where larger trends happen, as produced from tools like Orange, and how those instances can be studied within texts (Harvey & Pagel, 1991; Jockers, 2013). By alternating between using text mining tools and then closely examining those finding through comparing them across versions, the history of a project can be more easily mapped from the original onto its "children." As Tornhill (2015) warns, the visible code of any one file is not enough to understand a project. The present code has a connection to a deeper past (see chapter two). As shown through this chapter as an application of digging into text of other projects, while both Adventures 1.0.0 and Trialogue 0.0.8 have files sharing the same general names of those found in Snowman 1.3, an analysis of the structures within the source code show sometimes subtle interconnections such as the introduction of an additional property "links" in the Trialogue 0.0.8 and its role in displaying content in a chat-like presentation. The same can also be written about the removal of the functions from the "Passages.ts" file in the Adventures 1.0.0 code and what this demonstrates about the ways in which objects communicate with each other within the source code based on the absence of functionality seen in the original project. While text mining of both projects revealed similar files that might have been found by searching through every directory of the project without the tool's help, the analysis of each went deeper into how the difference manifested in the functionality based in the re-organization and assumptions of each story format. As Tornhill (2015) explained in moving beyond relying on only tools to study

code, “when it comes to software design, there’s no tool that replaces human expertise” (p. 8). Understanding the hidden relationships of projects starts with tools to help find areas to potentially study and then bringing a human understanding of code to explore how the concepts found within the code structures relate to each other beyond the obvious searching for keywords and similar names tools can provide.

This chapter opened on the history of the Mosaic web browser and its connection to the more famous Netscape’s Navigator and Microsoft’s Internet Explorer. Unlike the research presented in this chapter, such a comparison between the early versions of their source code and the original project are not possible. While the code to Mosaic is public, and so too is the version of Navigator that eventually became Firefox nearly a decade later, the code for Internet Explorer is not. Yet, what secrets might it show using the same research methods used in this chapter? What, too, might be found by exploring other projects using the companion methods of collecting keywords from one project and then using text mining to search through its source code before carefully reviewing the resulting files found? As explored in the previous chapter, projects are often based on the social values encoded in networks created from the dependency of one package on another. What might be found in the deeper history of many of the commercial projects and unreleased source code? Might they contain comments like the one highlighted in chapter one? Researchers may never truly know the impact of the concepts and code structures of Mosaic and if they found their way into influencing how webpages were viewed nearly 30 years later when the long life of Internet Explorer finally ended. The same can also be written of many projects yet to be studied as applications of all kinds are built on the structures, patterns, and keywords of the past with some, like Mosaic, serving as a textual origin

for projects across decades of usages. How might the reflections of the present by found in future applications?

CHAPTER 5: END OF BEGINNING

The history of the story format Snowman as presented across Ford (2016), Salter and Moulthrop (2021), and the collected tutorials of Baccaris (Baccaris, 2020, 2021) is of a seemingly minor work compared to its sister story formats, hardly worth mentioning for those wanting to use the authoring tool Twine. Yet, this dissertation paints a very different picture. While previous research has shown its popularity is low (Cox, 2020a), as the previous chapter outlined, Snowman was the foundation of some 20% of all known story formats in 2018 (DeMarco, 2018). It also incorporates influences across some 600 other packages on which it depends (chapter three) and has served as a direct influence of other story formats like Adventures and Trialogue (chapter four). How can the perception of Snowman from one set of works on one side and the evidence of this dissertation on the other be reconciled? Snowman must be considered through its social relationships. In the book *Critical Code Studies*, Marino (2020) establishes the use the term "critical" as an important aspect of studying code. As explained by Marino (2020), the "*critical* in critical code studies encourages also exploring the social context through the entry point of code" (p. 28; original emphasis). Through the repeating defining of code as a "social text," Marino (2020) creates an understanding of code in which can be studied through its textual relationships such as its community, usage between human and computer systems, and its connection to other projects (p. 5). Across three research chapters, this study has presented Snowman within its relationships to other software. In chapter two, this was how Snowman includes and uses the functions of other software to create the structure of a "stack." In chapter three, these were the packages on which Snowman depends and the power these could potentially display to be disruptive to the larger networks to which they belong through the implicit trust of one package on another. Finally, in the previous chapter, chapter

four, Snowman was shown to have heavily influenced two specific projects, Adventures and Trialogue, with potentially many others not included in the chapter. Put in this "social context," as defined by Marino (2020), the importance of Snowman is not as an end point, but as a beginning for other works extending forward in unexpected ways. As the three examples of the comment from the 6th edition of the Unix operating system (chapter one), protest by Marak Squires (chapter three), and role of the graphical web browser Mosaic (chapter four) demonstrate, the "strata," as proposed by Foucault (1969) is hard to entangle. What exactly is the beginning of these histories, and can the narrative of one truly be separated from another? What is the history of the 6th edition of the Unix operating system without the spreading of its infamous comment? Would the act of Squires through the project *colors* have had the same effect if other projects did not have such implicit trust in one another? What would accessing webpages look like without the "root" software of Mosaic to help shape the interactive experiences of potentially tens-of-millions of users over nearly three decades? Finally, of course, what is Twine without the influence of Snowman on authors and programmers wanting to use it as a foundation for their own work? How might, in a more personal connection, this study become very different if I had not taken over maintenance of Snowman? Would the work on the Twine Specification documents had happened if I did not, in the words of D'Ignazio and Klein (2020) have the "direct experience" (p. 21) needed to work first on the documents and then have the "human expertise" to explain the structures of the Snowman code (Tornhill, 2015; p. 8)? The social context of all these projects is understood from the entry point of their code and then how the cultural and social relationships influence each other.

In this final chapter, the results of research into the social relationships through code of Snowman are reviewed. In the next section, Findings and Contributions, the research questions

are included along with the theoretical framing, research results, and contributions of each chapter. Next, in the next section, Limitations, the boundaries of the research and presented results are examined, explaining the reasonings for why they were placed and observed in respect to the research performed. Finally, in the last section, Future Research, I propose new research questions and paths following a focus on what more can be studied on Twine and its story formats and for the field critical code studies more generally through comparing code projects. Research is never done, and the work on better understanding code as a "social text" must continue beyond this study and other, similar works.

Findings and Contributions

The research performed and reported across three chapters in this dissertation were driven by questions seeking to better understand the connection between the Twine story format Snowman and the code on which it is dependent and interacts. As explained in chapter one, Snowman was chosen as a site of study for several reasons. First, there is a notable lack of academic research into Snowman, with it missing from multiple books on Twine, its history, and how to use the authoring tool to create stories (Ford, 2016; Salter & Moulthrop, 2021). Second, Snowman has served as the basis for multiple story formats and was the foundation for some 20% of all known story formats in 2018 (DeMarco, 2018). Third, and as discussed in chapter one, I have close, “direct experience” (D’Ignazio & Klein, 2020) with the code of Snowman as its current maintainer and this experience led to my co-authoring documents to help others understand story formats and associated formats (Cox et al., 2019) beyond my previous work on the Twine Cookbook (Cox, 2021c). Based on these three reasons, this study presents research into Snowman from a textual perspective, examining the ways in which its code “calls” to the past, is built on the trust of the present, and how its concepts are reflected in projects on which it

is based. This section moves through each question driving this dissertation with a short summary of the theory, findings, and how the research and methods contribute to the approaches used in each chapter.

Research Question 1: How can the metaphor of "strata" (Foucault, 1969) serve as a lens to understand the archaeology (Parikka, 2015) of function calls (Soloman, 2013) within the story format Snowman?

In the book *The Archaeology of Knowledge*, Foucault (1969) establishes the metaphor of "strata" as linked to studying of history deeper than the obvious connections. Those researching history should move into, as an example Foucault (1969) gives, the ways in "wars" can be linked to the "history of sea routes" (p. 3). Foucault (1969) raises an important aspect of all historical narratives as being entangled in a mesh of the past and present without a clear linear progression between one and the next. Building from this work, Parikka (2015) shifts the focus of "strata" into the conceptual space of geology, pointing to how all media can be investigated through applying a form of archaeology where the digital present coexists with a material past at the same time. Like Foucault (1969), Parikka (2015) calls for a study of the points of "double articulation" where both past and present are enmeshed with each other (p. 36). Parallel to Parikka (2015), Soloman (2013) has examined "function stacks," the ways in which present functionality "calls" to past code through its interfaces, as being linked to how each "layer" in the larger stack exists in a "paradoxical arrangement" with each other as this acts to both "open up possibilities" while also "always already constrained and preconditioned" by other systems, echoing the same findings on media from Bolter and Grusin (2003) and their use of the concept of "remediation." To this understanding of interfaces and the structure of a stack, Galloway (2006) and Bratton (2015) add studies of the ways in which power and control are concentrated

in how systems expose functionality and values through the interfaces they provide to each other. Across a stack, those on the "bottom" serve as having the greatest influence on, as explained by Soloman (2013).

This chapter investigates three lines of code found in the Snowman 1.4 source code. Specifically, these are Lines 205 – 207 in the "src/Story.js" file, serving as the point where the code hands over control from its running to a possible external actor such as a human reading the story. These lines of code also act as a "double articulation," as explained by Parikka (2015), by being a point where not only can external forces intercede in the working of the story format, but also where multiple software libraries are used at the same time, showing not only the ways in power and control are centered on interfaces Galloway (2006), but also the ways in the arrangement of the function stack by the three lines of code also points to how it opens and closes the expressions of the next layer in the arrangement (Soloman, 2013). Beginning with the software library jQuery, the chapter digs into the history of how it came about and its role within the Snowman before continuing with another library, Underscore. Between both jQuery and Underscore, values are processed, and data returned "up" the stack, until it encounters functionality based on the programming language JavaScript. Within Snowman 1.4, when a reader clicks on a link, there is an entangling of the present code of the story format with code written in a deeper past across jQuery, Underscore, and the use of concepts within JavaScript. Hidden in the seeming simple interaction is an exchange of data and power across a complex function stack in the lens of different "strata" where each interconnect with each other with the summation of their functionality as equally important as any one part in understand how both they are working separately and how the structure in which they exist adds to this understanding of how data is processed by the story format Snowman.

While examining only three lines of code within a single file in the story format Snowman, this chapter also points to the ways in which all code projects are built on histories both obvious and more hidden within its own contexts. To recall the metaphor of "strata" embraced by Foucault (1969) and Parikka (2015), the past and present of history exists at the same time in many points and a deeper understanding of the influences of code especially can be gleamed by digging into how these points help to understand how past code structures can serve to shape the present within function stacks (Soloman, 2013). While focused on Snowman and its connection to other JavaScript libraries, this chapter also establishes a method by which other code could be studied by moving beyond the obvious connection to the past and investigating how other code libraries and packages are used. Through the interfaces of each project, a degree of control (Galloway, 2006) is created in what the code agrees to accept and what it will ignore (Brown, 2015). The programming interface of each project defines not only its own influence within the project in which it is defined but also an understanding of other structures in which it interacts and exchanges data. Through researching these points of interaction, greater insights can be found in how systems influence and affect each other.

Research Question 2: What can be learned from studying the "network" (Latour, 2007) of the code and testing dependencies of the story format Snowman through the ways in which "switches" play a role in affecting its arrangement and relationships (Castells, 2010)?

In the book *Software Takes Command*, Manovich (2013) defines "cultural software" as all software connected to "actions we normally associate with 'culture'" (p. 20). Building from work like Manovich's (2013), Marino (2020) defines code as a "social text" with different audiences, both computer and human (p. 4). Seeking to recontextualize the term "network," Latour (2007) stresses the defining aspects of a network is the "work" distributed across "net"

relationships. These exist as both visible and more seemingly invisible, stretching across multiple entries and their own relationships to others. To the concept of networks, Castells (2010) brings the metaphor of a "switch" as a center point acting as "privileged instruments of power" (p. 510). For Castells (2010), all networks inevitably give rise to unequal relationships where one or more of the entries within them act as a "switch" for others. At the same time, these networks affect more than the digital locations in which they might arise and have the potential of "shaping social structure" (p. 510). The changes affecting the digital can also pressure more material relationships. From Manovich (2013) and Marino (2020) come an understanding of code as existing within relationships affecting other systems, both computer and human. From Latour (2007), the concept of a "network" as composed of distributed relationships across both obvious and more hidden connections. This chapter bridges from the conceptual network into considering how smaller parts within it how greater influence than others and especially between the digital and social. Castells (2010) serves as an anchoring of Latour's (2007) articulation with networks containing "switches" acting to pressure those around and connected to it in different ways.

This chapter examines the code and testing dependencies of the Twine story format Snowman 1.4. Among these, including every dependency of all other packages, are hundreds of interconnections across over 600 packages. In visualizing these relationships, the concept of a "switch" from Castells (2010) becomes apparent across three configurations. First, there are many highly connected projects on which many others are dependent. These, like the projects of *inherits* and *defined* examined in the chapter, often represent only a handful of people with smaller number of changes over longer periods of time (Halliday & Harband 2012; Schlueter, 2011). Second, there are many projects like *browserify* and *cssnano* that have seen many changes but are themselves dependent on many other projects, creating a center who projects dependent

on but they themselves are dependent on others. Third, there is a final configuration identified in the Snowman 1.4 dependency network: those along the edges. There are many projects who appear along the edge of the graph without which other projects along a dependency connection would fail. These exist in not a direct dependency to any projects, but often multiple levels deep with many existing as dependencies of hundreds of others throughout a series of trust relationships from one to the next across the network. These equally contribute to the "work" of the net number of relationships with the Snowman 1.4 dependency network, as explained by Latour (2007). By showcasing examples, this chapter points to how the social influence on Snowman exists across a network of forces stretching across hundreds of other projects spanning the work of many people across both the recent past and older libraries of code.

Not only does the code found in any one project often depend on things solved by others, but these trust and labor relationships affect each other. This points to how the "network" of dependencies across code is not only a security issue, in the example of the project *colors* highlighted, but this chapter also highlights how the present code of many projects reaches into a distributed past across hundreds of people and decades of work. This "network" of code projects is not only a history of labor, but also, in the words of Castells (2010), have the potential of "shaping social structure" (p. 510). There is an obvious reference between the changes in the highly connected projects and those dependent on them, but this also extends to the smaller projects on which many depend multiple levels deep. A change in how, for example, the project *inherits* works could ripple across tens-of-thousands of projects as the update to the project *colors* did within the incident noted in the chapter. At the same time, the example of the project *colors* also points to how the trust embedded in the network often goes unquestioned across code structures and their own programming interfaces. The "work" of the hundreds of smaller projects

is not equal with many serving as a "switch" without, perhaps, being aware of how future work would depend on them.

Research Question 3: How can applying a "macroanalysis" approach (Jockers, 2013) of using both "distant" (Moretti, 2013) and "close", comparative readings (Harvey & Pagel, 1991) of both "child" and "grandchild" source code be used to show a "legacy" of the story format Snowman in those projects based on it?

Code projects are often based on previous work. This can take the form of building on ideas, concepts, or simply wanting to experiment with different coding structures for the same functionality. Yet, when it comes to tracing the "legacy" of coding structures between an original and those projects based on it, the research involved can often be complicated. Frequently, projects might be written in a different programming language or incorporate changes for additional functionality only present in the newer projects. Tornhill (2015) provides a useful way to start an investigation by understanding code as a "crime scene." By treating the source code of a project as text outside of a computer running it, the code can be researched using what is named "static analysis," a concept common in security and history connected to software forensics (Rival & Yi, 2020; Sallis et al., 1996). At the same time, borrowing from literary studies brings the method of "distant reading," the analysis of large or many textual sources where a single human would struggle trying to read and process the amount of relationships or easily find patterns (Moretti, 2013). Static analysis, based on software studies, and distant reading, pulling from Literary Studies, both provide tools for understanding text and finding patterns. Yet, as Jockers (2013) explains, distant reading by itself can often miss smaller details by only looking for large trends. There is a need, using a concept created by Jockers (2013), to perform a "macroanalysis" where the more traditional use of close reading, a detailed analysis of

a single document or lines within a larger work, with the newer distant reading, pairing a macro search with a "micro" analysis of patterns found using digital humanities methods like text mining and pattern analysis. By using keywords gleamed from an original work, other projects can be searched for names or structures and then these searches closely examined for how they meet certain criteria or if the same pattern from the original appears in the "child" work.

This chapter follows the methods set out by Jockers (2013) as changed using static analysis and code studies through how Tornhill (2015) understands code as a "crime scene." This chapter begins with collecting a set of keywords based on the JavaScript properties and functions within Snowman 1.3. This version of Snowman was picked because it serves as a "branching" point for many other story formats based on it including the project Adventures 1.0 (2017) explored in the chapter. Based on the keywords collected for the two major concepts in Snowman, Story and Passage, the text-mining tool Orange is used to perform a search across all the files in a project for those matching the usage of the keywords per concept (Demšar et al., 2013). Beginning with Adventures 1.0, the project is compared to the keywords from Snowman 1.3 and instances of matches are explored for how they compare to the same places in the original code. This pattern is then repeated for Trialogue 0.0.8. Overall, both Adventures 1.0 and Trialogue 0.0.8 were found to closely match the same file structures and concepts of Snowman 1.3 with a greater change in formatting in Adventures 1.0 because of the use of the programming language TypeScript. In each case, the comparisons between the same-named files also revealed changing of functionality to match their audiences with Adventures 1.0 containing many genre-related data structures and Trialogue 0.0.8 with functionality to support its visual layout changes. Both projects also seemed to mirror the naming conventions of Snowman 1.3 in their

understanding of a story format containing both Story and Passage core concepts with Adventures 1.0 adding more such as Inventory and Shop for its own purposes.

While Trialogue is still receiving updates, Adventures 1.0 has been archived by its original author. The same is also true of Snowman 1.3, which has become two different branches of 1.4 and 2.0 based on the work of a new maintainer (Cox, 2017a). By basing the research in performing archival work, this chapter also established the use of macroanalysis as a useful method for understanding code through the combination of searching for keywords and structures across a project based on an original analysis of a project and then looking for these terms in branched projects. Through using macro searches and micro analysis, this chapter created methods for understanding potentially very large projects containing hundreds if not thousands of files through, first, collecting keywords and then performing a distant reading using text mining tools like Orange before then using matches to perform a close reading of how code structures and patterns are different in the newer project than the original. Such methods, outside of their application in this chapter, hold great promise for studying other projects such as how Mozilla's open-source Firefox is based on the code of the commercial Netscape Navigator created nearly a decade before. It also has many applications across many other situations where there is code access to both the original and "branched" versions for comparison and analysis.

Limitations

Each chapter included in this dissertation had self-imposed limitations to keep the scale of each chapter within a manageable size. Because each chapter could easily grow into a large-scale research project of its own, these limitations serve not only to keep the chapter within a reasonable scale, but also to maintain the entire project as being in conversation with itself across

the document, serving not as a single method for research for understanding the Twine 2 story format Snowman, but also how it exists within relationships to other libraries and projects based on its structures and concepts. This section explores three limitations, one per chapter, and why they were imposed to limit the size of each chapter.

"True" depth of function stack and lack of access to source code for some applications and operating systems.

Chapter two explores the function stack of three lines of code found within Snowman 1.4. Within the chapter, the software libraires of jQuery and Underscore are discussed. Yet, the full function stack of the operation extends much more than is discussed within the chapter, moving from the internal operations of JavaScript within a web browser into how the application of the web browser works and then into the internal mechanics of the operating system in which it is running. Depending on the web browser, it might be possible to explore the source code with access available for Chromium, the base libraries used with the Chrome web browser (*Chromium*, 2023) or for the Firefox web browser (*Mozilla Search - Firefox*, 2023). Beyond having access to the web browser source code, a full investigation would also require looking into the operating system itself. The Windows operating system is commercial software licensed through Microsoft and does not provide access to its source code. This is also true of macOS, which is licensed through Apple. It could, potentially, be possible to examine the entire function stack beginning from Line 205 through to the lowest level available by running the code using an open source operating system such as those available using the Linux kernel such as Ubuntu (Ubuntu, 2023) or its sister project Debian (Debian, 2023). However, such an exploration would entail an extended analysis and understanding of not only how JavaScript runs within a web browser, but how the browser itself accessed hardware resources within the operating system and

then down to the hardware level. Such a detailed explanation could easily take up hundreds of pages. An accepted limitation of the chapter is an analysis of the function stack as it appears within a single programming language, JavaScript, and the libraries used within direct connection to the functionality discussed in the chapter.

Dependency network size considerations.

Chapter three examined the code dependencies of Snowman 1.4. Yet, among the hundreds of packages mentioned, only a few were selected for highlighting. This was a specific limitation placed on the chapter to not extend it significantly by examining each package and their own histories and inclusion as dependencies in other packages. Specific examples were chosen to showcase configurations rather than an exploration of the entire network and all its code, potentially tens-of-thousands if not hundreds-of-thousands of lines of code in total. An extended study could, potentially, examine the complete ways in which the Snowman 1.4 is dependent on other code and the ways in which even the smallest project played a role, if any, in how the code was processed. Because of the inclusion of testing and development dependencies for every package, it might be the case that the extended network, while seemingly expansive, is not used by Snowman 1.4 directly, with only a single function or file from the hundreds of packages being involved in the processing of Snowman's source code during building and testing processes. However, an investigation of this size would be both very time-consuming and ultimately reflect the same results shown in the visualization of the network within the chapter. The limitation of only six examples was placed on the chapter to showcase patterns within the network rather than everything involved in every package, a potentially very expansive project and work for a single dissertation chapter among a larger work reflecting other aspects of research into similar subjects.

Limitation of only two projects examined using text mining and close reading methods.

Chapter four only examines three code projects. The first is the Snowman 1.3 code to establish a baseline of the keywords based on the names of properties and methods within its source code. Next, Adventures 1.0 and Trialogue 0.0.8 are compared against the Snowman 1.3. Yet, within the methods expressed in the chapter using distant reading tool searches, there is a possibility to quickly search through code projects to identify files of interest for later close reading and comparative methods. Potentially, this chapter could have included many other projects for investigation beyond the two chosen. This would have presented the ability to examine many other projects to see if they too used the same code structures and naming pattern mentioned within the chapter found in Snowman 1.3 and replicated in Adventures 1.0 and Trialogue 0.0.8. However, because the time required to review these additional projects on a file-by-file, and sometimes a line-by-line comparison, a limitation was placed on the chapter to curtail the possibility of the chapter outgrowing the entire document through reviewing many other projects and even a single or couple of important changes per projects. These methods hold great potential for future research, but within the chapter were only applied across two projects in connection to an original to keep the scale of the project within a single dissertation chapter rather than a book-sized project by itself.

Future Research

As discussed in chapter one, the Twine story format Snowman has not seen much coverage or examples in either Ford's (2016) or Salter and Moulthrop (2021) books. Snowman is also often neglected when it comes to tutorials and guides (Baccaris, 2020, 2021). Yet, it has inspired multiple "children" in the form of multiple story formats based on the original work of

Chris Kilmas through Snowman 1.3 and additional changes by Daniel Cox into Snowman 1.4 and its 2.0 branches. Based on this lack of academic coverage of Snowman within those already discussing Twine, there is a need to investigate not only how people use the story format to create stories in Twine, but the ways in which Snowman has served as an inspiration and foundation for other projects. This study is the start of some of this work, examining not only how Snowman relates to other libraries it includes and uses within its processing, but also how it is positioned within a network of forces and how these forces influence what it can offer authors when using it to create new stories. This section details three possible paths forward for research like the methods examined in this document. First, future research could examine the ways in which the HTML storage of Twine affects the construction and interpretation of story formats when handling that data. Second, future research could focus on the authoring experience in Twine in connection to how decisions are made on story formats and how an author understands themselves in reference to what expressions are available to them. Third, this section ends with pointing toward how the methods used in chapter four could provide greater insights into the historical and textual connections between other software projects.

How do the HTML storage elements of Twine affect the conceptualization of story format functionality?

As covered within chapter three, Snowman 1.3 has two core concepts: Story and Passage. They appear not only in the Snowman 1.3 code but are also mirrored in those story formats based on it examined within this study: Adventures and Trialogue. This suggests they might be common in other story formats based on concepts with the same name appearing within the HTML storage elements created by Twine. Future research could investigate other story formats outside of the Snowman family such as Harlowe and SugarCube to see how these same concepts

appear and in what ways the data of the Twine 2 HTML elements appear and are used as properties of objects within JavaScript within the other story formats. Do they use the same code structures in the same way? If so, is this an influence of the HTML elements? Does every story format need the same concepts because of their appearance within the storage layer of every story format? Put a different way: does the presence of the HTML elements predict object creation in the programming languages processing them? Is there a way to understand Twine outside of the Story and Passage object metaphors within coding structures? And, if so, what might different metaphors bring to the same data for understanding its structures and presenting the data to people experiencing the content? Like with the work done on the story format TwineSpace, itself also based on Snowman, and the use of the metaphor of a "scene" (Cox et al., 2022), is there room within the community of Twine to re-imagine the same data using different structures and relationships based on the same or other storage methods or additional HTML elements and attributes? Shifted outside of HTML, could new expressions become available based on a different storage language and thus processing layer of new story formats? At the same time, Twine is not a traditional software platform and story formats can be used outside of the authoring tool to create compatible and non-compatible works. There may come a point where additional data or storage formats could be used with story formats outside of Twine to make Twine-like experiences for web browsers without the need of the Twine 2 HTML elements themselves (Edwards & Cox, 2021). Future research is needed to push at the edges of how important the authoring tool Twine is for what people assume is the HTML output created by it produced by other tools using the same storage approaches and structures.

How do authors understand themselves in reference to what is, and is not, available in a story format?

While there has been some work done on trying to quantify the different story formats used by authors across time (Cox, 2020a), much more work is needed to study how people understand story formats and the authoring process more generally (Daiute et al., 2021). Why does an author choose one story format over another? Do they make this choice based on certain factors such as available functionality or ease in performing certain tasks? Do any authors feel "trapped" within a story format based on issues like not wanting to translate between one story format and another for a larger story? Are there any authors who have moved between story formats based on certain issues they encountered when using a story format? Under what contexts might an author choose one story format over another? There are many questions surrounding the authoring experience within Twine which have not been studied or at least studied with a clear focus on Twine as an authoring tool exclusively. This lack of research opens the door to many approaches to research including a focus on the editing interface provided by a story format and how much this contributes to an understanding of its available functionality as well as how much the documentation of each story format helps (or hinders) an understanding of what is available from the story format when creating stories. There are also questions about how much "programming" an author is comfortable with in a story format, as Twine itself is often presented as a tool without a need for "coding" with such phrasing also appearing on its own homepage. Is the inclusion of needing to use JavaScript, for example, something which prevents people from using Snowman over, say, Harlowe, which presents itself as a programming language (Arnott, 2022)? There is also room for future research to examine the perception of coding from certain people as being "against" the perceived views of something outside of their knowledge or not important to their learning despite Vee (2017) explicitly making the

connection between greater literacy and ability to make better informed decisions in digital contexts.

What can similar structures and naming patterns show us about the textual relationships between software projects?

Chapter four establishes the use of pairing distant and closing reading methods for researching code projects where keywords are gathered and used to search for similar code structures when comparing projects by treating the code to a static analysis using text mining tools (Rival & Yi, 2020; Tornhill, 2015). While the chapter uses these methods to compare Twine story formats and their changes, the same methods could also be applied to other subjects. Future research using the same methods could compare, for example, the open-source code of the graphical web browser Mosaic with its later "grandchild" of Mozilla Firefox, searching for and comparing changes from 1993 to 2002 through comparing the assumptions of provided functionality to users of web browsers. At the same time, the methods could also be used to explore across similar projects implementing the same functionality but released under different licenses. For example, there is a historical archive of the Microsoft Disk Operating System (MS-DOS), a popular operating system pre-dating the later graphical Windows operating system (*MS-DOS v1.25 and v2.0 Source Code*, 2018). This could be studied against the FreeDOS open-source code implementing the same concepts but under different authors (*FDOS Kernel*, 2018). There are many comparisons to be made following this same model through examining historical code to newer versions of the same general implementations or standards, researching if naming conventions and code structures are similar. For any instance of using these methods, the most important aspect for future research would be access to the text files composing each project for use with text mining tools such as Orange for easier searching (Demšar et al., 2013). If the code

can be read and processed by text mining tools, patterns can be found across dozens or potentially hundreds of files for similar structures and keywords before being reviewed more closely for how each usage is similar or different than each other.

End of Beginning

As of this writing, Snowman has seen 491 changes from 2014 to 2022 with nearly all of those changes coming from either its original maintainer, Chris Klimas, from 2014 to 2019, or me, Daniel Cox, from 2019 to 2022 (Cox, 2017a). As I write this in June 2023, there have been no changes to Snowman in 2023 with a minor version update based on the newer versions of its own code dependencies. Despite this, the future of Snowman is brighter than it ever has been because of the many projects based on its concepts and structures. As a single project, Snowman is unlikely to ever gain ground on the usage statistics of its sister story formats I recorded in 2020 (Cox, 2020a). Yet, the single fact some 20% or more of all known story formats in 2018 are based on Snowman show the potential for it, like the graphical web browser Mosaic before it, to be the "root" of a vast tree of other projects expanding outward in unexpected ways (DeMarco, 2018). Like the projects on which it is based, Snowman has inherited some cultural values and has, in ways yet to be studied, passed these on to other people (Manovich, 2013). Will future papers and books on Twine include more of Snowman's history? Or will it slowly fade as a steppingstone along a longer path toward much more famous works on which it influenced along the way? It is my hope this study, and any work to be published from future research arising from it, continues to investigate and present the history of Snowman and projects like it. As this study has shown, there is always more to code than what is visible on a surface level. Finding these connections to past software and present dependencies requires digging into the social and technical relationships of different projects to help understand why each arose, how they

influence each other through aspects like the hidden labor within them, and what the future of a project might hold based on its structures.

REFERENCES

About—Node.js. (2023). Node.Js. https://nodejs.org/en/about

Alfadel, M., Costa, D. E., Shihab, E., & Adams, B. (2022). On the Discoverability of npm Vulnerabilities in Node.js Projects. *ACM Transactions on Software Engineering and Methodology*, 3571848. https://doi.org/10.1145/3571848

Andreessen, M., & Bina, E. (1994). NCSA Mosaic: A Global Hypermedia System. *Internet Research*, *4*(1), 7–17. https://doi.org/10.1108/10662249410798803

Anthropy, A. (2009). *Town*. Internet Archive. https://web.archive.org/web/20090721140212/http://www.auntiepixelante.com/town/

Anthropy, A. (2012). *Rise of the videogame zinesters: How freaks, normals, amateurs, artists, dreamers, dropouts, queers, housewives, and people like you are taking back an art form* (Seven Stories Press 1st ed). Seven Stories Press.

Arnott, L. (2023). *Harlowe*. https://foss.heptapod.net/games/harlowe/-/tree/branch/default

Arnott, L. (2013, December 4). *Versions of Twine—Twine Wiki*. Internet Archive. https://web.archive.org/web/20131204150747/http://twinery.org/wiki/versions_of_twine

Arnott, L. (2022, June 20). *Harlowe 3.3 manual*. https://twine2.neocities.org/

Ashkenas, J. (2022). _.unescape(). In *Underscore*. https://underscorejs.org/docs/modules/unescape.html

Ashkenas, J. (2009). *Underscore.js*. https://cdn.jsdelivr.net/gh/jashkenas/underscore@0.4.0/underscore.js

Ashkenas, J., & Gonggrijp, J. (2022). *Underscore-esm.js* (1.13.6). https://underscorejs.org/docs/underscore-esm.html

Baccaris, G. C. (2020). *The Twine® Grimoire, Vol. 1*. Itch. https://gcbaccaris.itch.io/grimoire-one

Baccaris, G. C. (2021). *The Twine® Grimoire, Vol. 2*. Itch. https://gcbaccaris.itch.io/grimoire-two

Backus, J. W., Mitchell, L. B., Beeber, R. J., Nelson, R. A., Best, S., Nutt, R., Goldberg, R., Herrick, H. L., Sayre, D., Hughes, R. A., Sheridan, P. B., Stern, H., & Ziller, I. (1956). *The FORTRAN Automatic Coding System for the IBM 704 EDPM: Programmer's Reference Manual*. International Business Machines Corporation (IBM).

Berge, P., Cox, D., Murray, J., & Salter, A. (2022). Adventures in TwineSpace: An Augmented Reality Story Format for Twine. In M. Vosmeer & L. Holloway-Attaway (Eds.),

Interactive Storytelling (Vol. 13762, pp. 499–512). Springer International Publishing. https://doi.org/10.1007/978-3-031-22298-6_32

Berners-Lee, T. (1992). *Review—Erwise*. https://www.w3.org/History/19921103-hypertext/hypertext/Erwise/Review.html

Birsan, A. (2021). Dependency Confusion: How I Hacked Into Apple, Microsoft and Dozens of Other Companies. *Medium*. https://medium.com/@alex.birsan/dependency-confusion-4a5d60fec610

Bogost, I. (2007). *Persuasive games: The expressive power of videogames*. MIT Press.

Bolter, J. D., & Joyce, M. (1987). Hypertext and creative writing. Proceeding of the ACM Conference on Hypertext - HYPERTEXT '87, 41–50. https://doi.org/10.1145/317426.317431

Bolter, J. D. (2001). *Writing space: Computers, hypertext, and the remediation of print* (2nd ed). Lawrence Erlbaum Associates.

Bolter, J. D., & Grusin, R. (2003). *Remediation: Understanding new media*. MIT Press.

Boylorn, R. M., & Orbe, M. P. (Eds.). (2021). *Critical autoethnography: Intersecting cultural identities in everyday life* (Second edition). London.

Bratton, B. H. (2015). *The stack: On software and sovereignty*. MIT Press.

Brown, J. J. (2015). *Ethical programs: Hospitality and the rhetorics of software*. University of Michigan Press.

browserify. (2010). GitHub. https://github.com/browserify/browserify

Cassel, D. (2022). The Most Famous Comment in Unix History: "You Are Not Expected to Understand This." In T. Bosch (Ed.), *You are not expected to understand this: How 26 lines of code changed the world* (pp. 63–68). Princeton University Press.

Cassel, D. (2017). "You are Not Expected to Understand This": Unix's Most Notorious Code Comment. *The New Stack*. https://thenewstack.io/not-expected-understand-explainer/

Castells, M. (2010). *The rise of the network society*. Wiley-Blackwell.

Chat with Emil Filla & Ľudovít Fulla. (2018, July 10). *Slovak National Gallery*. https://fillafulla.sng.sk/

Chouhan, A. (2023). *[Bug] Infinite loop in colors.js*. Marak/Colors.Js - Issues. https://github.com/Marak/colors.js/issues/345

Chromium. (2023). Google Git. https://chromium.googlesource.com/chromium/src.git

.closest(). (2023). In *jQuery API Documentation*. https://api.jquery.com/closest/

Cox, D. (2013a). *Category—Twine Tuesday*. Digital Ephemera. https://videlais.com/category/twine-tuesday/

Cox, D. (2017a). *Snowman*. GitHub. https://github.com/videlais/snowman

Cox, D. (2022a). *Snowman 1.0.0.* Archive of Twine 2 Story Formats. GitHub. https://github.com/videlais/twine2-story-formats-archive/blob/02a4405ccf7f54a37a49cc0cd129344de914c3bf/snowman-1.0.0/format.js

Cox, D. (2020a). Static Echoes: Exploring the Life and Closing of the Free Twine Hosting Service, philome.la. *Electronic Literature Organization Conference 2020*. https://stars.library.ucf.edu/elo2020/asynchronous/talks/17

Cox, D. (Ed.). (2021a). Story Formats. In *Twine Cookbook*. https://twinery.org/cookbook/introduction/story_formats.html

Cox, D. (2013b). *Twine 1.3.5: A short tutorial on using Twine (in Windows)*. YouTube. https://www.youtube.com/watch?v=JWgI3yYCfgI

Cox, D. (Ed.). (2021b). Twine 2.0.4—Release Notes. In *Twine Cookbook*. https://twinery.org/cookbook/releasenotes/twine2/2.0.4.html

Cox, D. (Ed.). (2021c). *Twine Cookbook*. https://twinery.org/cookbook/

Cox, D. (2022b). We Make How We Learn: The Role of Community in Authoring Tool Longevity. In C. Hargood, D. E. Millard, A. Mitchell, & U. Spierling (Eds.), *The Authoring Problem* (pp. 65–72). Springer International Publishing. https://doi.org/10.1007/978-3-031-05214-9_5

Cox, D. (2016). *Twine 2.0 Tutorials*. YouTube. https://www.youtube.com/playlist?list=PLlXuD3kyVEr5tlic4SRe6ZG-R9OyS1T4d

Cox, D. (2017b). *Twine 2.1 Tutorials*. YouTube. https://www.youtube.com/playlist?list=PLlXuD3kyVEr7bucZtQPpOZHjbUuGKaf2V

Cox, D. (2018). *Videlais/snowman@4f7525f*. GitHub. https://github.com/videlais/snowman/commit/4f7525fa3f7481f8f8a157d6c5ee0be86696e954

Cox, D. (2019a). An Oral History of Twee. *Digital Ephemera*. https://videlais.com/2019/06/08/an-oral-history-of-twee/

Cox, D. (2019b). *Learning Twine 2.2*. YouTube. https://www.youtube.com/playlist?list=PLlXuD3kyVEr6T06I71pxVxJn5KK1B0rWO

Cox, D. (2020b). *Learning Twine 2.3*. YouTube. https://www.youtube.com/playlist?list=PLlXuD3kyVEr5jWoG0oDygKWOgFC3qrKN-

Cox, D., Berge, P., Murray, J., & Salter, A. (2022). *TwineSpace: A Twine 2 story format supporting 3D models and mixed reality projects.* (1.0.0). Zenodo. https://doi.org/10.5281/ZENODO.6915351

Cox, D., Klimas, C., & Edwards, T. M. (2019). *Twine 2 HTML Output (v1.0.1)* (1.0.1). https://github.com/iftechfoundation/twine-specs/blob/master/twine-2-htmloutput-spec.md

cssnano/cssnano. (2015). GitHub. https://github.com/cssnano/cssnano

Dahl, R. (2010). *Joyent & Node*. Google Groups. https://groups.google.com/g/nodejs/c/lWo0MbHZ6Tc

Daiute, C., Cox, D., & Murray, J. T. (2021). Imagining the Other for Interactive Digital Narrative Design Learning in Real Time in Sherlock. In A. Mitchell & M. Vosmeer (Eds.), *Interactive Storytelling* (Vol. 13138, pp. 454–461). Springer International Publishing. https://doi.org/10.1007/978-3-030-92300-6_46

Dalton, J.-D., Cambridge, K., & Bynens, M. (2012). *Lo-Dash* (0.1.0). Internet Archive. https://web.archive.org/web/20120512131557/http://lodash.com/

.data(). (2023). In *jQuery API Documentation*. https://api.jquery.com/data/

de Souza, C., Froehlich, J., & Dourish, P. (2005). Seeking the source: Software source code as a social and technical artifact. *Proceedings of the 2005 International ACM SIGGROUP Conference on Supporting Group Work - GROUP '05*, 197. https://doi.org/10.1145/1099203.1099239

Debian. (2023). https://www.debian.org/

DeMarco, M. C. (2018a). *Twine Story/Proofing Format Catalog*. GitHub. https://github.com/tweecode/format-catalog

DeMarco, M. C. (2018b). *A Catalog of Twine Story Formats*. http://mcdemarco.net/tools/hyperfic/twine/catalog/

Demšar, J., Curk, T., Erjavec, A., Gorup, Č., Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., & Zupan, B. (2013). Orange: Data Mining Toolbox in Python. *Journal of Machine Learning Research*, *14*, 2349–2353.

D'Ignazio, C., & Klein, L. F. (2020). *Data feminism*. The MIT Press.

Document Object Model (DOM). (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model

Ducheneaut, N. (2005). Socialization in an Open Source Software Community: A Socio-Technical Analysis. *Computer Supported Cooperative Work (CSCW)*, *14*(4), 323–368. https://doi.org/10.1007/s10606-005-9000-1

ECMA-262, 6th edition, June 2015. (2015). Ecma International. https://www.ecma-international.org/wp-content/uploads/ECMA-262_6th_edition_june_2015.pdf

ECMA-262, 15th edition, June 2023. (2023). Ecma International. https://tc39.es/ecma262/

Edwards, T., & Cox, D. (2021). *Twee 3 Specification (v3.0.2)*. Interactive Fiction Technology Foundation. https://github.com/iftechfoundation/twine-specs/blob/master/twee-3-specification.md

Edwards, T. M. (2019). *SugarCube v2.* GitHub. https://github.com/tmedwards/sugarcube-2

Edwards, T. M. (2021). *SugarCube v2 Documentation*. https://www.motoslave.net/sugarcube/2/docs/

Element: Click event. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Web/API/Element/click_event

Ellison, C. (2013, April 10). Anna Anthropy and the Twine revolution. *The Guardian*. https://www.theguardian.com/technology/gamesblog/2013/apr/10/anna-anthropy-twine-revolution

event. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Web/API/Event

event.target. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Web/API/Event/target

Everyday Types. (2023). In *TypeScript Documentation*. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html

FDOS Kernel. (2018). FDOS. GitHub. https://github.com/FDOS/kernel

Foley, M. J. (2012). *Who built Microsoft TypeScript and why*. ZDNET. https://www.zdnet.com/article/who-built-microsoft-typescript-and-why/

Ford, M. (2016). *Writing interactive fiction with Twine*. Que.

Foucault, M. (1969). *The archaeology of knowledge*. Vintage Books.

Foucault, M. (1994). *The order of things: An archaeology of the human sciences*. Vintage Books.

Friedhoff, J. (2014). Untangling Twine: A Platform Study. *DiGRA 2013 - Proceedings of the 2013 DiGRA International Conference: DeFragging Game Studies*. http://www.digra.org/wp-content/uploads/digital-library/paper_67.compressed.pdf

Galloway, A. R. (2006). *Protocol: How control exists after decentralization*. MIT Press.

Ghosh, R. A., & Prakash, V. V. (2000). The Orbiten free software survey. *First Monday*, *5*(7). https://doi.org/10.5210/fm.v5i7.769

Goodin, D. (2021). *A new type of supply-chain attack with serious consequences is flourishing*. Ars Technica. https://arstechnica.com/gadgets/2021/03/more-top-tier-companies-targeted-by-new-type-of-potentially-serious-attack/

Goswami, P., Gupta, S., Li, Z., Meng, N., & Yao, D. (2020). Investigating The Reproducibility of NPM Packages. *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 677–681. https://doi.org/10.1109/ICSME46990.2020.00071

Halliday, J., & Harband, J. (2012). *defined.* Inspect JS. GitHub. https://github.com/inspect-js/defined

Handbook. (2012). In *The TypeScript Handbook*. https://www.typescriptlang.org/docs/handbook/intro.html

Harvey, P. H., & Pagel, M. D. (1991). *The comparative method in evolutionary biology*. Oxford University Press.

Hertel, G., Niedner, S., & Herrmann, S. (2003). Motivation of software developers in Open Source projects: An Internet-based survey of contributors to the Linux kernel. *Research Policy*, *32*(7), 1159–1177. https://doi.org/10.1016/S0048-7333(03)00047-7

Historical yearly trends in the usage statistics of JavaScript libraries for websites, March 2023. (2023). Q-Success. https://w3techs.com/technologies/history_overview/javascript_library/all/y

History of the Mozilla Project. (2023). Mozilla. https://www.mozilla.org/en-US/about/history/

HTML: HyperText Markup Language. (2023). In *Mozilla Developer Network.* https://developer.mozilla.org/en-US/docs/Web/HTML

Hudson, L. (2017, October 25). *Twine, the Video-Game Technology for All*. https://www.nytimes.com/2014/11/23/magazine/twine-the-video-game-technology-for-all.html

Introduction to events. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks/Events

Jockers, M. L. (2013). *Macroanalysis: Digital methods and literary history*. University of Illinois Press.

jQuery: New Wave Javascript. (2006, February 3). Internet Archive. https://web.archive.org/web/20060203025710/http://jquery.com/

jQuery 1.0 – Alpha Release. (2006, June 30). *jQuery 1.0 – Alpha Release | Official jQuery Blog*. https://blog.jquery.com/2006/06/30/jquery-10-alpha-release/

The jQuery Foundation and Standards. (2014, January 15). *Official jQuery Blog*. https://blog.jquery.com/2014/01/15/the-jquery-foundation-and-standards/

Kaczynski, F. (n.d.). *Adventures.* Retrieved April 26, 2023, from http://adventures.longwelwind.net/

Kaczynski, F. (2017a). *Adventures.* GitHub. https://github.com/Longwelwind/adventures

Kaczynski, F. (2017b, September 15). *Longwelwind/adventures@2619a50*. GitHub. https://github.com/Longwelwind/adventures/commit/2619a50138502cf16ea8868e13d554c89c01e1a9

Kieffer, R., & Brigante, F. (2022). *Npmgraph* (2.9.1). https://npmgraph.js.org/

Kittler, F. A. (1999). *Gramophone, film, typewriter*. Stanford University Press.

Klimas, C. (2023a). *Chapbook.* GitHub. https://github.com/klembot/chapbook

Klimas, C. (2006, March 28). Code and Other Oddments. *Gimcrack'd: An Exhibition of Narrative and Machinery*. https://web.archive.org/web/20060328165735/http://gimcrackd.com/etc/src/

Klimas, C. (2008). *Writing With Twee*. Internet Archive. https://web.archive.org/web/20080123111435/http://gimcrackd.com/etc/doc/

Klimas, C. (2013, November 14). *Homepage—Twine*. Internet Archive. https://web.archive.org/web/20131114011254/https://twinery.org/

Klimas, C. (2015a). Twine 2, now a real(?) app. *Chris Klimas*. https://chrisklimas.com/blog/2015-04-13-twine-2-now-a-real-app/

Klimas, C. (2015b, April 15). *2.0.4—Releases*. GitHub. https://github.com/klembot/twinejs/releases/tag/2.0.4

Klimas, C. (2018). *Snowman 1.3 Update*. GitHub. https://github.com/videlais/snowman/tree/c26f06979d92288ff41bb09d0cec8fa96d77dfa7

Klimas, C. (2023b). *2.6.0—Releases*. GitHub. https://github.com/klembot/twinejs/releases/tag/2.6.0

Knuth, D. E. (1997). *The art of computer programming* (3rd ed). Addison-Wesley.

Koenitz, H., & Eladhari, M. P. (2019). Challenges of IDN Research and Teaching. In R. E. Cardona-Rivera, A. Sullivan, & R. M. Young (Eds.), *Interactive Storytelling* (Vol. 11869, pp. 26–39). Springer International Publishing. https://doi.org/10.1007/978-3-030-33894-7_4

Krill, P. (2009, April 4). *JavaScript creator ponders past, future*. Developer World - InfoWorld. https://web.archive.org/web/20090404173354/http://www.infoworld.com/d/developer-world/javascript-creator-ponders-past-future-704

Latour, B. (2007). *Reassembling the social: An introduction to Actor-Network-Theory*. Oxford Univ. Press.

Maass, W. (2004). Inside an open source software community: Empirical analysis on individual and group level. *"Collaboration, Conflict and Control: The 4th Workshop on Open Source Software Engineering" W8S Workshop - 26th International Conference on Software Engineering*, *2004*, 65–70. https://doi.org/10.1049/ic:20040267

Manovich, L. (2013). *Software takes command: Extending the language of new media*. Bloomsbury.

Marino, M. C. (2020). *Critical code studies: Initial methods*. The MIT Press.

Mauchly, J. W. (1982). Preparation of Problems for EDVAC-Type Machines. In B. Randell (Ed.), *The Origins of Digital Computers* (pp. 393–397). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-61812-3_31

McCullough, B. (2018). *How the Internet happened: From Netscape to the iPhone* (First edition). Liveright Publishing Corporation.

Moretti, F. (2007). *Graphs, maps, trees: Abstract models for a literary history* (Paperback ed). Verso.

Moretti, F. (2013). *Distant reading*. Verso.

Mozilla Search—Firefox. (2023). Search Fox. Mozilla Central. https://searchfox.org/mozilla-central/source

MS-DOS v1.25 and v2.0 Source Code. (2018). Microsoft. GitHub. https://github.com/microsoft/MS-DOS

Noble, S. U. (2018). *Algorithms of oppression: How search engines reinforce racism*. New York University Press.

npm (2023a). Node Package Manager (9.7.1). https://www.npmjs.com/

npm: Threats and Mitigations (2023b). npm Documentation. https://docs.npmjs.com/threats-and-mitigations

Ofoeda, J., Boateng, R., & Effah, J. (2019). Application Programming Interface (API) Research: A Review of the Past to Inform the Future. *International Journal of Enterprise Information Systems*, *15*(3), 76–95. https://doi.org/10.4018/IJEIS.2019070105

.on(). (2023). In *jQuery API Documentation*. https://api.jquery.com/on/

Ong, W. J. (2012). *Orality and literacy: The technologizing of the word* (30th anniversary ed.; 3rd ed). Routledge.

Our Mission and Goals. (n.d.). Interactive Fiction Technology Foundation. https://iftechfoundation.org/mission/

Parikka, J. (2015). *A geology of media*. University of Minnesota Press.

Performance. (2023). In *TypeScript*. https://github.com/microsoft/TypeScript/wiki/Performance

Porpentine. (2012). *CYBERQUEEN*. Interactive Fiction Database. https://ifdb.org/viewgame?id=8rib3ksuex22l5pl

Qix. (2015). *Node-is-arrayish.* GitHub. https://github.com/Qix-/node-is-arrayish

Quinn, Z. (2013). *Depression Quest: An Interactive (non)Fiction About Living with Depression*. http://www.depressionquest.com/

Reference Manual: FORTRAN II for the IBM 704 Data Processing System (X4115.2007). (1958). International Business Machines Corporation (IBM); Computer History Museum.

Resig, J. (2006). *BarCampNYC Wrap-up*. https://johnresig.com/blog/barcampnyc-wrap-up/

Rival, X., & Yi, K. (2020). *Introduction to static analysis: An abstract interpretation perspective*. The MIT Press.

Robertson, A. (2021, March 10). *Text Adventures: How Twine remade gaming*. The Verge. https://www.theverge.com/c/22321816/csk-twine

Ropek, L. (2022). *An Open-Source Developer Just Nuked Two Apps, Causing Chaos*. Gizmodo. https://gizmodo.com/an-open-source-developer-just-caused-a-whole-lot-of-cha-1848331944

Roth, E. (2022, January 9). *Open source developer corrupts widely-used libraries, affecting tons of projects*. The Verge. https://www.theverge.com/2022/1/9/22874949/developer-corrupts-open-source-libraries-projects-affected

Ryan, J. (2010). *A history of the Internet and the digital future*. Reaktion Books.

Sallis, P., Aakjaer, A., & MacDonell, S. (1996). Software forensics: Old methods for a new science. *Proceedings 1996 International Conference Software Engineering: Education and Practice*, 481–485. https://doi.org/10.1109/SEEP.1996.534037

Salter, A., & Moulthrop, S. (2021). *Twining: Critical and creative approaches to hypertext narratives*. Amherst College Press.

Salter, A., & Murray, J. (2014). *Flash: Building the Interactive Web*. MIT Press.

Salter, A., & Stanfill, M. (2020). *A portrait of the auteur as fanboy: The construction of authorship in transmedia franchises*. University Press of Mississippi.

Schlueter, I. (2011). *Isaacs/inherits.* GitHub. https://github.com/isaacs/inherits

Scope. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Glossary/Scope

Shalizi, C. (2011). Graphs, Trees, Materialism, Fishing. In J. Goodwin & J. Holbo (Eds.), *Reading Graphs, maps & trees: Responses to Franco Moretti* (pp. 131–154). Parlor Press.

Sharma, A. (2022). *Dev corrupts NPM libs "colors" and "faker" breaking thousands of apps*. Bleeping Computer. https://www.bleepingcomputer.com/news/security/dev-corrupts-npm-libs-colors-and-faker-breaking-thousands-of-apps/

Sink, E. (2003). *Memoirs From the Browser Wars*. https://ericsink.com/Browser_Wars.html

Soloman, R. (2013). LAST IN, FIRST OUT. *Amodern*. https://amodern.net/article/last-in-first-out/

Squires, M. (2010). *Colors.js* (1.4.0). GitHub. https://github.com/Marak/colors.js

Squires, M. (2022, January 7). *Marak/colors.js@074a0f8*. GitHub. https://github.com/Marak/colors.js/commit/074a0f8ed0c31c35d13d28632bd8a049ff136fb6

Stanfill, M. (2019). *Exploiting fandom: How the media industry seeks to manipulate fans*. University of Iowa Press.

Story Formats—Twine Wiki. (2014). Twine Wiki. Internet Archive. https://web.archive.org/web/20140426042129/http://twinery.org/wiki/story_format

Text Mining—Keyword-Based Text Document Scoring. (n.d.). Orange Datamining. Bioinformatics Laboratory, University of Ljubljana. Retrieved June 4, 2023, from https://orangedatamining.com/workflows/Text-Mining/page/2/

The Future Frontier: Computing on NCSA Mosaic's 10th Anniversary. (2009). NCSA. https://web.archive.org/web/20090620091511/https://www.ncsa.illinois.edu/Conferences/MosaicEvent/

this. (2023). In *Mozilla Developer Network*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Tornhill, A. (2015). *Your code as a crime scene: Use forensic techniques to arrest defects, bottlenecks, and bad design in your programs*. The Pragmatic Bookshelf.

Twine Committee. (n.d.). Interactive Fiction Technology Foundation. Retrieved May 26, 2023, from https://iftechfoundation.org/committees/twine/

Ubuntu. (2023). Ubuntu. https://ubuntu.com/

Usage Statistics and Market Share of jQuery for Websites, March 2023. (2023). Q-Success. https://w3techs.com/technologies/details/js-jquery

Usage Statistics and Market Share of Underscore for Websites, March 2023. (2023). Q-Success. https://w3techs.com/technologies/details/js-underscore

van Kemenade, P. (2022a). *What is Trialogue?*. GitBook. https://phivk.gitbook.io/trialogue/

van Kemenade, P. (2022b, February 6). *Release 0.0.8*. Trialogue. GitHub. https://github.com/phivk/trialogue/releases/tag/v0.0.8

van Kemenade, P. (2022c, July 8). *Credits, thanks & license*. Trialogue. GitBook. https://phivk.gitbook.io/trialogue/

Vee, A. (2017). *Coding Literacy: How Computer Programming is Changing Writing*. MIT Press.

Wardrip-Fruin, N. (2011). Digital Media Archaeology: Interpreting Computational Processes. In E. Huhtamo & J. Parikka (Eds.), *Media Archaeology: Approaches, Applications, and Implications* (pp. 302–322). University of California Press.

Wexelblat, R. L. (Ed.). (1981). *History of programming languages*. Academic Press.

Wheeler, D. J. (1952). The use of sub-routines in programmes. *Proceedings of the 1952 ACM National Meeting (Pittsburgh) - ACM '52*, 235–236. https://doi.org/10.1145/609784.609816

Woods, R. (2022, June 15). *Internet Explorer is dead—A look back at Microsoft's browser history*. XDA Developers. https://www.xda-developers.com/internet-explorer-browser-history/

Zimmermann, M., Staicu, C.-A., Tenny, C., & Pradel, M. (2019). Small World with High Risks: A Study of Security Threats in the npm Ecosystem. *28th USENIX Security Symposium (USENIX Security 19)*, 995–1010. https://www.usenix.org/conference/usenixsecurity19/presentation/zimmerman