Università
della
Svizzera
italiana

**Software
Institute**

SENSORIAL SOFTWARE EVOLUTION COMPREHENSION

Gianlorenzo Occhipinti

July 2022

Supervised by
Prof. Dr. Michele Lanza

Co-Supervised by
Dr. Csaba Nagy

Dr. Roberto Minelli

Abstract

The comprehension of software evolution is essential for the understandability and maintainability of systems. However, the sheer quantity and complexity of the information generated during systems development make the comprehension process challenging.

We present an approach based on the concept of synesthesia (the production of a sense impression relating to one sense by stimulation of another sense), which represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution. The approach is exemplified in SYN, a web application that enables sensorial software evolution comprehension. We applied SYN on real-life systems and present several insights and reflections.

Dedicated to my past, present, and future
supporters…

Acknowledgements

One year ago, when I decided to enroll as a double degree student at USI, I could have never imagined how that could change my life. First of all, I would like to thank my advisor Prof. Dr. Michele Lanza. I feel fortunate to have been your student. You opened up my eyes, giving me the chance to see the world from a different perspective. I was fascinated by your senior experience and by your philosophical thoughts.

This endeavor would not have been possible without the support of my two co-advisors, Dr. Csaba Nagy and Dr. Roberto Minelli. Thank you for all the time that you dedicated to me. It was essential for the success of this work.

Words cannot express my gratitude to my family and my parents, Emanuele and Vera. I have been far away from home for five years, but I always felt your unconditional love and care from you. Without your support, I would never be able to come this far.

I am incredibly grateful to my love Biancamaria and her wonderful family. Your warmth made me feel part of the family from the very first moment. I will never forget all the things that you've done for me.

Thanks should also go to Andrea. Together, we strengthened ourselves through our stunning university path. I owe special thanks to many people who contributed to this work by making my life much better. Thank you, Daniel, Ottavio, Federico, and Carmen, for all the cool stuff we did together, which was always the cause of fun, joy, and enthusiasm.

To my Sicilian friends Francesco, Alessio, Marco, Angelo, Daniele, Davide, and many others who are too numerous to mention: in different ways and times, you all have played an essential role in my life.

Thanks should also go to my friends Emanuele, Davide, and Samuele. I have been playing and working with you since I was 15, when we started our Minecraft community. Thanks to you, my passion took shape and became my job in life.

Lastly, I would like to thank myself for the determination and commitment I have always put into projects I have worked on. As my grandmother said, I was a rough diamond, but now it is time to shine.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 1971, Dijkstra made an analogy between computer programming and art [15]. It stated that it is not essential to learn how to compose software; instead, it is necessary to develop its own style and implications. Software development is a complex process involving many people and development tools and their interactions. This is one of the multiple factors that characterize software complexity. Modern software systems are characterized by sheer size and complexity. Software maintenance takes up most of a system's cost. It is hard to quantify the impact of software maintenance on the global cost of the software. Researchers estimated it to be between 50% and 90% [11, 49, 18, 48]. Many factors influence the maintenance cost; among these is the understanding activity needed to perform maintenance tasks [8]. Comprehending software evolution is essential for systems' understandability and, consequently, maintainability. However, the sheer quantity and complexity of the information generated during systems development challenge the comprehension process. Lehman and Belady, in 1985, were among the first to observe that maintaining a software system becomes a more complex activity over time [35]. The term "software evolution" was used for the first time in their set of laws. One goal of software evolution analysis is to identify potential defects in the system's logic or architecture.

Numerous techniques have been presented in the literature to facilitate program comprehension [33, 9, 52, 60, 1, 54]. The main challenge they have to deal with is identifying relevant aspects to be presented so that the user does not get lost in the myriad of information. Software visualization is a specialization of information visualization with a focus on software [34].

In literature, many visualization techniques have been presented to support a complex software system's analysis. Usually, a massive quantity of multivariate evolutionary data needs to be depicted. Several tools have been proposed in the literature to do that [40]. The central idea of this thesis is a visualization technique to support evolution analysis, complemented by an auditive depiction of the evolution. Many researchers studied the advantages given by audio as a communication medium [2, 55, 3, 38, 37]. The term "program auralization" refers to communicating information about the program in an auditory way [13].

We present an approach based on synesthesia, the production of a sense impression relating to one sense by stimulation of another sense. The approach represents the evolutionary process through an interactive visual depiction of the evolving software artifacts complemented by an auditive portrayal of the evolution. Our technique models and mines large git repositories. The approach is exemplified in SYN, a web application that enables sensorial software evolution comprehension.

1.1 Contributions

We summarize the main contributions of this work as follows:

- an approach to mine and model the history of a system's evolution;
- an interactive, visual 3D representation of evolving software artifacts;
- an auralization approach to compose music based on a system's evolution;
- a supporting tool that implements our approach;
- case studies on open-source systems.

1.2 Document Structure

This document is organized as follows:

- In Chapter 2, we describe state of the art in software visualization, repository mining, and software auralization. We look at evolution models and 2D/3D visualizations.
- In Chapter 3, we describe our approach based on mining software repositories and modeling their evolution, visualizing evolving software artifacts, and auralizing the system's evolution.
- In Chapter 4, we present SYN, a supporting tool that implements our approach.
- In Chapter 5, we preliminary validate our approach by analyzing five open-source software systems and reporting our findings.
- In Chapter 6, we summarize our work and discuss possible directions for future work.

Chapter 2

State of the Art

2.1 Software Visualization

Software maintenance and evolution are essential parts of the software development lifecycle. Both require that developers deeply understand their system. Mayrhauser and Vans defined *program comprehension* as a process of "*knowledge to acquire new knowledge*" [57]. Generally, programmers possess two types of knowledge: general and software-specific knowledge. Software comprehension aims to increase this specific of the system and can leverage software visualization techniques for this purpose. Software visualization supports understanding software systems by visually presenting various information about them, e.g., their architecture, source code, or behavior. Stasko et al. [19] conducted a study in 1998 that shows how visualization arguments human memory since it works as external cognitive aid and thus, improves thinking and analysis capabilities.

There are cases when software visualization can be used to aid the analysis activity. For example, when programmers need to comprehend the architecture of a system [43], when researchers analyze version control repositories [21], or to support developers' activities [36].

According to Butler et al. [5] there are three categories of visualization:

- Descriptive visualization. Widely used for education purposes, the visualization is used to present data to other people.
- Explorative visualization. Used to discover the nature of the data being analyzed. With this visualization, the users do not necessarily know what they are looking for; e.g. they explore possibilities for improvement.
- Analytical visualization. Adapted when we need to find something known in the available data.

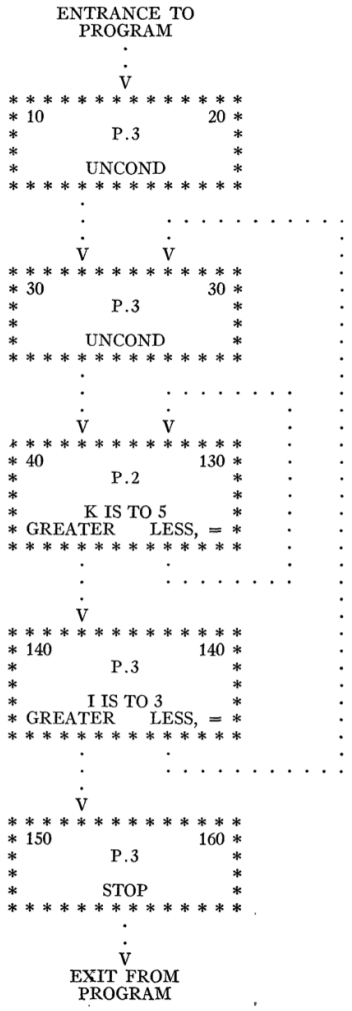


FIGURE 2.1: Flowchart presented by Haibt in 1959

Software visualization approaches vary with respect to two dimensions: the level of abstraction and the visualized data. According to the type of the data, we can classify visualization as:

- Evolutionary visualizations: Depicts the development history of a system. Mainly used to find the cause of problems in software.
- Static visualizations: Used to present information extracted with static analysis of the software. It provides information about the structure of the system.

- Dynamic visualizations: Shows results of dynamic instrumentation of the software execution. It provides information about the behavior of the system.

Moreover, the level of abstraction can be classified as follows:

- Code-level visualization: where fine granted sourcecode information is highlighted, such as the lines of code.
- Design-level visualization: used to visualize self-contained source code entities, such as classes in object-oriented systems.
- Architectural-level visualization: depicts the system architecture and the relationships among its components.

The earliest software visualization techniques in the literature used 2D diagrams. For example, Haibt, used them already in 1959 and provided a graphical outline of a program and its behavior with flowcharts [22]. As shown in Figure 2.1, they were 2D diagrams that described the execution of a program. He wrapped each statement in a box, representing the control flow with arrows. Ten years later, Knuth confirmed the effectiveness of flowcharts [31]. He found that programs around that time were affected by a lack of readability. Therefore, he introduced a tool to generate visualizations from the software documentation automatically. Nassi and Schneiderman [42], in 1973, introduced the Nassi–Shneiderman diagram (NSD), shown in Figure 2.2, to represent the structure of a program. The diagram was divided into multiple sub-blocks, each with a given semantic based on its shape and position. In the 80s, researchers followed two main directions for software visualization. The first was the source code presentation. For example, Hueras and Ledgard [23] then Waters [58] developed techniques to format the source code with a prettyprinter. The second direction was the program behavior, mainly for educational purposes. One of that period's most prominent visualization systems was Balsa-II [4] (Figure 2.3). Balsa-II was a visualization system that, through animations, displayed the execution of an algorithm. Programmers could customize the view and the control execution of the algorithm to understand them with a modest amount of effort. The program was domain-independent, and learners could use it with any algorithm. Around the end of the 80s, Müller et al. [41] released Rigi (Figure 2.4), a tool to visualize large programs. It exploited the graph model, augmented with abstraction mechanisms to represent systems components and relationships.

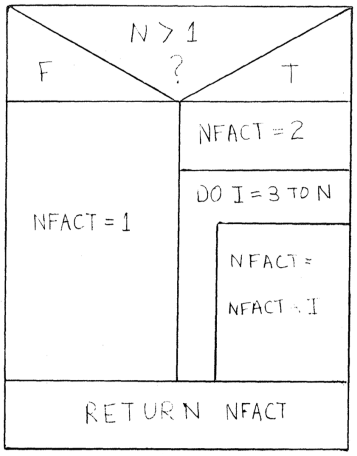


FIGURE 2.2: NSD of the factorial function.

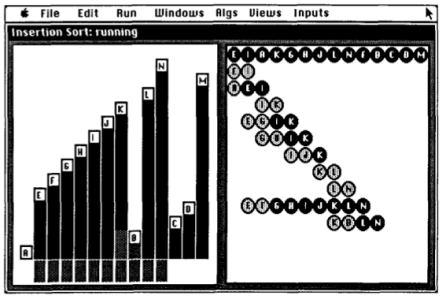


FIGURE 2.3: Balsa-II

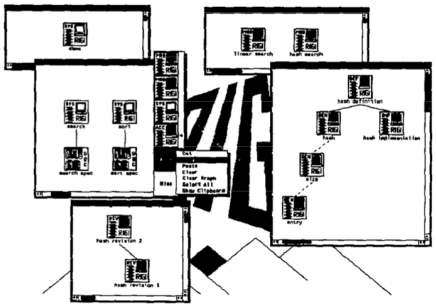


FIGURE 2.4: Rigi

In the 1990s, there was more interest in the field of software visualization. In 1992, Erik et al. introduced a new technique to visualize line-oriented statistics [16]. It was embodied in Seesoft (Figure 2.5), a software

visualization system to analyze and visualize up to 50,000 lines of code simultaneously. In their visualization, each line was mapped to a thin row. Each row was associated with a color that described a statistic of interest, e.g., red rows are those most recently changed, and blue are those least recently changed.

One year later, De Pauw et al. [12] introduced Jinsight (Figure 2.6), a tool to provide animated views of object-oriented systems' behavior.

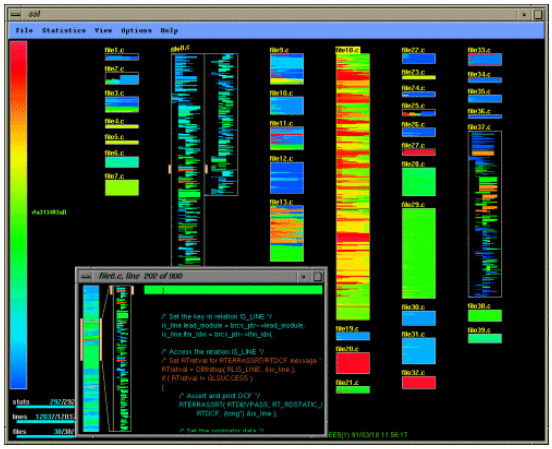


FIGURE 2.5: Seesoft

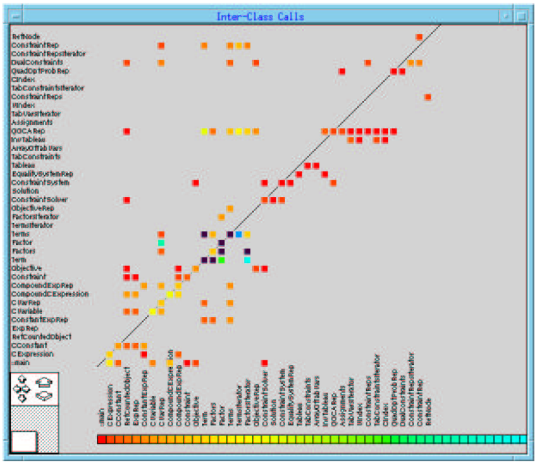


FIGURE 2.6: Jinsight

That period was favorable also for experimenting with novel research directions for visualization, such as 3D visualization and Virtual Reality. In 1998, Chuah and Erick [6] proposed three techniques to visualize project data. They leveraged glyphs, a graphical object that represents data through visual parameters. The first technique was the Timewhell glyph (Figure 2.7), to visualize time-oriented information (number of lines of code, number of errors, number of added lines). The second technique was the 3D wheel glyph (Figure 2.8). It encoded the same attributes of the time wheel and used the height to encode time. Infobug (Figure 2.9) glyph was the last technique, where each glyph was composed of four parts, each representing essential data of the system, such as time, code size, and the number of added, deleted, or modified code lines.

FIGURE 2.7: Timewhell

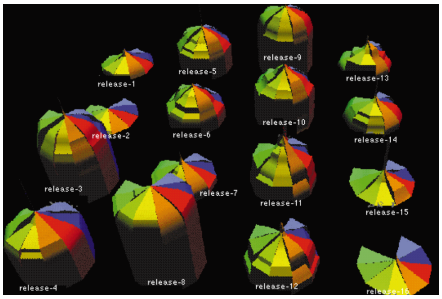


FIGURE 2.8: 3D wheel

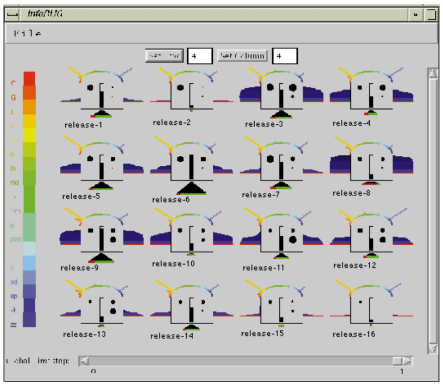


FIGURE 2.9: Infobug

Also in 1998, Young and Munro [61] explored representations of software for program comprehension in VR.

Finally, in 1999, Jacobson et al. [24] introduced what we now know as de facto the standard language to visualize the design of a system: UML.

At the beginning of the 21st century, thanks to the spread of version control systems and the open-source movement, visualizing a software system's evolution became a more feasible activity thanks to publicly accessible system information. As a result, many researchers focused their work on software evolution visualization.

Lanza [33] introduced the concept of the Evolution Matrix (Figure 2.10). It was a way to visualize the evolution of software without dealing with a large amount of complex data. Furthermore, this approach was agnostic to any particular programming language. The Evolution Matrix aimed to display the evolution of classes in object-oriented software systems. Each column represented a version of the system, and each row represented a different version of the same class. Cells were filled with boxes whose size depended on evolutionary measurements. The evolution matrix allows us to make statements on the evolution of an object-oriented system at both the system and class levels. For example, in Figure 2.11, at system level, we are able to recover information regarding the size of the system, the addition and removal of classes, and the growth or stagnation phases in the evolution.

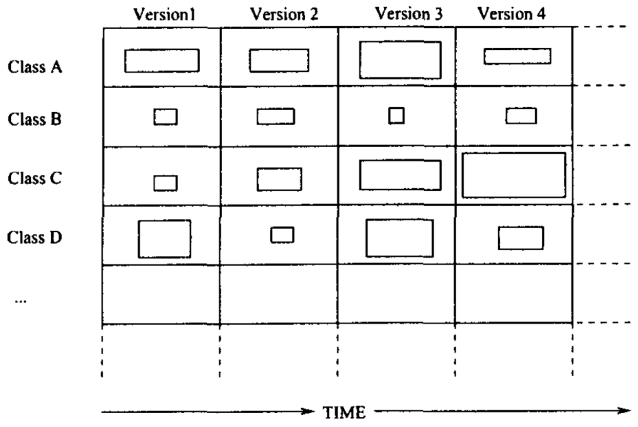


FIGURE 2.10: A schematic display of the Evolution Matrix

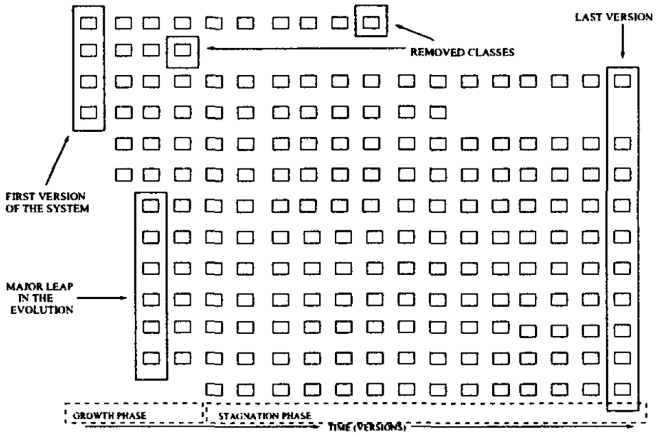


FIGURE 2.11: Some characteristics of the Evolution Matrix

Taylor and Munro [53] demonstrated that it was possible to use the data contained in a version control repository to visualize the evolution of a system. They developed Revision Tower, a tool that showed change information at the file level. Pinzger et al. [44] visualized the evolution of a software system through Kivat diagrams. RelVis, the tool they developed, depicted a multivariate visualization (Figure 2.12) of the evolution of a system. It was built on Kiviat diagrams, designed to visualize multivariate data such as source code and evolution metrics. During the same year, Ratzinger et al. presented EvoLens [45], a visualization approach and tool to explore evolutionary data through structural and temporal views. Langelier et al. [32] investigated the interpretation of a city metaphor [30] to add a new level of knowledge to the visual analysis. D'Ambros and Lanza [9] introduced the Discrete-Time Figure concept (Figure 2.13). It was a visualization technique that embedded historical and structural data in a simple figure. Their approach depicted relationships between the histories of a system and bugs. They presented Evolution Radar [10] (Figure 2.14), a novel approach to visualizing module-level and file-level logical coupling information.

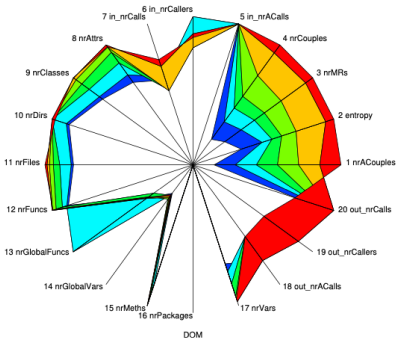


FIGURE 2.12: RelVis

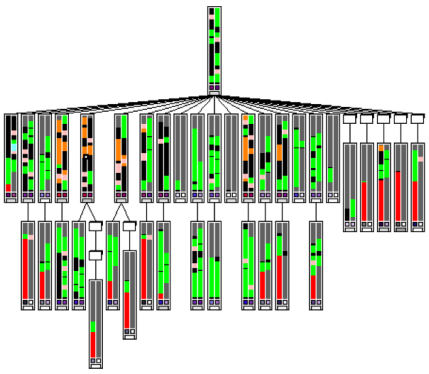


FIGURE 2.13: Tree of Discrete Time Figures

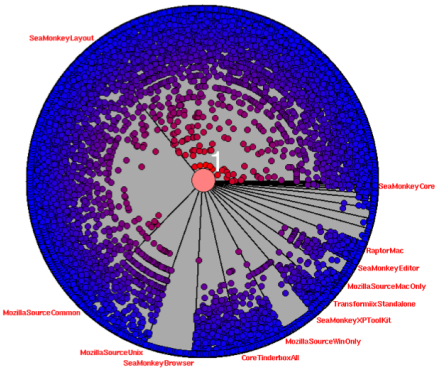


FIGURE 2.14: Evolution Radar

Steinbrückner and Lewerentz [52] described a three-staged visualization approach to visualize large software systems. Their visualization was supported by a tool called Evo-Streets. Each stage of their approach yields a specific model that evolved through the stages. In the first stage, they created a model containing all the structure of a software system and its evolution. In the second stage, geometrics information is added, such as the city layout or landscape elevation. Finally, in the third stage, they added projections, colors, and symbols to the visualization.

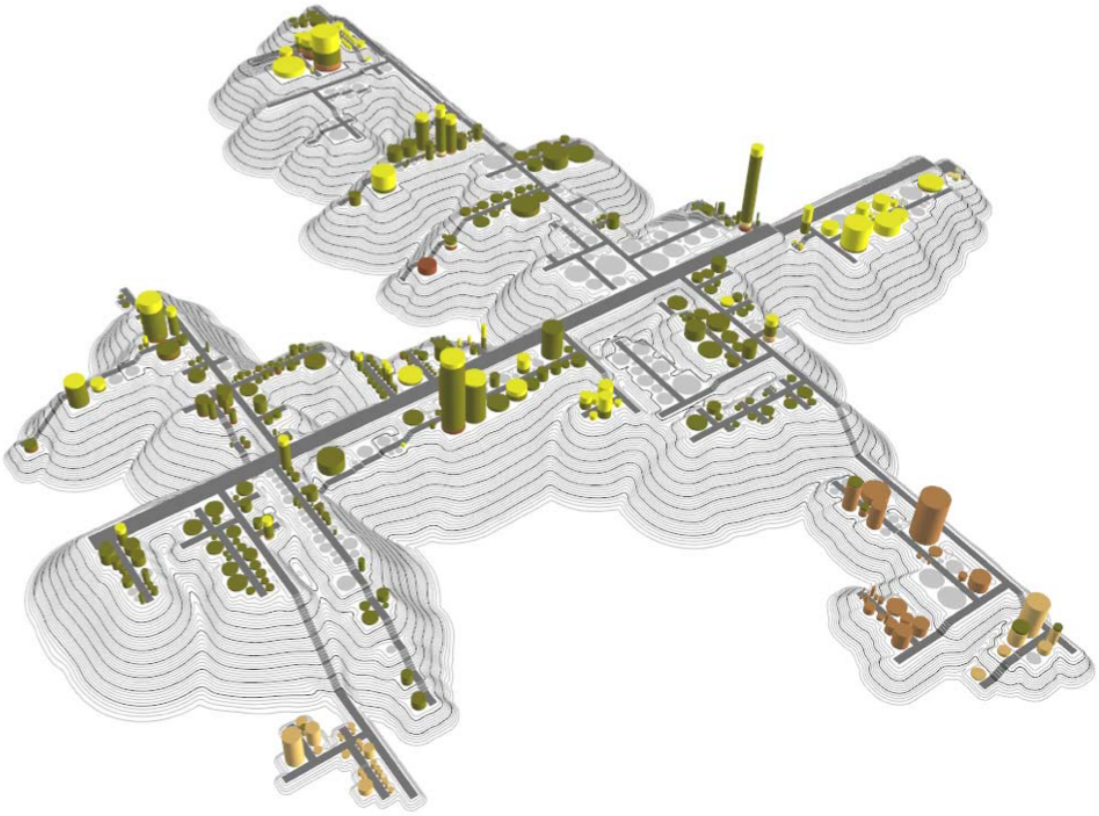


FIGURE 2.15: Evo-Streets

Wettel revised the city metaphor to represent metrics meaningfully [60]. In his thesis, he represented packages as districts and classes as buildings. The metaphor was used for various purposes, e.g., reverse

engineering, program comprehension, software evolution, or software quality analysis. He claimed that the city metaphor brought visual and layout limitations; for example, not all visualization techniques fit well. Under those circumstances, he preferred simplicity over accuracy, so he obtained a simple visual language that facilitated data comprehension. His approach was implemented as a software visualization tool called CodeCity (Figure 2.16).

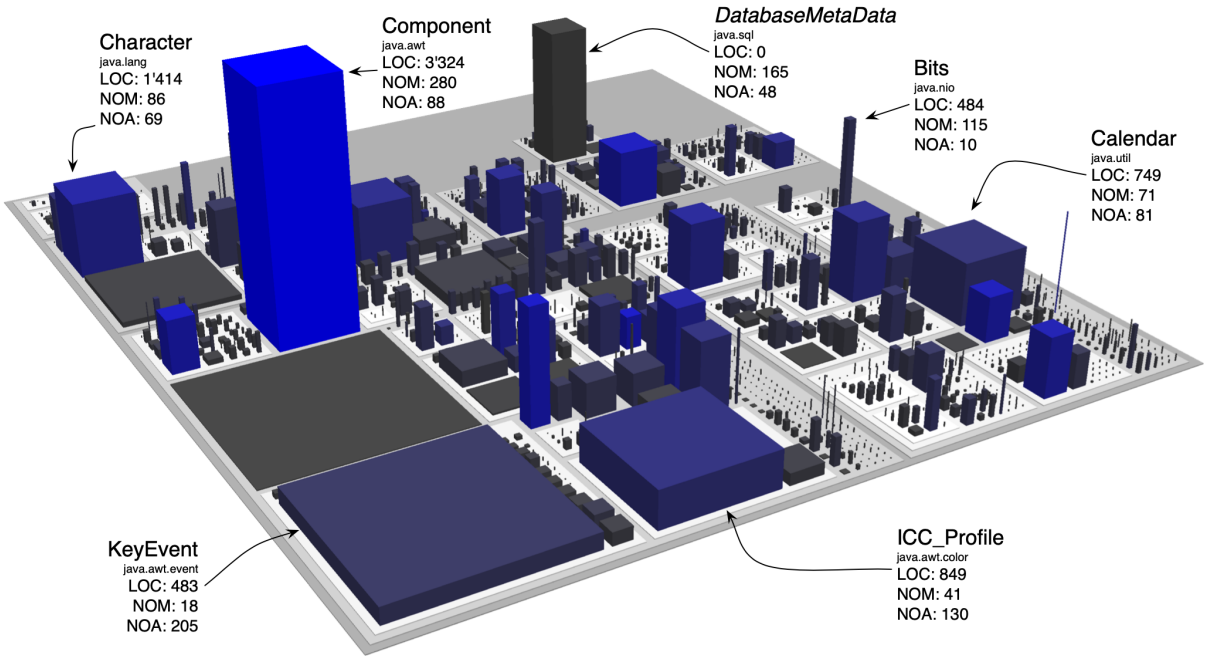


FIGURE 2.16: CodeCity

Ens et al. [17] applied visual analytics methods to software repositories (Figure 2.18). His approach helped users comprehend co-evolution information by visualizing how source and test files were developed together. Kapec et al. [27] proposed a graph analysis approach with augmented reality. They made a prototype of a tool that provided a graph-based visualization of software, and then they studied some interaction methods to control it with augmented reality. Schneider et al. [47] presented a tool, CuboidMatrix (Figure 2.19), that employed a space-time cube metaphor to visualize a software system. A space-time cube is a well-known 3D representation of an evolving dynamic graph. Merino et al. [39] aimed to augment software visualization with gamification. They introduced CityVR (Figure 2.17), a tool that displays a software system through the city metaphor with a 3D environment. Working with virtual reality, they scaled the city visualization to the physically available space in the room. Therefore, developers needed to walk to navigate the system.

Khaloo et al. [28] revised the idea of gamification with a 3D park-like environment. They mapped each class in the codebase with a facility. The wall structure depended on the class' constituent parts, e.g., methods and signatures. Finally, we mention Alexandru et al., who proposed a method to visualize software structure and evolution with reduced accuracy and a fine-grained highlighting of changes in individual components [1]. Figure 2.20 shows a view of Evo-Clocks, the tool they developed.

2.2 Software Evolution Analysis

Version control systems track historical data in repositories about the evolution of a system. Git has become the most popular version control system today since Linus Torvald introduced it in 2005. Many collaboration platforms (e.g., GitHub and GitLab) also rely on it. With the increased popularity of such platforms,

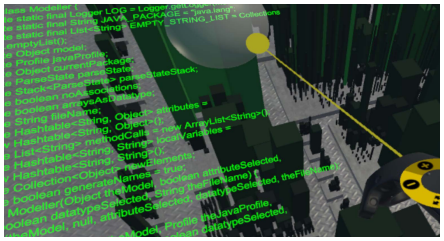


FIGURE 2.17: CityVR

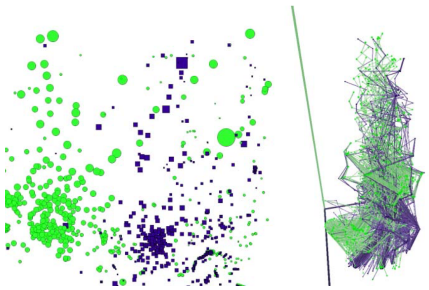


FIGURE 2.18: ChronoTwigger

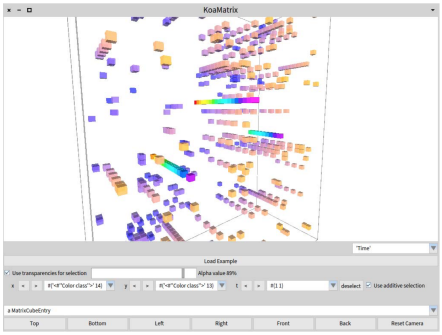


FIGURE 2.19: CuboidMatrix

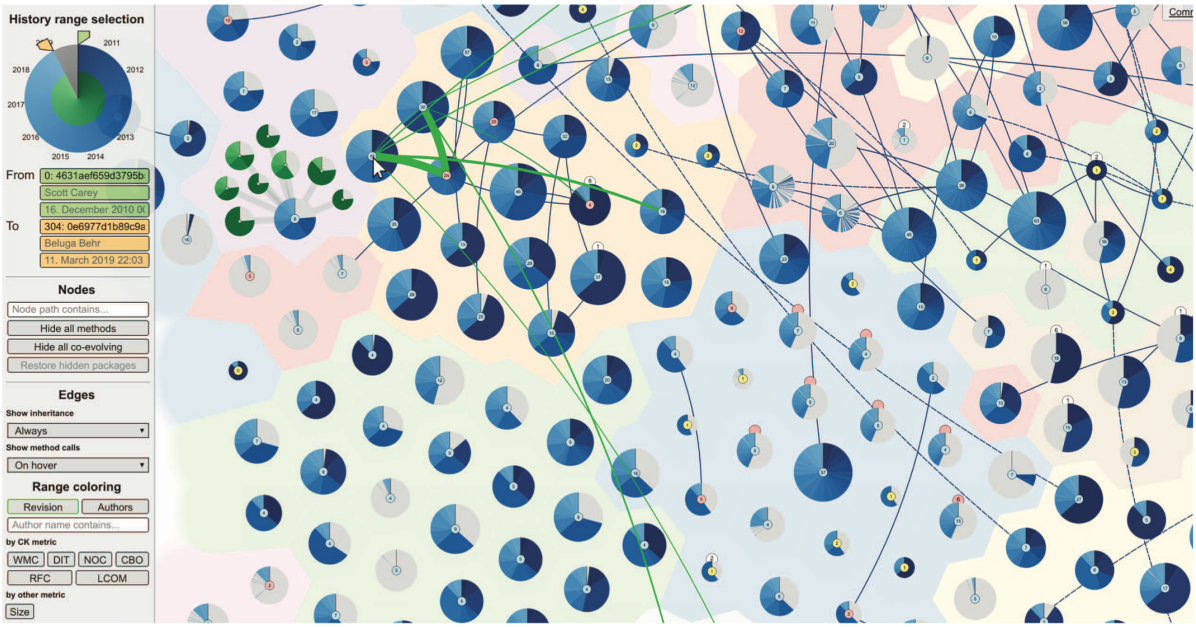


FIGURE 2.20: Evo-Clocks

millions of open-source systems are developed publicly. They have also become popular targets for Mining Software Repositories (MSR) research.

D'Ambros et al. [54] presented an analysis and visualization techniques to understand software evolution. They developed an approach based on a Release History Database (RHDB). It is a database that stores historical information about source code and bugs. The strength of RHDB (Figure 2.21) was the association between historical versions of flies and bugs. Having this information stored in a database, they were able to run an evolution analysis to obtain information such as the number of developers needed to fix a bug.

Finally, they concluded two main challenges in MSR:

- Technical challenge: repositories contain a sheer amount of data, posing scalability problems.
- Conceptual challenge: how to leverage the collected data. Most of the approaches to visualizing software evolution have unanswered questions about the effectiveness of the comprehension.

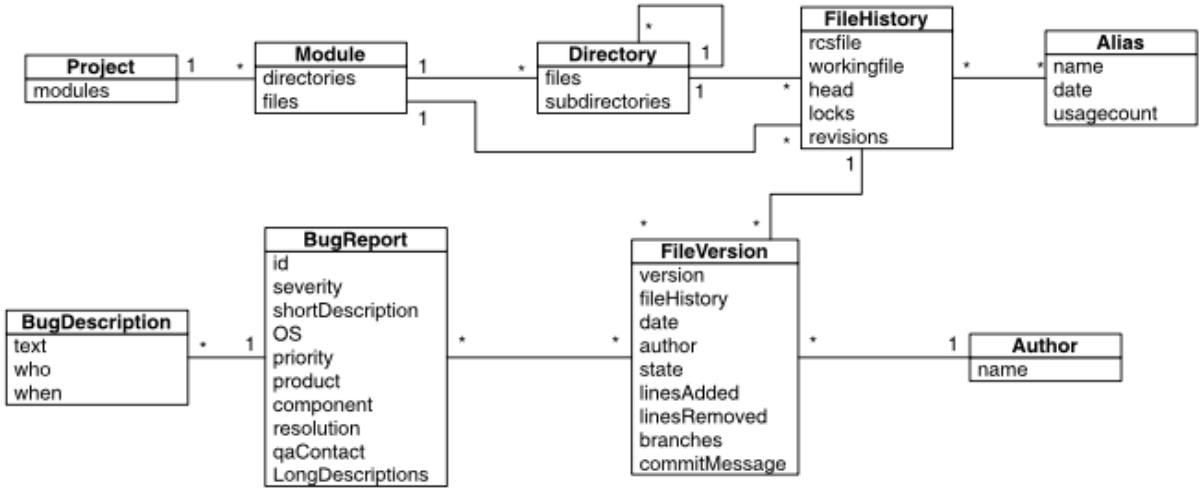


FIGURE 2.21: RHDB

In 2022, there are around 200 million GitHub repositories.[1] Even if it seems a promising data source, Kalliamvakou et al. raised some issues with its mining [26]. For example, they found that a repository does not always match a project. A reason for this can be that most repositories had very few commits before becoming inactive. Over 70% of the GitHub projects were personal when they did their research, and some weren't used for software development. Finally, the last perils they raised were related to GitHub features improperly used by developers. They considered only projects with a good balance between the number of commits, the number of pull requests, and the number of contributors to find actively developed repositories.

Spadini, Aniche, and Bacchelli [51] developed a Python framework called PyDriller, enabling users to mine software repositories. Their tool can be used to extract information about the evolution of a software system from a git repository.

We also mention the work of Salis and Spinellis [46]. They introduced RepoFS, a tool that allows navigating a git repository as a file system. Their approach sees commits, branches, and tags as a separate directory tree. Figure 2.22 shows an example of a repository data structure.

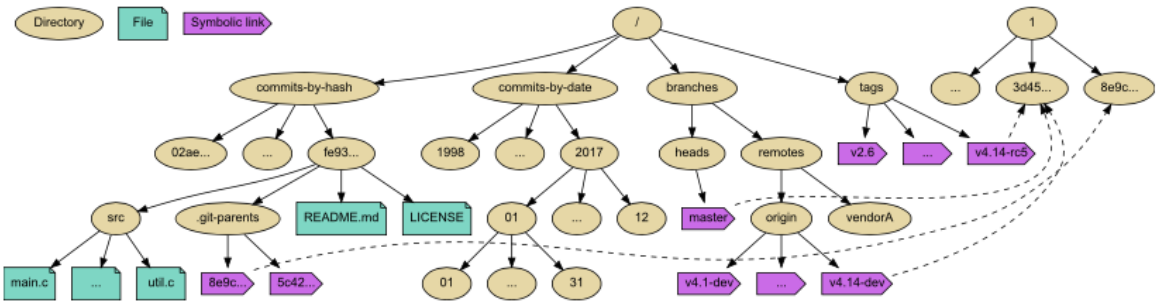


FIGURE 2.22: RepoFS

Clem and Thomson [7], members of the semantic code team at GitHub, built a static analyzer of repositories to implement symbolic code navigation. That feature was released on GitHub in 2020 years ago and lets developers click on a name identifier to navigate to the definition of the selected entity. They were looking for a solution without scalability problems. Moreover, they built the symbolic navigation feature around the following ideas:

[1] https://en.wikipedia.org/wiki/GitHub

- Zero configuration needed by the owner of a repository.
- Incrementality of the process. There was no need to process the entire repository for every commit made by a developer. Instead, they analyzed only the files changed.
- Language agnosticism of the static analysis.

Working on that feature, they recognized the difficulty of scaling a static analysis to large and rapidly changing codebases. Nevertheless, their idea was to have an agnostic static analyzer, but they could not reach this goal, and they were forced to implement it for nine programming languages.

2.3 Program Auralization

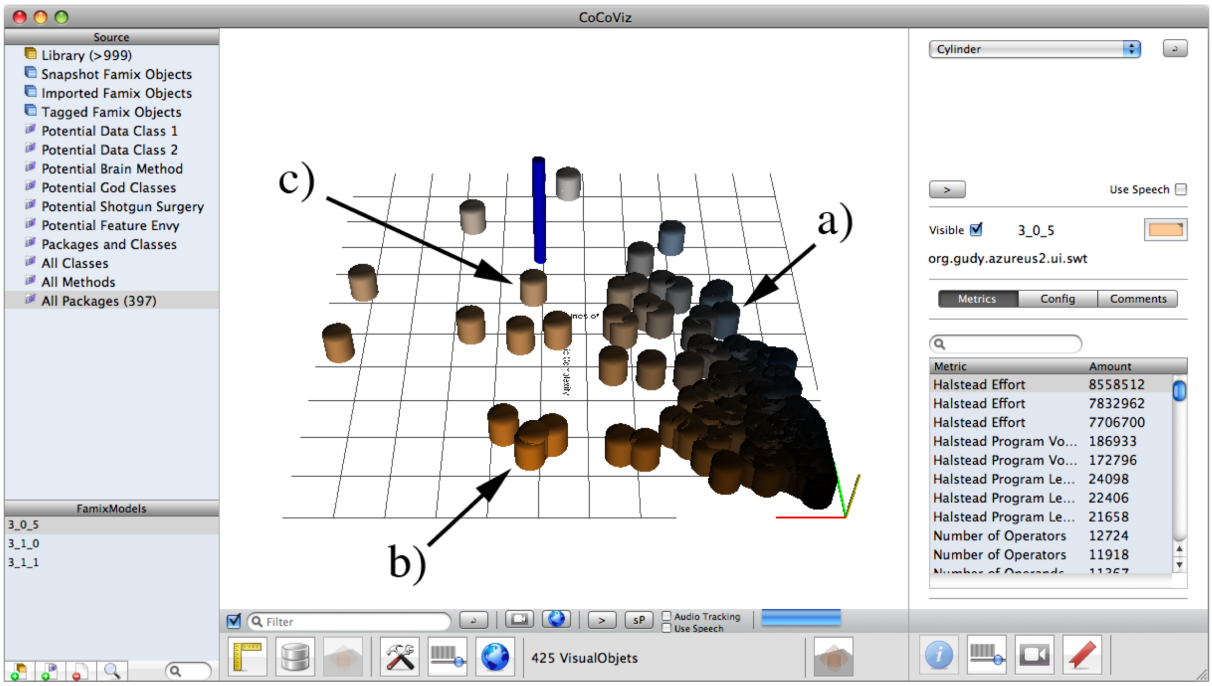


FIGURE 2.23: CocoViz

External auditory representations of programs (known as "program auralization") is a research field getting even more interest in recent years.

Sonnenwald et al. made one of the first attempts [50]. They tried to enhance the comprehension of complex applications by playing music and special sound effects. This approach was supported by a tool called InfoSound. It was mainly adopted to understand the program's behavior.

Many other researchers followed this first technique. To cite some of them, DiGiano and Baecker [14] made LogoMedia, a tool to associate non-speech audio with program events while the code is being developed. Jameson [25] developed Sonnet audio-enhanced monitoring and debugging tool. Alty and Vickers [56] had a similar idea. Using a structured musical framework, they could map the execution behavior of a program to locate and diagnose software errors.

Despite the usefulness of these tools, they adopted an essential kind of mapping, and thus they had a limited musical representation. Vickerts [55] found the necessity of a multi-threaded environment to enhance the comprehension given by the musical representation. He proposed adopting an orchestral model of families of timbres to enable programmers to distinguish between different activities of different threads.

Boccuzzo and Gall [3] supported software visualization with sonification, the use of non-speech audio to convey information or perceptualize data. They used audio melodies to improve navigation and comprehension of their tool, called CocoViz (Figure 2.23). Their ambient audio software exploration approach exploited audio to describe an entity's position in space intuitively. Thanks to the adoption of surround sound techniques, the observers perceived the origin of an audio source so it could adjust their navigation in the visualization. Each kind of entity played a different sound based on mapping criteria.

McIntosh et al. [38] explored the use of parameter-based sonification to produce a musical interpretation of the evolution of a software system. Their technique mapped musical rests to an inactive period of development, consonance, and dissonance to interesting phenomena (like co-changing components). Finally, Mancino and Scanniello [37] presented an approach to transforming source code metrics into a musical score that can be both visualized and played.

2.4 Conclusion

We have seen many different techniques and tools focused on visualizing the source code of software systems, their evolution, or their metrics. Our evolution focuses on the evolution of a software system and how its metrics change over its history. In contrast to what some tools did, we do not focus on the evolution of code bugs.

The codebase of a system is composed of a group of files. In our approach, each file represents a system entity that mutates over time. It is not based on a previously identified metaphor, such as CodeCity or CityVR with the city metaphor. We created a new layout where the position of each entity is defined by its discovery time.

At present, git has become the standard tool for version control. Having this in mind, we aim to find a suitable model to represent the histories of mined git repositories. Therefore, we created a model inspired by the EvolutionMatrix, but with adjustments to make it work with git.

As Clem and Thomson [7] team did, we propose a scalable approach that works with large repositories. It differs from what they did because we are not focused on a semantic analysis of the source code; instead, we extract source code metrics. Moreover, our technique is purely language-agnostic.

Finally, we extended our approach with an auralization approach to compose music based on a system's evolution. Conversely to state-of-the-art tools, we used a multithreaded environment to play the musical notes. Whereas CocoViz used audio melodies to support the navigation of space created for the visualization, we mapped sounds to the magnitudes of changes in a given moment.

Chapter 3

Approach

Comprehending the evolution of a software system is a complex activity, mainly because of the sheer amount of data and its complexity. The term "software evolution" was coined for the first time by Lehman in 1985 in a set of laws [35]. He stated that the complexity of a system is destined to increase over time as the system needs to be adapted to its evolutionary environments. To be maintained, software systems need to be comprehended by developers, and this activity can be supported with software visualization.

The development activity is often supported by a Version Control System (VCS) software for tracking and managing file changes. VCSs have been widely adopted for the last 40 years. Revision Control System (RCS) is one of the oldest, and it was introduced in 1980. Consequently, between 1990 and 2020, developers introduced several VCSs. The most important ones were Concurrent Versions System (CVS) introduced in 1990, Perforce (1995), Subversion (2000), Mercurial, and Git (2005).

One of the most adopted ones is Git, introduced in 2005 by Linus Torvalds[1]. Millions of repositories use it on GitHub and GitLab. For this reason, we focused on systems versioned with this protocol. Git is a versioning control system that tracks all the changes made to every system file. Internally git holds all the information we need to reconstruct the history of a repository.

In this chapter, we present our sensorial approach to visualizing a software system using a visual and auditive depiction of its evolution. To fulfill this purpose, we leverage synesthesia, the production of a sense impression relating to one sense by stimulation of another sense. Moreover, we also present how we reconstruct and model the history of a repository.

In this chapter, we present the three main steps of our approach as follows:

- first, we model the system's evolution;
- next, we visualize it;
- finally, complement the visualization with an auditive portrayal of evolutionary data.

[1] https://github.com/git/git/commit/e83c5163316f89bfbde7d9ab23ca2e25604af290

3.1 Evolution Model

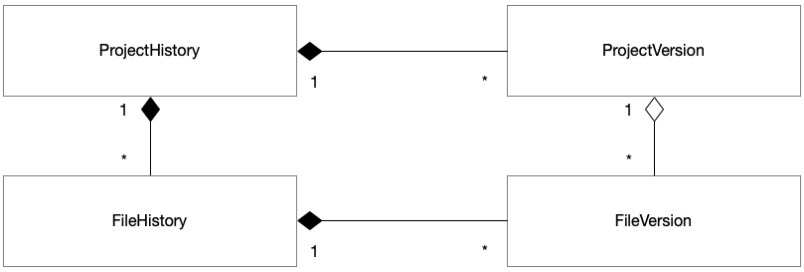


FIGURE 3.1: Evolutionary Model

Various approaches have been proposed to analyze aspects of software evolution. In 2005, Tudor Girba presented Hismo [20], a model centered around the notion of history as a first-class entity. Our approach is based on this work. The need to develop a novel evolutionary model comes from the fact that Hismo was designed to work with another versioning system: Subversion (SVN). There are several differences between SVN and git. In terms of design, the most important is how they track changes. SVN works with the concept of "snapshot" while git works with the concept of "commits". In SVN, when a file has been changed, a new revision of the whole system is created, and consequently, the number of revisions is incremented. In contrast, in git, only the modified files would get committed, and thus we don't have a new snapshot of the system every time. Therefore, we took Hismo as the starting point of our model and adapted it to the git protocol. The Hismo model was based on three concepts:

- Snapshot. A representation of the entity whose evolution is studied.
- Version. A representation of a system's version. It defines the time when a snapshot was made.
- History. An entity that holds a set of Versions.

We replaced the concept of Snapshot with a FileVersion. It represents the version of a file at a particular point in time. Instead of being related to every version of the system, it is related only to the Versions when the file was updated. Moreover, we made a distinction between File entities and Project entities. So, we mapped the concept of History to FileHistory and the idea of Version to ProjectVersion. The entity responsible for holding both of them is called ProjectHistory. Figure 3.1 depicts the relationships among these concepts. To summarize, these are the four main concepts of our evolutionary model:

- **ProjectHistory**: represents the history of a repository. It holds two sets: a set of FileHistories and a set of ProjectVersions.
- **FileHistory**: represents a file inside the repository. We consider each file as an entity of the system. Even if the entity's name or location is changed, our model will treat it as the same. So, our approach is resilient to renaming and moving activities. Each FileHistory holds a set of FileVersions, each representing a different version of the entity at a particular point in time.
- **ProjectVersion**: represents a commit or a version of the system. For each changed file inside a commit, the respective ProjectVersion contains a FileVersion representing that change. A ProjectVersion holds contextual information about the commit, such as its timestamp, hash, and message.
- **FileVersion**: represents the version of a file at a particular point in time. It is responsible for holding all the evolutionary information of an entity, such as the last action on a file.

3.1.1 Historical information retrieval

To model the history of a repository, we need to extract the historical information from git. Git works with the concept of branches. Each branch can be seen as a different repository timeline. Usually, developers use branches to develop features and merge the developed code in a branch that contains the stable codebase. They create a "merge commit" to do that. Each time developers create a new git commit, they deploy a new version of the system that records all the changes made to the commits' tracked files. Internally, in git, all git stores all the commits as nodes of a commit-tree. The root node represents the repository's first commit and has no parents. All the other nodes represent the commits made during the whole lifecycle of the repository. Each commit usually has only one parent representing the previous commit. There is one case where a commit might have more than one parent: merges commits.

Each repository should have a branch containing stable, production-ready code as a convention. Usually, this branch is named "main" or "master". In our approach, we aim to analyze the timeline of this stable branch. We start from the root of the commit tree, which represents the initial commit, and then we traverse the whole tree. We do not consider "merge commits" during this process since they already incorporate previous commits, and thus they would be considered twice. Once we have extracted all the valid commits that reside on the stable branch, we need to extract all the representative information for a ProjectVersion.

Git can recognize the following file actions:

- **ADD**. A file is added to the repository.
- **DELETE**. A file has been removed from the repository.
- **MODIFY**. The contents of a file have been modified.
- **RENAME**. A file's name has been changed but the file remained in the same parent directory.
- **MOVE**. A file was moved from one location to another. This action is detected whether the file's name remains the same.

From a commit, we could also extract additional information such as the name of the file being modified, the action made on a file, the number of lines added and removed, and the file paths before and after the changes. We used the commit's information to track all the paths of an entity. We can update the entity path when it was renamed or moved to follow it during its lifecycle.
When we reconstruct the history of a repository, each FileHistory starts with a FileVersion representing an ADD action. During the lifecycle of a repository, files can be deleted. In this case, the last FileVersion held by a deleted FileHistory represents a DELETE action.

Figure 3.2 shows an example of building an instance of the evolution model. First, we create a ProjectHistory with a set of ProjectVersions and a set of FileHistories. After that, we start to traverse the repository's commit tree. For each commit, we create a new ProjectVersion that represents a new version of the system. We inspect the commit's changelog and create a new FileVersion for each list entry. Every time we find a newly added file in the changelog of a commit, we create a new FileHistory. In the example, in version 1, three new files were added to the repository (A, B, C). Thus, three new FileHistories were created. Each change was mapped to a FileVersion (FV) and consequently added to the respective FileHistory and ProjectVersion. We did the same for ProjectVersions 2 and 3.

3.1.2 Partial historical representation

We had as a goal to develop a scalable approach to mine software repositories. This way, it should always be possible to analyze large repositories in an acceptable amount of time. In other words, our approach

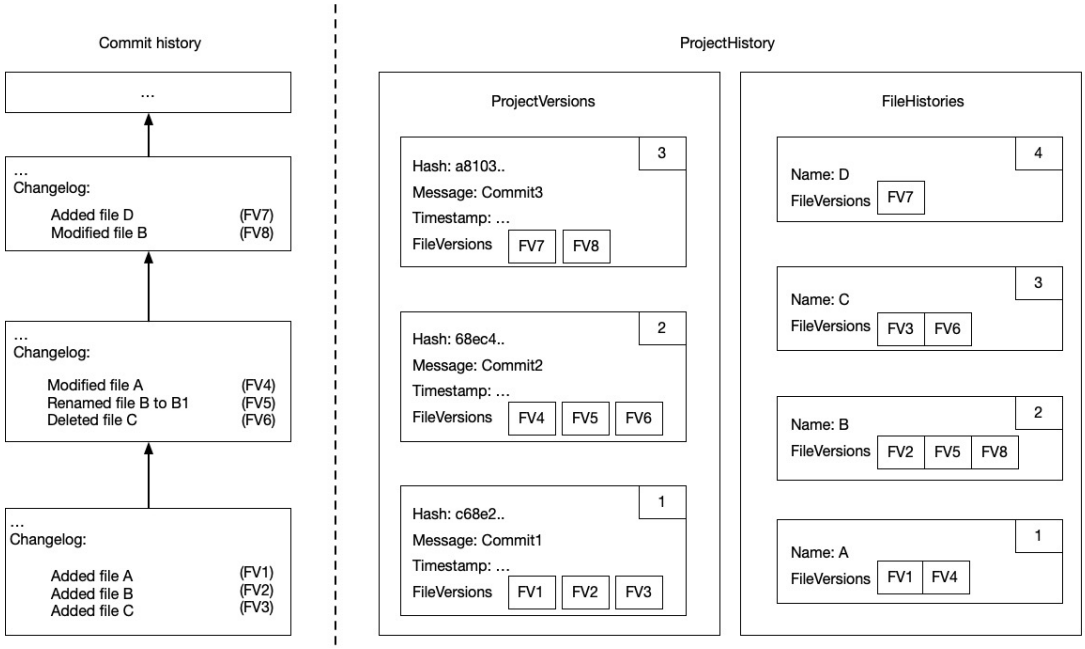


FIGURE 3.2: Rebuilding history example

needs to be scalable. GitHub host the code of some notorious open-source systems, such as LibreOffice, Elasticsearch, and Linux. They all have more than 500,000 commits each. Thus, we cannot aim to reconstruct their histories with a single analyzer, as it would take too much time. At the time of writing, the repository of Linux had 1,090,563 commits. To checkout from one commit to another, we could assume that git needs one second. As a result, just to navigate through the whole history of Linux, we would need 11 days. Moreover, in this simple estimation, we are omitting the time the analyzer needs to extract metrics from every file on each version.

To overcome this mining issue, we present a scalable approach based on the concept of partial history. A partial history holds information about a specific range of time of the ProjectHistory. It can be seen as a subset of a ProjectHistory. We can split the repository's history into multiple parts, each represented by a partial record. Then when all the analyses are completed, we merge them to reconstruct the whole story of the repository.

Figure 3.3 shows an example PartialHistory representation. We split the commit tree into multiple chunks and then run the analysis on each. In the end, the final history will be represented by merging all the PartialHistories. Nonetheless, we can build PartialHistories in parallel, but we cannot do the same for the final History, because the final merge needs to be done sequentially. The sequence needs to follow the order of the commit tree. In Figure 3.3, for example, PartialHistory1 represents the history from commit 1 to commit 10, PartialHistory2 represents the history from commit 11 to commit 20, and PartialHistory3 represents the history from commit 21 to commit 30. Therefore, the commit order is respected if we merge them in this order: 1, 2, 3. The result of a single analysis and a parallel analysis must be identical. To ensure that, we need to pay attention to the merge operations of our analysis. When we merge the history of a repository with a partial history, we need to preserve the characteristics of our model. In particular, if FileHistory is already present in our history, we do not have to duplicate it, but instead, we need to update it.

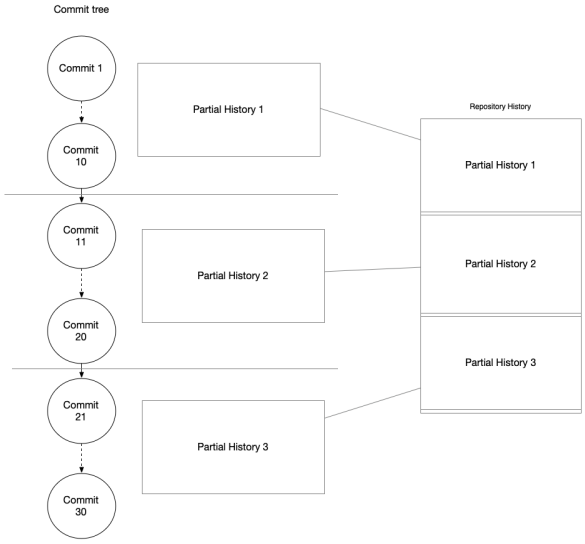


FIGURE 3.3: Partial history example

FileType	Metrics
File	SIZE
Textual	LOC, LinesAdded, LinesRemoved, SIZE
Binary	SIZE
Java	SLOC, LOC, LinesAdded, LinesRemoved, SIZE
JPEG	SIZE

TABLE 3.1: Example of metrics collected and inherited for each FileType

Evolutionary metrics

Every version of the system holds a set of files. Each file is represented by a FileVersion, which is part of a FileHistory. For the visualization, we collect metrics representing the files' states. Since we aim to have a language-agnostic approach, we selected only language-agnostic metrics. However, the set of metrics can be easily extended. We defined a taxonomy to classify and categorize all the files in a system. Each category is then mapped to a set of metrics. Metrics can also be inherited from parent categories.

Figure 3.4 shows an example of a possible taxonomy definition. Table 3.1 shows the final set of metrics associated with each file type. We compute the metric SIZE for each file type since it is inherited from the root file type. Moreover, Java FileType also inherits the metrics of the Textual FileType.

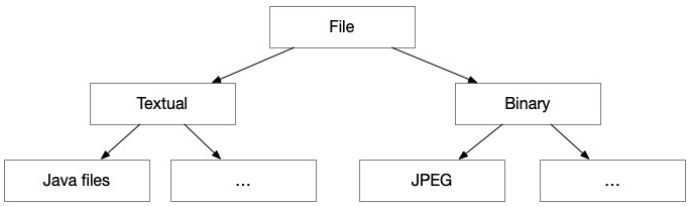


FIGURE 3.4: Example of a file type's taxonomy

We defined a set of metrics as can be seen in Table 3.1. We collect the file's size, the number of Lines of Code (LOC), the number of Source Lines of Code (SLOC) by ignoring comments and empty lines, and the

number of lines added and removed. These are basic source code metrics to demonstrate our approach. The list can be extended with additional metrics depending on the purpose.

3.2 Visualization

We represent a ProjectHistory with two kinds of visualization: a 2D visualization, which uses a matrix and works better with small systems, and a 3D visualization.

3.2.1 2D Representation

This visualization is based on the Evolution Matrix approach by Lanza [33], but adapted to our more flexible evolution model.

A ProjectHistory is a holder of ProjectVersions and FileHistories. A ProjectVersion represents a commit, and a FileHistory represents the history of a file. The connection between these two entities is a FileVersion that describes the state of a file in a system's version.
We can represent a ProjectHistory as a matrix with the following properties:

- Each column of the matrix represents a ProjectVersion, a commit of the repository.
- Each row of the matrix represents a FileHistory, the history of a file.
- Each cell of the matrix represents a FileVersion, the state of a file at a specific point in time defined by the commit.

An empty cell represents a FileHistory (i.e., row) that was not modified in a ProjectVersion (i.e., column). This concept was not present in the Evolution Matrix of Lanza because its model worked with SVN, and thus, it worked with incremental snapshots.

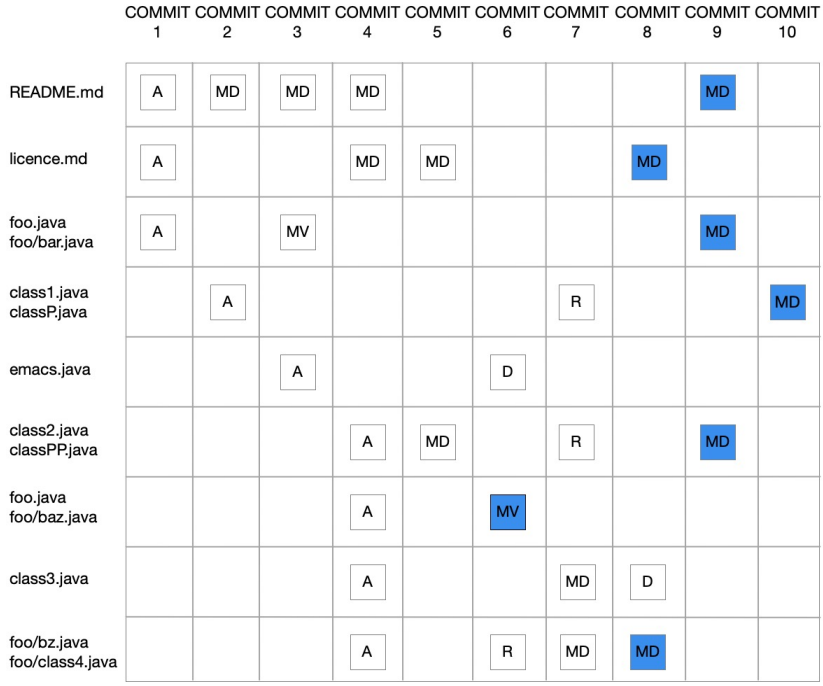


FIGURE 3.5: 2D Representation of the structural evolution of a repository

Figure 3.5 depicts the evolution of a repository with 14 files and 10 commits. In this figure we only show the structural evolution of a repository, without considering file metrics as Lanza did. Each row represents

the history of a file; therefore, it is associated with a set of FileVersions represented by the squares inside each cell. Actions on a file are labeled as: A for ADD, MD for MODIFY, MV for MOVE, R for RENAME, and D for DELETE. As we can see, the first action made on each file is an ADD. Some files end their history with a DELETE. Notice that `foo.java` and `foo/bar.java` are represented by the same FileHistory because they represent the same logical entity in the system. In commit C_3, `foo.java` was moved under `foo/` and renamed to `bar.java`. The same goes for `foo.java` and `foo/baz.java` represented by the seventh FileHistory.

We can read this matrix as follows:

- **Vertically (by rows)**, if we are interested in the history of a particular entity. For example, the FileHistory represented by the first row in Figure 3.5 represents the history of `README.md`, that was added in the first ProjectVersion (commit C_1) and then modified in the second, third, fourth, and ninth ProjectVersion. Figure 3.5 is an excellent example of why we cannot rely only on the file name to identify a system's entity. We notice that `foo.java`, represented by the third FileHistory, was added with the first ProjectVersion and then moved with the third into `foo/bar.java`. Then, in commit C_4, a new file `foo.java` was added. Nonetheless, the name of the files are the same, they represent two different entities.

- **Horizontally (by columns)**, if we are interested in which entities were updated on each ProjectVersion. For example, on the first ProjectVersion, we have added the `README.me` file, the `licence.md` file, and the `foo.java` file.

Figure 3.5 also provides an example of how we reconstruct the system's state after a commit. As we have seen, a ProjectVersion does not represent a system snapshot. Instead, it represents only the changes made to the previous version. To reconstruct the system's state at a specific version, we need to consider, for each FileHistory, the last change before that version. Under those circumstances, for each FileHistory, we must go back in time until we find the rightmost change. Of course, if the rightmost difference was a DELETE, we ignore the related FileHistory. In Figure 3.5, the state of the system after C_{10} is composed of the FileHistories that hold a row with a blue square.

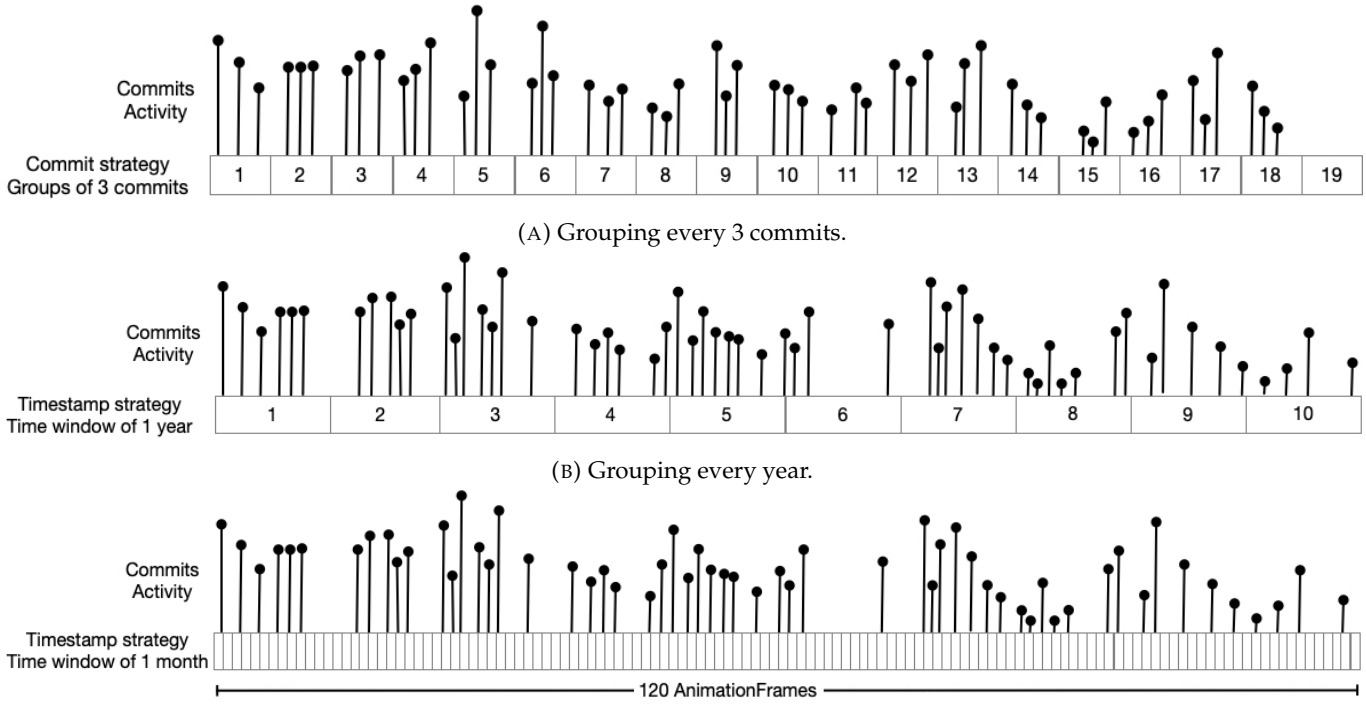
3.2.2 3D Representation

We aim to make an interactive 3D representation to ease the comprehension task of a developer. Interactive visualizations help users understand more information than a non-interactive version of the same graph. The interaction allows the user to discover a connection among the visualization elements. It is not easy to achieve the same result with a static visualization. For example, if we look at the graph of a repository activity on GitHub[2], we cannot understand which file was modified at a given time.

One advantage of our approach is the possibility of having multiple sources of information displayed simultaneously. We aim to make system analysis easier by using human senses and leveraging synesthesia. The phenomenon of synesthesia occurs when stimulation of a sense or a cognitive pathway leads to the involuntary stimulation of another reason or a cognitive path. We experience synesthesia when two or more events are perceived as the same. For example, synesthetic people might associate the red color with the letter D or the green color with the letter A. There are many forms of synesthesia, each representing different perceptions, such as visual forms, auditory, and tactile.

To visualize a project, we introduce the concept of **view**. We define a view as a way to illustrate the evolution of a project given a set of specifications. This set of specifications determines how the view must be built. For example, if we want to traverse the repository history by year, this information is part of the specification. A view holds a set of frames, called **AnimationFrame**, each representing the repository's

[2] https://docs.github.com/en/repositories/viewing-activity-and-data-for-your-repository/analyzing-changes-to-a-repositorys-content#visualizing-additions-and-deletion-to-content-in-a-repository



(A) Grouping every 3 commits.

(B) Grouping every year.

(C) Grouping every month.

FIGURE 3.6: Example of three different grouping strategies applied over a system whose history is 10 years long with 57 commits

state at a specific moment. Therefore, the entire history of the repository is displayed by rendering these AnimationFrames sequentially, like in a movie.

Each repository has a unique history. There are young repositories whose history is one year long and old repositories with more than ten years of development activity. Many repositories are inactive on GitHub, whereas others might have a vast number of contributors raising the total number of commits daily. Therefore, we cannot provide a static approach to traverse the history of a repository. Every visualization has its own goal and its way of traversing time. For this reason, we provide two visualization strategies to group commits into AnimationFrames. Both of them traverse the whole history from the beginning until the end.

- Grouping by commits: the user specifies the number of commits (`n`) and we create one AnimationFrame for every `n` commits. With this strategy, the concept of time is lost because we consider only the position of the commit in the commit tree.
- Grouping by timestamp: the user specifies a time window (`ts`) and we create one AnimationFrame for every `ts` seconds. All the commits inside the time window are part of the AnimationFrame. Therefore, we might have empty AnimationFrames if no commits are made in that time window.

Figure 3.6 shows an example of three strategies applied over the same history. In Figure 3.6a we made an AnimationFrame every 3 commits and as a result, the notion of time is lost. With a commit grouping strategy, we never have an empty AnimationFrame. In Figure 3.6b and Figure 3.6c the situation is different as they have empty AnimationFrames. Overall, the examples highlight the importance of a well-designed time window and underline our need to leave the user the choice of the time window's length. For example, younger repositories might have a daily time window, while older repositories might have a weekly, monthly, or yearly view.

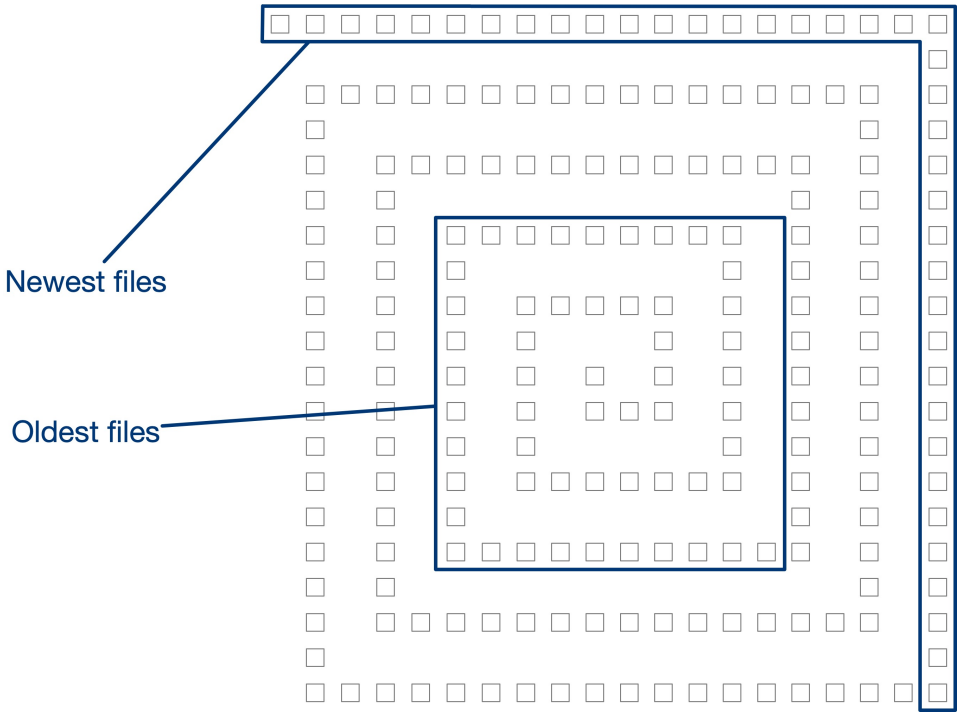


FIGURE 3.7: Outward spiral layout

To represent the system's state, an AnimationFrame holds a set of **ViewFigures**, each representing a file of the system.

A ViewFigure holds a set of properties used by the render process to draw a glyph representing a file.

Layout

A layout specifies how graphical entities should be laid out. Various layout strategies have been proposed to visualize software evolution. One of the most famous approaches is the city metaphor [59], presented by Wettel & Lanza, where a file's position depends on its package. It works very well with small and medium systems, and they stated that the interactivity and navigability could be substantially slowed down with large systems.

Our main goals are scalability and incrementality. The visualization needs to scale, even with a very large repository, and it must visualize the system's evolution incrementally. Therefore, the user can immediately distinguish older files from newer ones.

Under those circumstances, we adopted a spiral layout with an outward direction. It is shown in Figure 3.7. We set a constant gap between entities representing files. Older files are positioned at the center of this spiral, whereas newer files are always close to borders.

We added the *position* property to a ViewFigure to describe its location in the 3D environment.

Color

Synesthesia occurs when we experience an involuntary stimulation of a cognitive path during the stimulation of another sense. We present how we use color to describe file actions and simulate how time passes between actions. As a result, we decided to map each git action with color (see Figure 3.8). This mapping can be personalized through the *color* property of ViewFigure.

FIGURE 3.8: Mapped colors to git actions

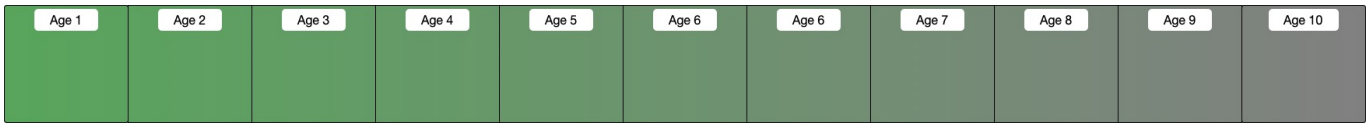


FIGURE 3.9: The aging process of an entity whose last action was an ADD and the maximum age is 10.

In addition to the color, each ViewFigure has another property: *age*. It represents the time elapsed between the last action made on a file and the currently displayed AnimationFrame. We selected two strategies to compute the age of an entity:

- Aging by commits: the user specifies the number of commits (`n`), and then after `n` commits, the age of a file is incremented.
- Aging by timestamp: the user specifies a time window (`ts`) and then after `ts` seconds the age of a file is incremented.

When an action is made on a file, its age is reset to zero.

The user also can set the maximum age. When reached, the color of the entity must be equal to the `base color` chosen by the user.

We initially set the base color equal to grey. Figure 3.9 provides an example of how the color of an entity changes toward the base color. In this figure, we used green as the base color and represented additions with ten aging steps.

All the files shared the same criteria to compute the age. The primary purpose of the aging is to immediately distinguish files modified recently from files modified in past AnimationFrames.

Shape and Opacity

To easily distinguish file types, we added a *shape* and an *opacity* property to a ViewFigure.

We did not set a pre-defined set of shapes because we preferred to let each implementation have its own set of models. This was done because it is not guaranteed that all the visualization libraries share the same model. The idea is to map each file type with a shape previously selected by the user.

Opacity was also added to allow the user to emphasize file types if needed.

Height

An important piece of information we want to represent is the value of a metric. SYN computes the values of a set of metrics, so we let the user choose his preference to map metrics to the files' height.

Figure 3.10 summarizes what we said so far. To visualize a Git repository, we use a view, a class that holds all the information needed to render it on screen. To depict the evolution, we traverse the repository's history by a group of commits (generated with a time window or every n commits). Each group is represented by an AnimationFrame, which holds a set of ViewFigures representing a file. The ViewFigure's position is used to describe the creation date of a file. The last action determines the color and age. A chosen metric is used to compute the height of the ViewFigure, and finally, the file type is mapped to shape and opacity.

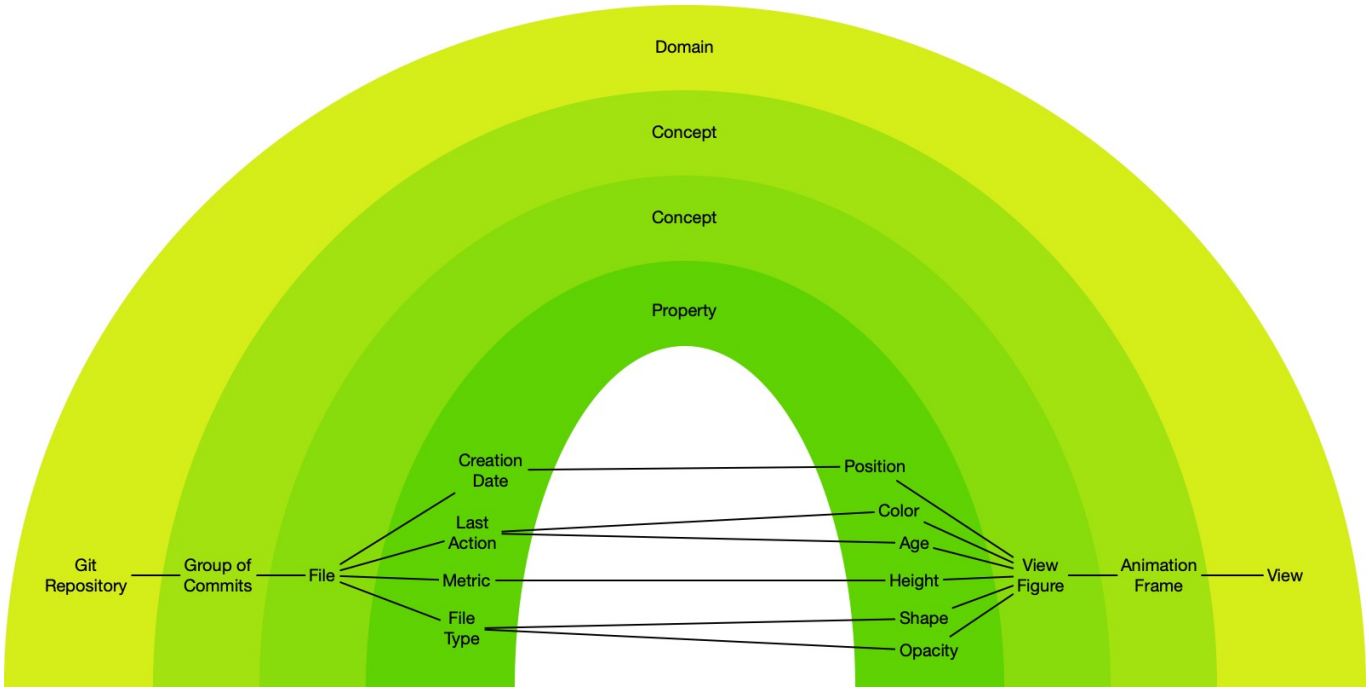


FIGURE 3.10: Mappings of file properties and metrics to view specifications

3.3 Evolution Auralization

In the previous sections, we detailed our approach to visualizing the evolution of a software system. In addition to sight, our approach also stimulates another sense: hearing. To complement our evolutionary visualization, we devised an approach to *"auralize"* the evolution of a software system. Auralization is a term introduced to be used in analogy with visualization to describe rendering audible (imaginary) sound fields [29]. The intuition is to create a melody, starting from the evolutionary data extracted from the version control system, to augment our visualization with complementary pieces of information.

Many parameters influence a musical composition, such as beats per minute (i.e., BPM), pitch, or note. In our approach, we consider the following parameters:

- **Tempo**: indicates the speed of a musical composition. It is measured in BPM.
- **Measure**: represents a single unit of time featuring a specific number of beats. In our approach, this value is constant (i.e., 1 second). For every measure, we play a new AnimationFrame.
- **Pitch**: is the quality that makes it possible to judge sounds as "higher" and "lower" in a sense associated with musical melodies.
- **Amplitude**: determines how loud a note is (i.e., volume).

According to Vickerts [55], to distinguish between different activities on the code, we need an *"orchestral model"* where each instrument represents a certain activity. Following this idea, our approach is composed of three instruments, each representing a different aspect of the evolution of a software system:

- **Bass drums** represents the **number of commits** in an AnimationFrame. We use this instrument also to dictate the **Tempo** of our musical composition: the higher the number of commits, the higher the tempo. We normalized the values in the interval $[60, 200]$. The **amplitude** of this instrument is constant.

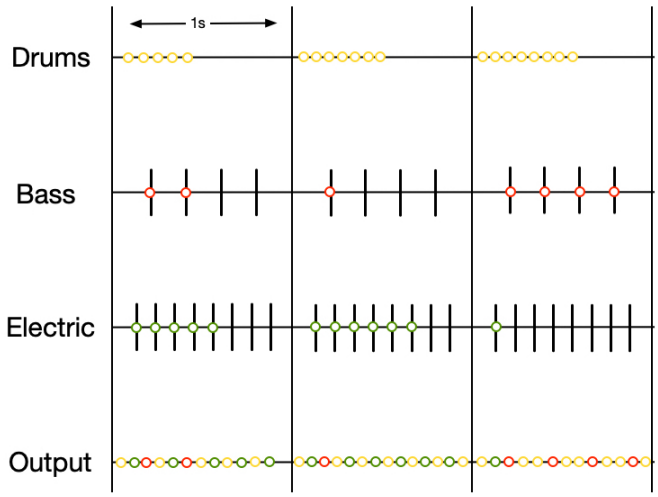


FIGURE 3.11: Example of how the auralization approach works.

- **Bass sound** represents the number of **deleted files** in an AnimationFrame. We use the number of deleted files to determine the repetition and the amplitude of the sound. We divided the measure into quarters and play this instrument one to four times depending on the value of the metric represented. We linearly normalized the amplitude in the range $[0.1, 0.8]$ depending on the metric's value.
- **Electric sound** represents the number of **added files** in an AnimationFrame. We use the number of deleted files to determine the repetition and the amplitude of the sound. We divided the measure into octaves and play this instrument one to eight times depending on the value of the metric represented. We linearly normalized the amplitude in the range $[0.2, 0.85]$ depending on the metric's value.

The output of our approach is a musical composition representing a system's evolution. It should be noted that we obtained the parameters mentioned above by tuning the musical composition to our preference However, the approach is customized with different metrics or different thresholds.

Figure 3.11 provides an example of how this approach works. Each column represents a measure with a constant value of 1 second. Each row represents one instrument, except for the last, which represents the output composition. The number of notes played from each instrument depends on the value of the respective metric. Therefore, in all the measures was recorded an intense activity, especially in the third one. The number of deleted files was lower in the first two measures than in the last. Consequently, the final output is composed of a few bass notes on the first two measures and four bass notes on the last measure, the most active of this example. The same rules were applied for the electric sound, representing the number of added files.

Chapter 4

Implementation

In this chapter, we present how we designed and developed SYN, a tool that implements our approach described in the previous chapter.

4.1 Platform Overview

SYN is our tool to visualize and auralize a system's evolution. It is composed of a set of modules, as follows:

- **Core**: it holds classes representing basic SYN concepts such as ProjectHistories, ProjectVersions, and FileVersions. It also provides abstract classes, open to any implementation, to achieve the extensibility goal.
- **CLI**: provides a Command Line Interface (CLI) to users.
- **Analyzer**: implements the analysis approach described in Section 3.1.
- **Server**: provides GraphQL endpoints to retrieve data.
- **Visual Inspector**: the user interface to debug and visually depict information collected during the analysis.

A user has two options to interact with SYN: through the console with the commands provided by the CLI module or through a web application embedded in the Visual Inspector module. Figure 4.1 gives a high-level overview of the architecture. Arrows represent the dependencies among the modules. The heart of the system is the Core module. It implements the core model and provides classes to standardize the communication among the modules. There is also a class called ViewGenerator, in the core to create views over the collected repository's data. The Analyzer module has classes that act as a repository explorer to retrieve data from the repository history. Moreover, metrics are extracted with the metrics extractor component and put inside each FileVersion through the history builder component. CLI provides commands that directly call functions defined inside Analyzer. This is why it has both Core and Analyzer as dependencies.

The Visual Inspector holds the Graphical User Interface (GUI) of SYN. Information is retrieved through endpoints specified by the GraphQL server of the Server module, which, in turn, retrieves view data from the Core module.

All modules of SYN are written in Java, except the Visual Inspector, which is written with React and TypeScript. Modules such as SYN Server or Analyzer implementation could be changed without altering the codebase of other modules.

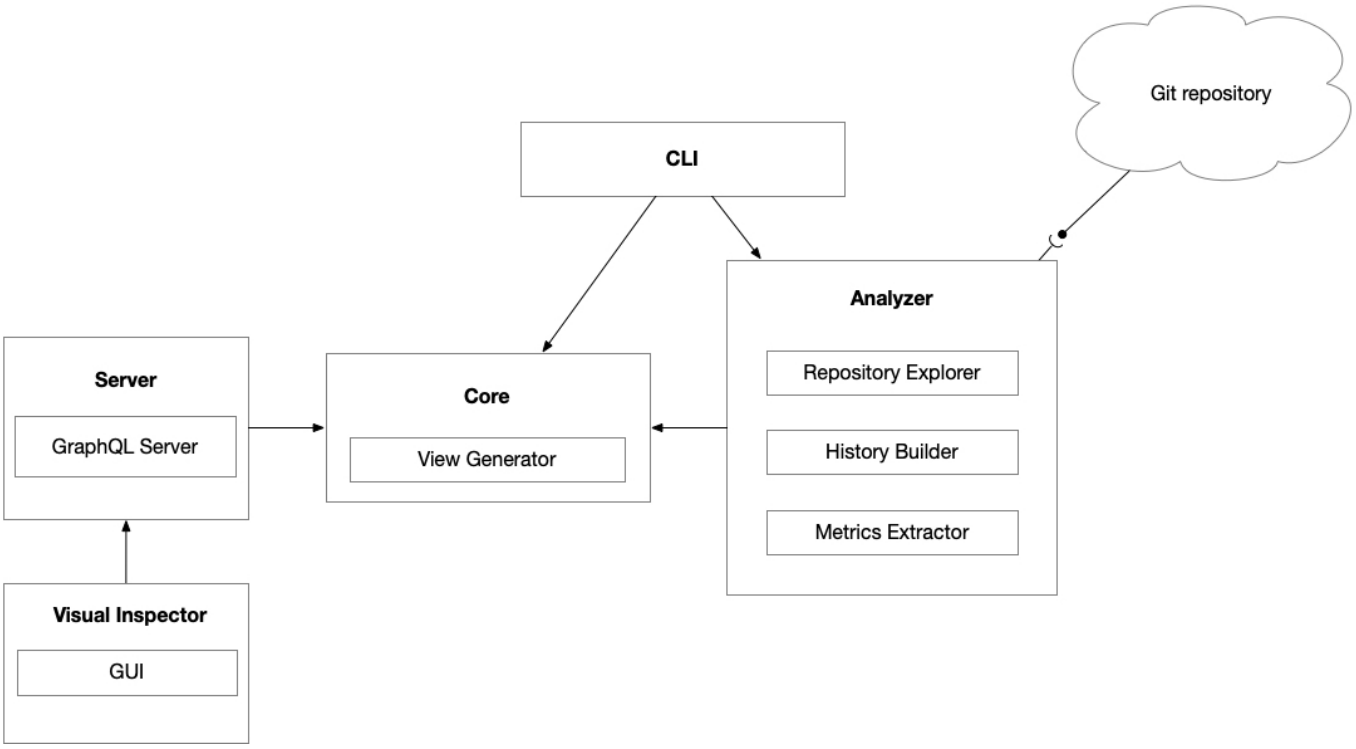


FIGURE 4.1: Architecture of SYN

4.2 Core

SYN Core is the module that holds all the entities introduced in Section 3.1. All the classes of our abstraction extend the `Entity` class, composed of the field `id`, to identify an object inside our domain. Classes inside the model of Core could be partitioned into four subdomains: Project, History, Analysis, and View.

4.2.1 Project

The first part of the model consists of the **Project** entities. The class diagram is shown in Figure 4.2. The abstract class `Project` represents a software system. It defines the following fields:

- `name`
- `projectHistory`: an object that represents the project's history. It holds the results of the analysis.
- `path`: the path of the analyzed git repository.

We defined two additional classes: `LocalProject` to represent a project in the local storage and `RemoteProject` to describe a project that needs to be retrieved from a remote source. `LocalProject` does not need any further fields to represent a local project, `RemoteProject` has a `projectURL` field. We might add more information, such as the git branch or the remote credentials; despite that, we decided to keep the implementation as simple as possible.

4.2.2 History

Each project might hold a **ProjectHistory** object representing its history. The `ProjectHistory` class holds a group of ProjectVersion and a group of FileHistories. The `FileHistory` class has the following fields:

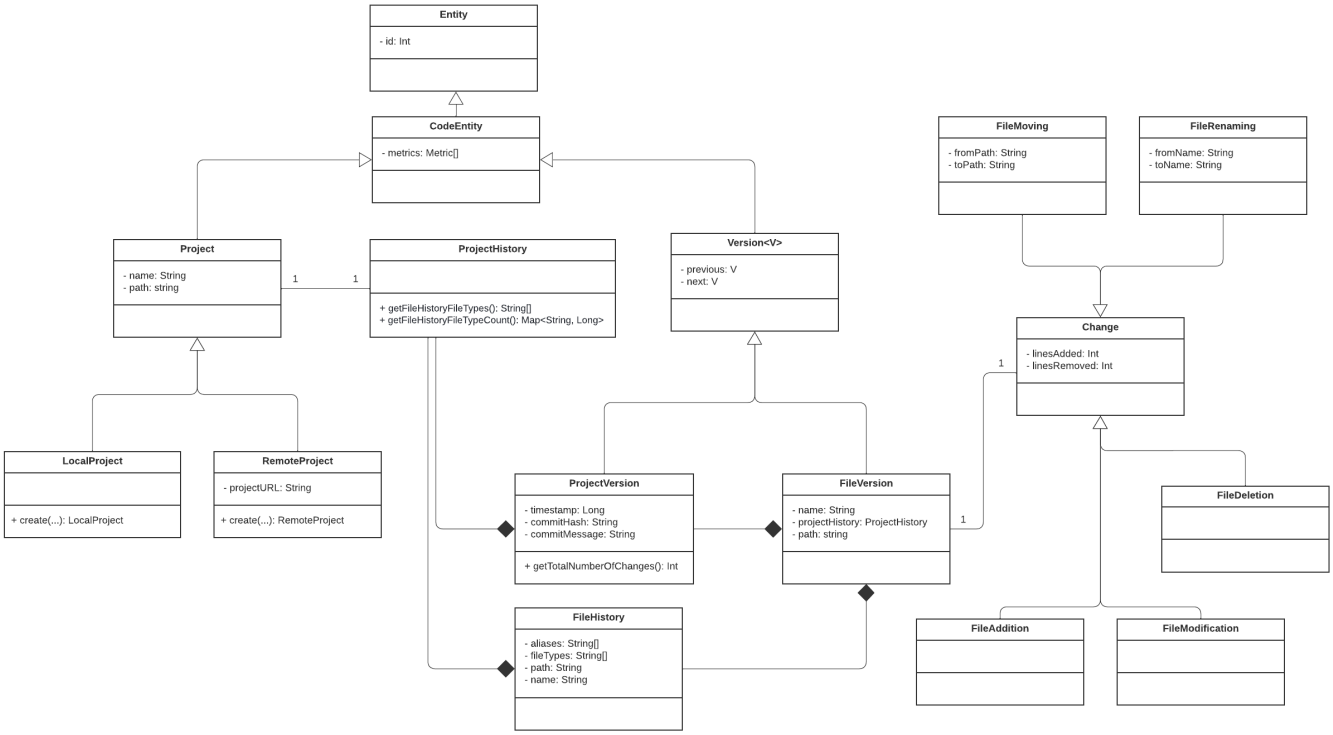


FIGURE 4.2: Class diagram of Project entities

- `aliases`: a list of paths related to the file. Usually, a single unique path identifies a file. However, since our approach tracks file moving and renaming, we store all the paths a file had throughout its evolution. In Section 4.4 we further explain the importance of this field.
- `fileTypes`: A set of Strings, each representing a type assigned to the file.
- `path`: the path of the file.
- `name`: the name of the file.
- `fileVersions`: a list of FileVersions related to this FileHistory.

With the `Version<V>` class, we represent the state of an entity at a particular point in time. The generic parameter `V` is a constraint to ensure the consistency of the class with the previous and the subsequent versions.

The `ProjectVersion` class is used to represent git commits; it defines the following fields:

- `fileVersions`: a list of FileVersions that are part of this commit.
- `timestamp`: expressed in seconds.
- `commitHash`: hash of the commit generated by git when the commit was made and used to identify a commit uniquely.
- `commitMessage`: the message written by the commit author when it was made.

Extending the Version class, the ProjectVersion class also inherits the previous and next fields that could be used to navigate through the history. Finally, the `FileVersion` class has the following fields:

- `parentProjectVersion`: the ProjectVersion holding this FileVersion instance. It retrieves the commit's related information, such as the timestamp or the message.
- `change`: represents the action on that file.
- `fileHistory`: represents the file being modified.

In our approach, we identified five different types of changes. They were implemented by extending the `Change` class, as depicted in Figure 4.2. The `Change` class declares two fields, `linesAdded` and `linesRemoved`, each one representing the number of lines added and, in turn, removed. As extensibility was one of our design goals, a new change type can be implemented by extending this class. For example, `FileMoving` and `FileRenaming` extend `Change` and specify two fields to track the path before and after a move or rename.

4.2.3 Analysis

The Core provides abstract classes to standardize the exchange of analysis results. The first class that the model defines is **AnalysisWorkDescriptor**. It is a holder of the information to instruct the analyzer on what it has to do. Namely, the project and the list of commits to be analyzed. The reason for this implementation choice is to allow multiple threads to run in parallel analyses without analyzing the same commit numerous times. We described this approach in Section 3.1

The **FileTypeManager** class is responsible for mapping a set of functions to file types. These functions are used to compute evolutionary metrics. We implemented the metrics presented in Section 3.1. However, any external module could easily extend this set of functions. This choice allowed external components to increase the set of metrics computed in SYN. Therefore, the analysis can be customized to collect relevant metrics for a specific project. Once the function to extract a metric given a File is written, the developer must associate it with the corresponding file type. For example, if in the future an engineer wants to write an Object-Oriented (OO) metric, such as the number of parents, they can create a function to compute it and associate the function to the OO.

Finally, the **ProjectAnalysisResult** class is used by an analyzer to return its results. This class makes a distinction between partial and total analysis results, allowing it to be used with the partial history retrieval approach defined in Figure 3.1.1. The fields declared in this class are:

- `project`: the projects of the analysis.
- `analysisCompleted`: whether the analysis is partial or total.
- `timestamp`: the timestamp when the analysis was done.
- `firstCommit`: the first commit considered in this analysis.
- `lastCommit`: the last commit considered in this analysis.
- `projectVersions`: a list of ProjectVersion found during the analysis.
- `fileHistories`: a list of FileHistories created during the analysis.
- `fileVersions`: a list of FileVersions created during the analysis.

This module also provides an abstraction of the core concepts of the Git protocol. It defines the `GitProject`, `GitCommit` and `GitChange` classes, each representing a repository, a commit, and a change on a file respectively. Thanks to this choice, an external module can provide a way to retrieve data from a git repository without sharing internal classes or dependencies with other modules.

4.2.4 View

In Subsection 3.2.2 we explained our visual approach. The **View** class defines how the visualization should be rendered. To display a repository's evolution, the user interface sequentially displays a list of AnimationFrames, each representing a state of the repository at a particular time. We designed the class **ViewAnimation** to define a frame of the view animation. This class has two properties:

- `representedEntities`: a list of ProjectVersions whose data had been used to create that frame.
- `viewFigureList`: a list of ViewFigure, each representing a FileHistory in the view.

The **ViewFigure** class holds graphical properties to display a FileHistory: `position`, `color`, `height`, `shape`, `age`, `opacity`, and `size`.

These properties are computed when a `View` object is instantiated. A view is created with the user's preferences. For example, if the user wants a specific color for a particular action, like dark green for additions, we store this information in the `ViewSpecification` class. A `ViewSpecification` is the crucial element of each view. All the views are generated from a `ViewSpecification` instance. The list of available properties defined in a `ViewSpecification` are:

- `versionGroupingStrategy`: the strategy to group commits and create AnimationFrames.
- `versionGroupingChunkSize`: the dimension of each group; it can be several commits or an amount of time.
- `colorPalette`: the mapping of the FileVersion's actions to colors.
- `agingGroupingStrategy`: the strategy group to define an age.
- `agingStepSize`: the dimension of each age; it can be the number of commits or the amount of time.
- `agingSteps`: the number of available aging steps.
- `mapperStrategy`: the strategy to compute the height of each entity once the metric's values are retrieved. For example, bucket strategy.
- `mapperStrategyOptions`: optional properties of the mapper if needed. For example, the maximum height.
- `mapperMetricName`: the metric's name that the mapper should consider before applying the selected MapperStrategy.
- `showUnmappedEntities`: whether the view should display entities without the selected metric.
- `fileTypeShape`: the mapping to associate a file type with a shape.
- `fileTypeOpacity`: the mapping to associate a file type with an opacity level.
- `figureSize`: the size of each figure representing a file.
- `figureSpacing`: the space between each figure representing a file.
- `showDeletedEntities`: whether the view should display deleted entities.
- `withGround`: whether the view should include a ground element.

We defined an abstract class **PositionLayout** whose implementation specifies how entities should be laid out and a class **MapperStrategy** whose implementation specifies how the height of the entity is computed. The set of mapping functions is extensible. To implement a strategy a developer needs to implement an interface that defines two methods: `void generateStrategy(final List<String> values)` and `double mapValue(final String value)`. The mapping process is composed of two phases:

- Generation phase: when the view is created, the method `generateStrategy` is called with all the values of the chosen metric.
- Retrieve phase: when ViewFigure's height is retrieved. It is set as the return value of the function `mapValue`.

In our initial implementation, we provided five different mapping strategies:

- LinearMapperStrategy: this strategy adopts a linear function to retrieve the height of a ViewFigure.
- NormalizerMapperStrategy: during the generation phases, all the values are normalized in an interval between zero and one.
- BucketCountStrategy: during the generation phases, all the values are partitioned into buckets. This function calculates the number of buckets for a given value. This strategy allows the customization of the number of buckets that are created. By default, this value is set to 100.
- BucketValueStrategy: relies on the BucketCountStrategy. During the generation, duplicate values are removed to distribute buckets equally.
- LinearBucketValueStrategy: relies on the BucketValueStrategy. During the generation phase, the maximum value of each bucket is recorded. Therefore, during the retrieved phase, it is used to make a liner proportion of the value and add them to the bucket number. For example, assuming that values of metrics are mapped to 100 buckets; the values $100, 200, 500$ are all mapped to the bucket 3. Therefore when we need to map the number 100, it is mapped to $3 + 100/500 = 3.2$; in the same way, the value of 200 is mapped to $3 + 200/500 = 3.4$ and the value of 500 is mapped to $3 + 500/500 = 4$.

4.3 CLI

CLI is a command-line interface to interact with SYN. It supports commands to control the analysis and utility commands to work with the analysis results of a project. Next, we describe the available commands in more detail.

Analysis commands

```
syn analyze auto -p <project_id> -o <output_file> -t <thread_count>
```

This command is used to run an analysis with SYN. Given the project id, the system automatically creates `thread_count` thread workers (5 by default), runs the analysis in parallel, and joins the results in the output file. The command ensures that each thread has a different git repository to work on.

```
syn analyze join -o <output_file> <...analysis_file>
```

This command joins analysis results into a single output file.

```
syn analyze manual -p <project_id> -g <git_repo_path>
     -rc <first_commit> -lc <last_commit> -o <output_file>
```

This command performs an analysis with SYN. It lets the developer choose the repository path and the first and last commits of the analysis. If the first commit is not specified, the analysis starts from the beginning of the history, and, in the same way, if the last commit is not specified, it completes with the end of the history.

```
syn analyze prepare -p <project_id>  -g <git_repo_path>
    -wn <workers_number> -of <output_folder>
```

This command creates a new folder with many worker descriptors serialized into JSON files. These worker files are used to instruct the Analyzer on how it analyzes a project. Based on the number of workers, the repository history is equally partitioned into chunks, each assigned to a worker.

```
syn analyze worker -p <project_id> -g <git_repo_path> -w <worker_file> -o <output_file>
```

This command runs the analysis on a project given its worker file. Furthermore, here the user can specify the path of the repository on the local storage. This was done because a repository can be analyzed by only one worker at a time.

Project commands

```
syn project list
```

This command prints a list of available projects to the console.

```
syn project create -n <project_name> -p <project_location>
```

This command is used to create a project given its name and its location. The location could be either a path or an URL.

```
syn project inspect -g <git_path> <project_id> <FileHistory_id>
```

This command inspects all the FileVersions of the indicated FileHistory. Furthermore, if the git repository is provided, SYN performs a double check to ensure that all the FileVersions were spotted. To do so, it exploits the output of the command `git log --full-history -- <FileHistory_path>`, keeping attention if the FileHistory's path was changed (git cannot do that).

```
syn util csv <project_id> -o <output_file>
```

This command produces a CSV containing all the commits' tracked information of the selected project.

4.4 Analyzer

The Analyzer module implements the analysis approach described in Section 3.1. To walk through the git-tree, it uses a Java git adapter called JGit.[1] We extended the Git classes provided by the Core module, and created the `JGitProject`, `JGitCommit` and `JGitChange` classes. Their job is to call the JGit API and retrieve historical information.

To run the analysis, we obtain first an `AnalysisWorkDescriptor` describing the job that the Analyzer has to do. The `JGitAnalysisWorkerDescriptorFactory` class partitions the commit tree and obtains a set of workers.

Figure 4.3 shows an example of a partition with three workers. First, the whole history of git is retrieved and stored in memory. Secondly, all the branches' commits are merged into a single list sorted by their timestamp. Then, the merge commits are removed since the changes recorded by them are duplicated and finally the resulting list is partitioned to match the requested number of workers.

The analysis process of a single worker contemplates the following steps:

1. We call the method `runAnalysis` of the class `ProjectAnalyzer` with a worker descriptor and a project as an argument.
2. The analyzer reads the git path of the repository and instantiates a new `GitProject`.
3. A list of commits, specified in the worker, is retrieved from the `GitProject`.

[1] https://www.eclipse.org/jgit/

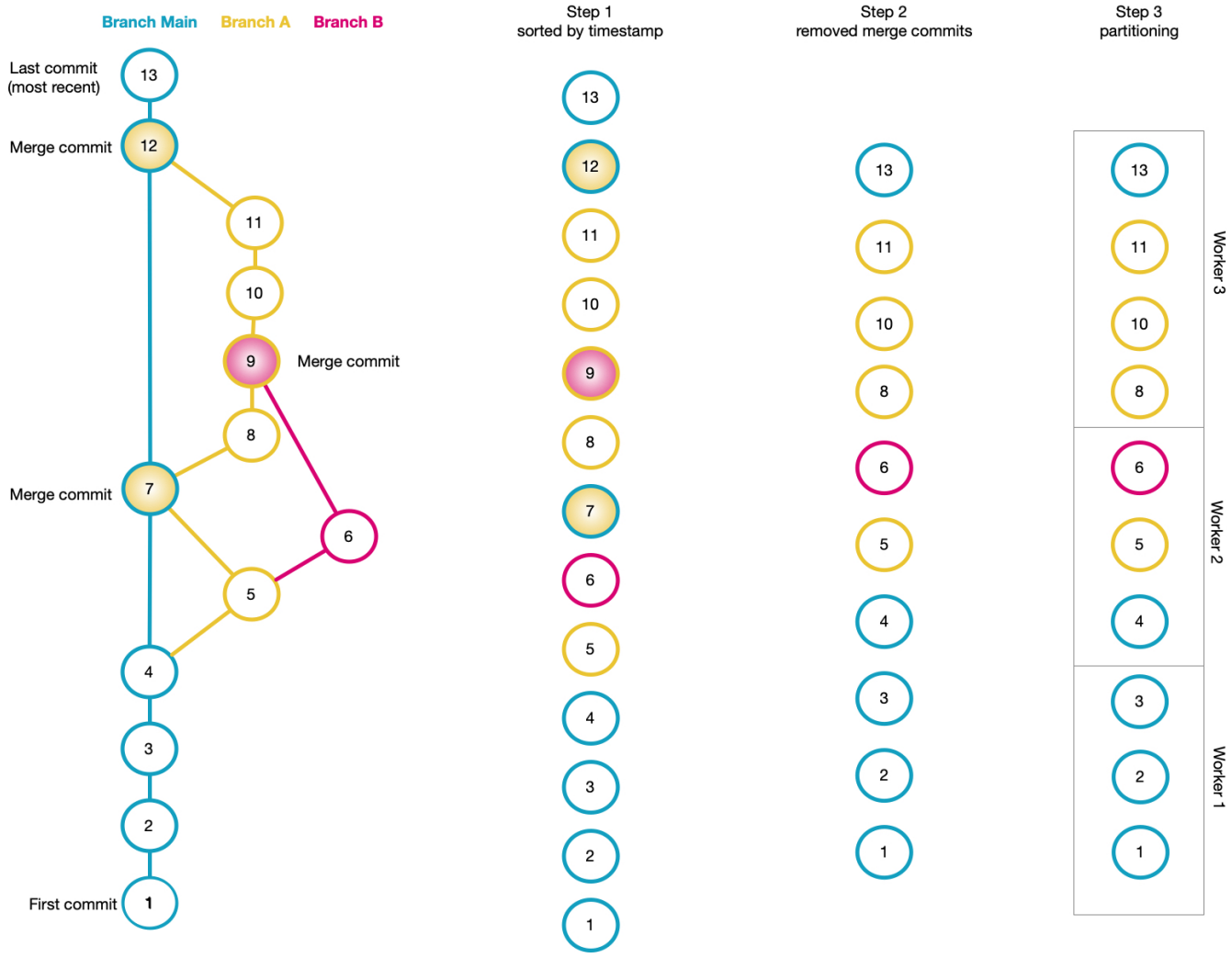


FIGURE 4.3: Example partitioning of a commit tree with three workers.

4. The first commit of the list is retrieved, and a new `ProjectVersion` is created with its details.
5. The analyzer runs the checkout command of git to restore the version of the system at the one specified by the commit.
6. The analyzer retrieves a list of modified files.
7. For each modified file, the analyzer creates a new `FileVersion`, links it with the corresponding `ProjectVersion` and `FileHistory` (creates it if it does not exist yet), and finally extracts all the evolutionary metrics expected for that file type.
8. Once all the modified files are analyzed, the analyzer repeats steps 5, 6, and 7, considering the subsequent commit of the list each time.
9. Once all the commits are analyzed, the analyzer returns the results as a `ProjectAnalysisResult` object.

The analyzer can parallelize with multiple workers the analysis of large repositories. Each worker produces a `ProjectAnalysisResults` that represents a partial history. To obtain the entire history of the repository, we join all partial analysis results. In SYN, each entity is identified by an id. The order in which entities are created is important because we visually sort entities based on their creation timestamp. Therefore, consistency between ids of FileHistories is necessary for the join algorithm. We said that each analysis result has a list of `FileVersions`, a list of `ProjectVersions` and a list of `FileHistories`. Analyzers are not meant to communicate with each other. Therefore they work in isolation, like in a sandbox. When we join the results, the concatenation of the `FileVersions` or the `ProjectVersions` is not a problem when the historical sequence is respected. These objects are mutually independent of each other. The challenge comes with the `FileHistories`. When the analyzer spots a file, if it was not discovered yet, it creates a new `FileHistory` with a unique id. This sounds like a problem with merging because we might consider the same file twice. In other words, we have an issue linking FileHistories between analysis results. Assuming for a moment that we do not have this obstacle, another situation might happen, more complicated than the previous one.

If a file changes its path in the middle of two analysis results, we have an entity with the old path in the first analysis and an entity with a new path in the second analysis. Unless we would not keep track of all the possible paths of an entity, it is impossible to reconstruct a connection between these two files. This is the reason that brought us to add the `aliases` field in the `FileHistory` class. In the join algorithm we developed, we employ a map to keep track of FileHistories. As a result, the file has a different id in any analysis result, guaranteeing consistency among FileHistories. Furthermore, we can ensure the absence of duplicates in the newly built repository history.

We present an algorithm (1) that takes as input a list of already discovered FileHistories and a `ProjectAnalysisResult`, and it returns a dictionary. This dictionary is used to map partialFileHistories, the FileHistory of an analysis result, to definitive FileHistories, which are part of the entire history of the repository. To obtain this map, it uses the following strategy:

1. If the last alias of a definitiveFileHistories is equal to the first alias of a partialFileHistory, then they represent the same file.
2. If in the partialAnalysisResult there is more than one FileHistory with the same first alias, only the first is mapped to a definitiveFileHistories, and the other partialAnalysisResult are mapped to a new FileHistory.
3. If a partialFileHistory has not matched an alias with a previously created definitiveFileHistories, then it represents a new definitiveFileHistories.

Algorithm 1 Algorithm to create a mapping between partialFileHistories and definitiveFileHistories

```
1:  procedure PARTIALTODEFINITIVEFH(projectAnalysisResult, partialFileHistories)
2:      partialToDefinitive ← Map()
3:      partialAliasToFH ← Map()                                          ▷ Strategy 1
4:      unmappedPartialFileHistories ← List()                             ▷ Strategy 2
5:      for all partialFH in partialFileHistories do
6:          firstAlias ← partialFH.aliases[0]
7:          if !partialAliasToFH.has(firstAlias) then
8:              partialAliasToFH.set(firstAlias, partialFH)               ▷ Strategy 1
9:          else
10:             unmappedPartialFH.append(partialAliasToFH)                ▷ Strategy 2
11:         end if
12:     end for
13:     for all definitiveFH in projectAnalysisResult do                  ▷ Strategy 1
14:         lastAlias ← definitiveFH.aliases[definitiveFH.length − 1]
15:         if partialAliasToFH.has(lastAlias) then
16:             partialFH = partialAliasToFH.get(lastAlias)
17:             definitiveFH.path = partialFH.path
18:             definitiveFH.aliases.addAll(partialFH.aliases)
19:             partialToDefinitive.set(partialFH, definitiveFH)
20:             partialAliasToFH.remove(lastAlias)
21:         end if
22:     end for
23:     unmappedPartialFH.addAll(partialAliasToFH)                        ▷ Strategy 3
24:     for all partialFH in unmappedPartialFH do
25:         definitiveFH = FileHistory(partial.name, partial.path)
26:         definitiveFH.aliases = partialFH.aliases
27:         partialToDefinitive.set(partialFH, definitiveFH)
28:     end for
29:     return partialToDefinitive
30: end procedure
```

4.5 Server

The Server module is responsible for providing the analysis results in an intermediate language between the front-end (Visual Inspector). We developed the server with Spring Boot, a popular Java application framework.[2]

GraphQL API

The server module provides a set of GraphQL[3] endpoints to retrieve information from SYN. GraphQL is a data query and manipulation language for APIs. It also provides the runtime engine for fulfilling the queries. This is an advantage compared to a REST API because, instead of always returning a predefined set of data, GraphQL only returns the needed information, making the communication more effective. In GraphQL, endpoints are divided into queries and mutations. Queries retrieve data, while mutations create or alter data. In the rest of this section, we present the mutations and queries defined in our GraphQL API.

Mutations

`createProject(projectName: String!, projectLocation: String!): Project`
Creates and initiates the analysis of a new project. It takes as parameters the project name and its path on the local machine or the URL as a String.

Queries

`projectList: [PartialProjectInformation]!`
Used to retrieve a list of projects. For performance reasons, only the name and the id of a project can be retrieved.

`view(projectId: Int!, viewSpecification: ViewSpecification!): View`
Returns a view of the project identified by `projectId`, built following the directives specified in the `viewSpecification` object. This query returns an object representing a view; thus, it has all the fields described in Subsection 4.2.4.

`partialView(projectId: Int!, viewSpec: ViewSpecification!, viewAnimationId: Int): View`
Returns a view with the next 100 AnimationFrames starting from `viewAnimationId`. This endpoint was created to improve the clients' performance. If the viewSpecification expected too many animations, the resulting view would be a bottleneck for the client. With this endpoint, it can still retrieve the view, and animation can be lazily loaded when required.

`fileHistory(projectId: Int!, fileHistoryId: Int!): FileHistory`
Retrieves the FileHistory identified with `fileHistoryId` of the project with `projectId`.

`projectVersions(projectId: Int!, projectVersionsId: [Int]!): [ProjectVersion]!`
Retrieves the ProjectVersion identified with `projectVersionsId` of the project with `projectId`.

`groupingPreview(projectId: Int!, viewSpecification: ViewSpecification!): Int`
Computes the number of AnimationFrames created with the given `viewSpecification`.

`fileTypeCounter(projectId: Int!): [FileTypeCounter!]!`
Returns a list for each file type with the number of occurrences in the project `projectId`.

`fileTypeMetrics(projectId: Int!, fileTypeFilter: [String]): [FileTypeMetrics!]!`
Returns a list for each file type with its metrics. Only the file types in the `fileTypeFilter` list are considered.

[2] https://spring.io/projects/spring-boot
[3] https://graphql.org/

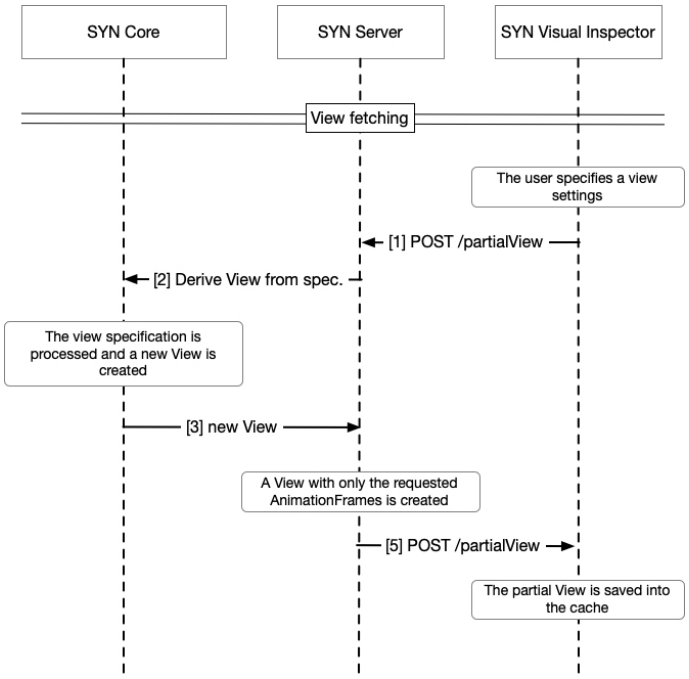


FIGURE 4.4: Example of the process of retrieving a view through GraphQL

Figure 4.4 explains how a View is retrieved through a GraphQL endpoint with a sequence diagram. Once the client, in this case, Visual Inspector, specifies a `viewSpecification`, it makes a POST request to the `partialView` endpoint with the `projectId`, the `viewSpecification` and the `viewAnimationId` of the AnimationFrame. The Server module, in turn, makes the same request to the Core module. A new View is created and retrieved from Server. It is modified to include only the AnimationFrames requested from the client. When the partialView is computed, it is returned to the client.

4.6 Visual Inspector

The Visual Inspector module is a web application to interact with SYN, written with React.js,[4] a popular JavaScript framework. The visualization is based on Babylon.js,[5] an open-source 3D library. It provides different customizations for the visualization, such as the entities' shape and color. These customizations are sent to the back-end server through a *view specification* file as shown in Figure 4.4.

The primary purpose of this application is to debug the view and explore all the possible visualization combinations of a system. The main page, shown in Figure 4.5, holds a list of projects analyzed with SYN.

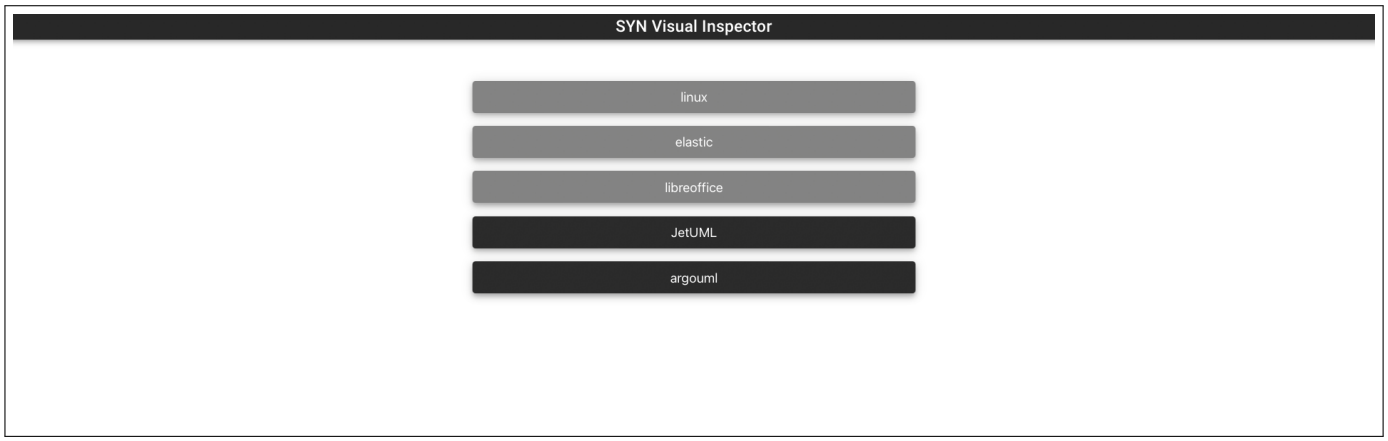


FIGURE 4.5: SYN Visual Inspector main page

4.6.1 Project setup

When a project is selected, the UI searches for a viewSpecification object in the web browser's local storage. If it is not present, the module shows the project setup. At the end of the setup, all the preferences selected are saved. The project setup is a wizard to configure a project's visualization, composed of five main steps as follows:

- Component selection: where the user can select which file types must be considered in the visualization. To help the user, SYN presents the project's file types distribution.
- Grouping version strategy: the user can choose how commits are grouped to create AnimatioFrames.
- Figure settings: to customize the shape, metrics, and opacity of each file type. The user can also choose which strategy must be used to compute the entity's height.
- View color: to customize the color of each git action and how aging must act.
- View settings: to customize general settings such as the speed of the visualization, shadows, or whether deleted entities should be visualized.

[4] https://reactjs.org
[5] https://www.babylonjs.com

Component selection

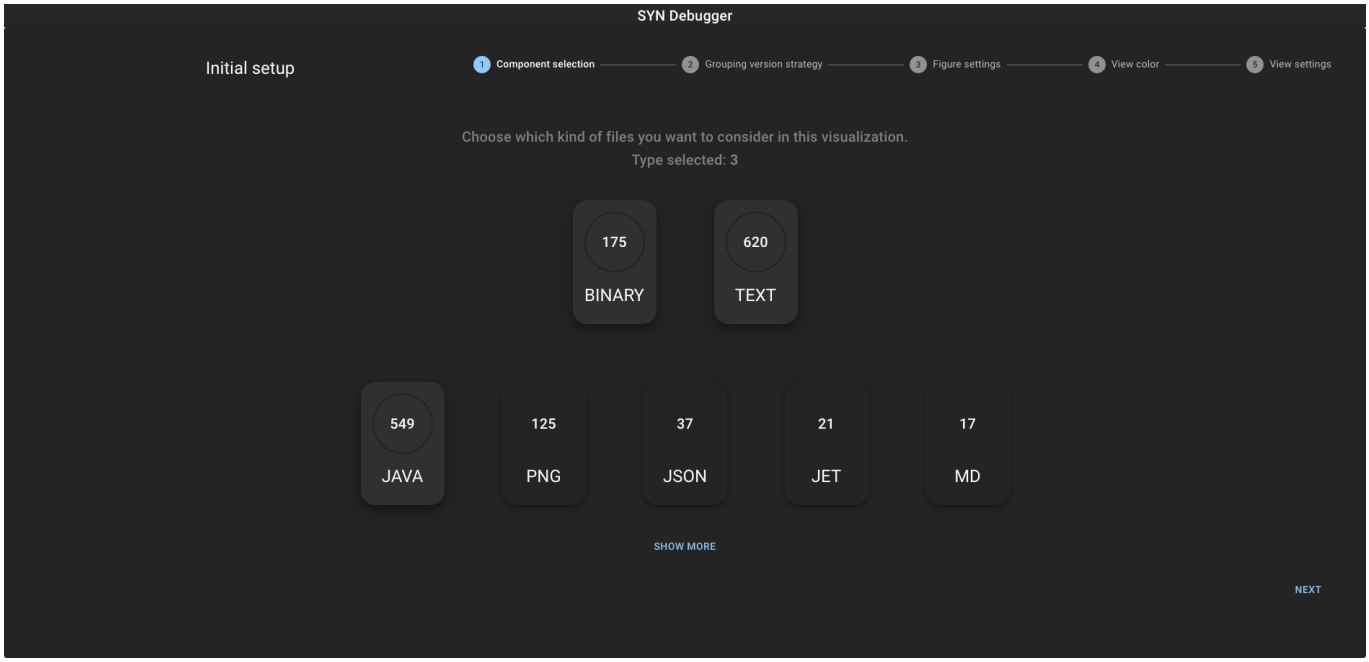


FIGURE 4.6: Project setup: component selection

Figure 4.6 presents the first step of the project setup. It shows a list of cards, each representing a file type. We can break the visualization area into two parts: on the top, there are two cards to represent binary files and text files. The sum of these two represents the total number of FileHistories in the system because each file must be either textual or binary. The bottom part shows the cards for the file types identified in the project. For example, if a file is named "foo.java" the JAVA card in this area represents it. The cards are sorted in descending order by their number of occurrences. This view considers all the file types present in the system. To show the complete list, users have to click on the "show more" button.

Grouping strategy

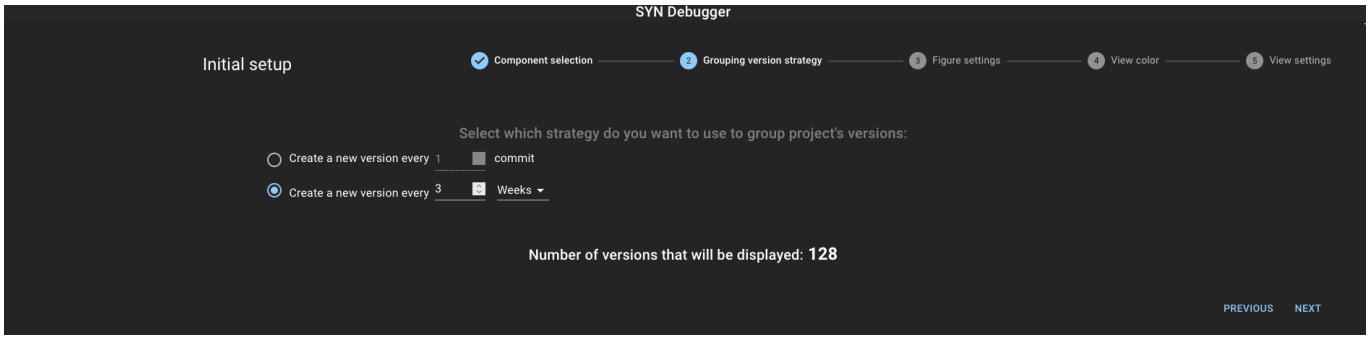


FIGURE 4.7: Project setup: grouping strategy

Figure 4.7 shows the second configuration step. It allows the user to choose how SYN groups ProjectVersions. Following what we have presented in Subsection 3.2.2, we provide two strategies to group ProjectVersions:

- by commits: we create an AnimationFrame every n commits.

- by timestamp: we create an AnimationFrame every n seconds.

To support the selection of the timestamp strategy, instead of manually computing the width of the time window, the user must only specify an amount and its time unit (i.e., hour, day, week, month, year). After the selection, the UI shows a preview of the grouping strategy to give an idea of how many animations are created with the selected settings.

Shape settings

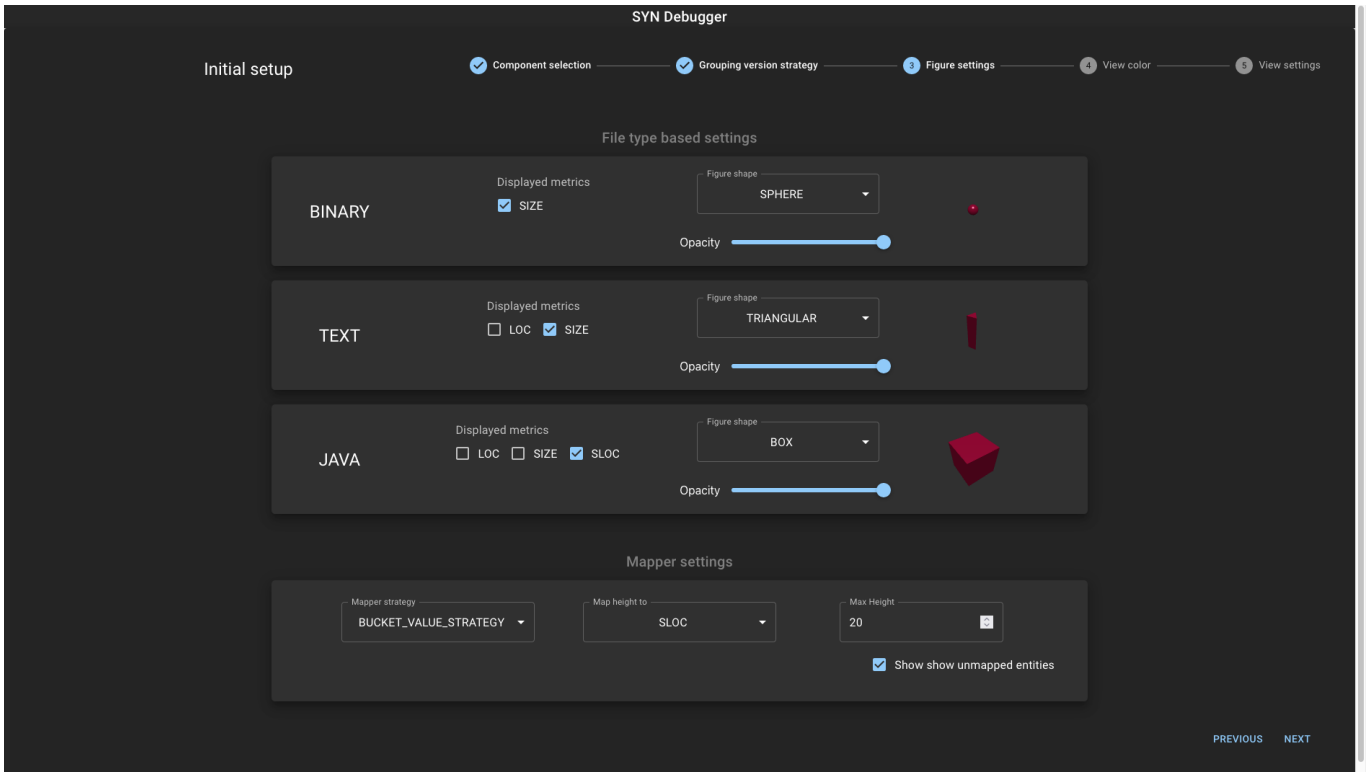


FIGURE 4.8: Project setup: shape settings

The third step of the setup is shown in Figure 4.8. It allows the user to express graphical preferences. The view creates a card for each file type selected in the "Component selection" step. The user can customize each card with the followings:

- Metrics: a list of metrics that should be visualized when a file is selected.
- Shape: in our visualization, we implemented five shapes: box, triangular, cone, sphere, and cylinder.
- Opacity: to control the transparency of each file type.

The user can also configure which metrics and mapper strategies determine the height of entities and the maximum height of the entities. The list of available metrics for the mapper comprises the metrics selected for each file type. The UI has a checkbox to specify whether FileHistories that do not have the selected metric should be rendered or not.

With all the settings available on this page, the user has control over how FileHistories are rendered. There are many possible combinations. For example, if we want a visualization that distinguishes between Java, text, and binary files, we can use the settings shown in Figure 4.8. It specifies a different shape for every file type, keeping the maximum opacity level for all of them. Furthermore, the height is computed

with a bucketValueStrategy that works with the SLOC metric. As a consequence, only Java files have a height in this visualization. File types that don't have the mapper metric are called "unmapped entities", and their height will be fixed.

View color

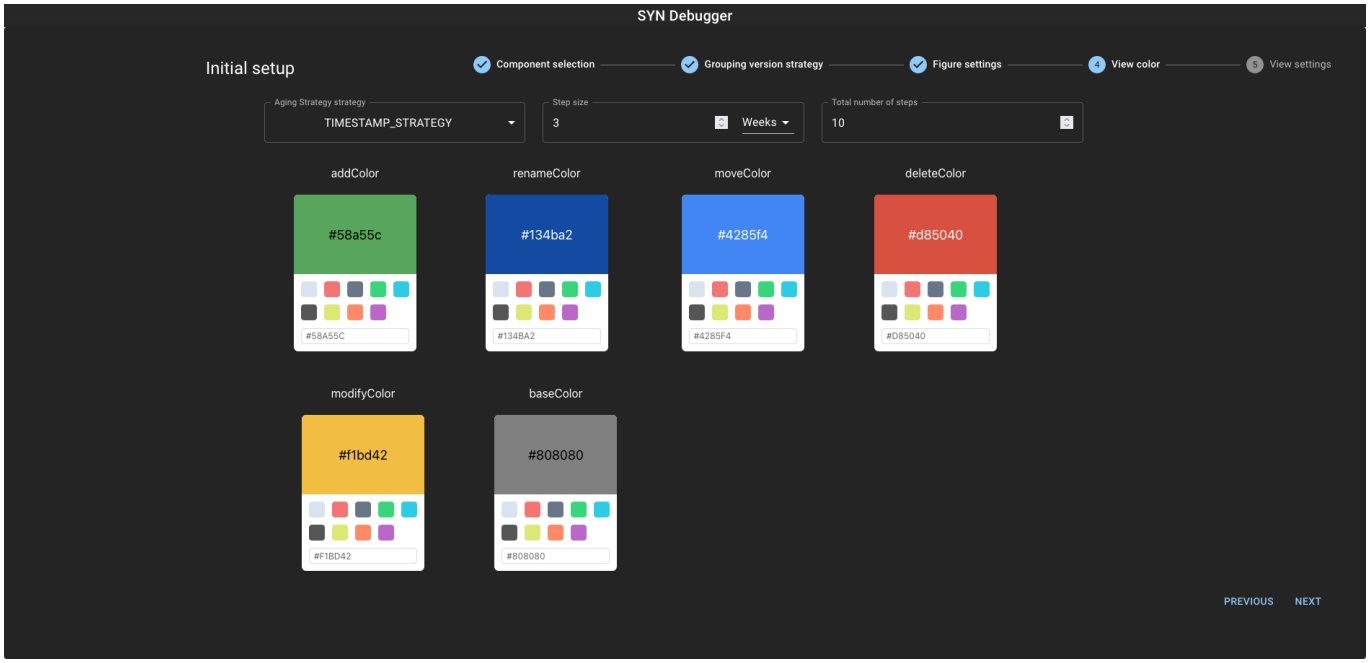


FIGURE 4.9: Project setup: view color

Figure 4.9 shows the color setup step. It configures the coloring preferences as follows: The user controls how aging is computed. As with grouping versions, aging can be done following two possible strategies: one by timestamp and one by commits. The color transition is linear. Therefore, the user can choose the number of steps between the original color of the entity and the base color.

Finally, each action has a distinct original color, customizable for the user. The color palette used by default is shown in Figure 4.9.

View settings

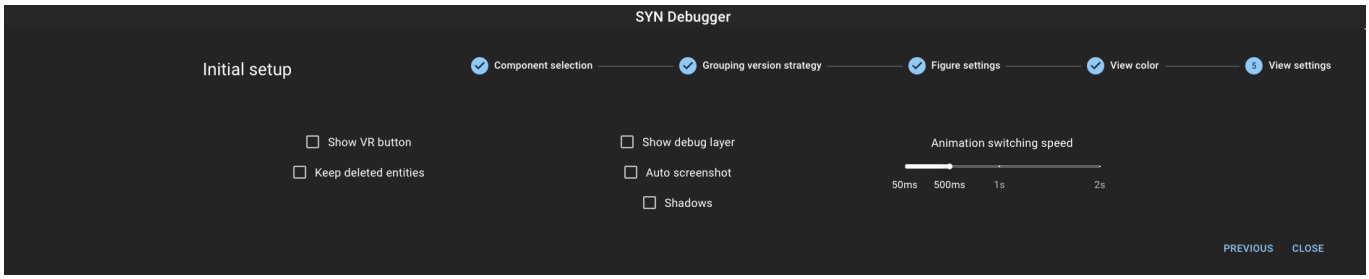


FIGURE 4.10: Project setup: view settings

The last step of the setup, shown in Figure 4.10, allows the user to express general preferences about the behavior of the UI.

- Show VR Button: enables the full immersion experience that must be enjoyed with a VR headset.
- Keep deleted entities: keeps deleted entities in the visualization. By default, these are not displayed.
- Show debug layer: makes the 3D engine's debugger visible.
- Auto screenshot: a screenshot of the visualization is generated with text indicating the time of the last commit included in the displayed AnimationFrame.
- Shadows: enables the shading of entities.
- Animation switching speed: the amount of time between rendering one animation and another when the autoplay mode is activated.

4.6.2 Project visualization

After having specified all the visualization preferences, they are stored in the web browser's local storage. The UI can retrieve a view from the backend server, given the project id and the view specification.

Initially, it calls the `partialView` query provided by the Server module. Once it gets the response, the visualization of the first AnimationFrame is automatically loaded. A pre-fetcher mechanism was also implemented. It works as follows: once half of the retrieved AnimationFrames have been displayed, it lazily requests a new partialView containing all the following AnimationFrames of the currently displayed view. This way, the jump between a partialView and another is not subjected to network timing issues because the partial view was already prefetched.

The initial display of the view is represented in Figure 4.11. The visualization settings of this view are:

- Version grouping strategy: timestamp, three weeks
- Color palette: default
- Aging: timestamp, three weeks with ten steps
- Height mapper: BucketValueStrategy on SLOC
- Deleted entities are not shown
- All the entities have the same shape and opacity
- All the available metrics are selected for each file type.

The main visualization area can be broken down into four parts: In box A, a 3D environment displays FileHistories on a virtual plane. The view angle and the zoom level of the camera can be controlled with the mouse. In box B, we have a card with general information about the project visualization. This card includes the project's name, the animation number, the dates of the animation frame, and its commit list. The slider shows the overall progress of the display, two buttons to jump to the subsequent or previous animation, and finally, one button to jump to the following animation with the time interval previously set. All the preferences specified during the project setup can be changed by clicking on the three dots in the top right corner. In box C, we have a card to inform the user of the number of entities the UI renders. And finally, box D appears when an entity is selected with the mouse. It shows more details of the entity as follows:

- the name and the current path of the entity. The current path is the entity's path in the last commit of the displayed animation frame.
- a table filled with all the metrics retrieved on the last commit of the displayed animation frame. The list of metrics is filtered based on the project setup.

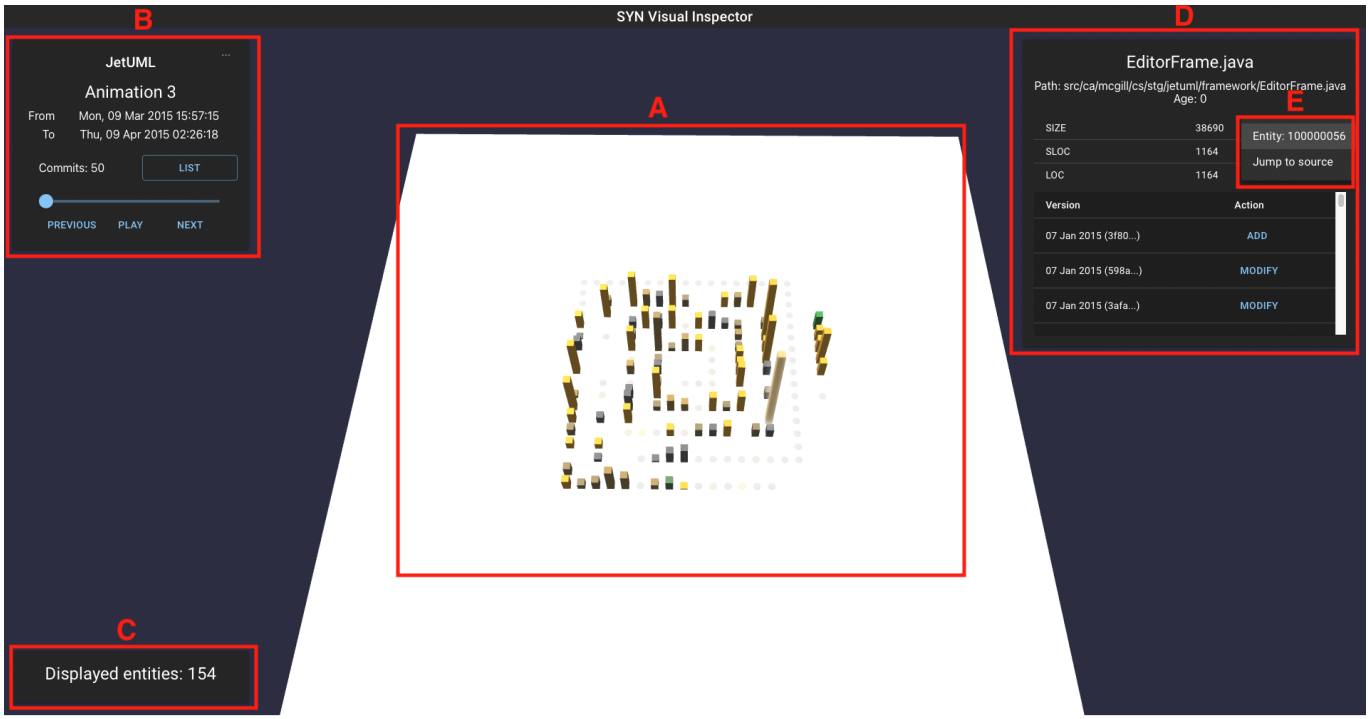


FIGURE 4.11: Visualization of JetUML with the default settings.

- a table filled with all the FileVersions associated with the selected FileHistory. It shows the commit hash and the action made on that commit for each FileVersion. If the project is hosted on GitHub, when the user clicks on the commit hash, the Visual Inspector opens a new tab with the code of the class at that revision. A tooltip with commit information appears if you hover on an action with the mouse.

- on right-click, a contextual menu gets displayed (box E). With this contextual menu, one can jump on GitHub to see the raw file in the last commit of the currently displayed animation.

Figure 4.12 provides another visualization example where deleted entities are shown in the visualization, and the shadows are displayed.

The Visual Inspector is a helpful tool for understanding and debugging SYN's analysis process. However, it has performance issues in rendering large systems. The main problem stems browser environment as it has limited resources. Even though Javascript has APIs to support a high-performance rendering of 3D graphics,[6] we experienced a significant frame rate (frames per second, FPS) drop when rendering large systems. To overcome this limitation and prove the versatility of our approach, we rendered large systems with POV-Ray,[7] an open-source tool for creating high-quality three-dimensional graphics. It works with `pov` files containing instructions to specify how the image should be rendered. We developed an extension of SYN that, given a View, produces a `pov` file for each AnimationFrame, following the same approach adopted in the Visual Inspector. Figure 4.13 provides an example of a big system rendered with POV-Ray.

[6] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API
[7] http://www.povray.org

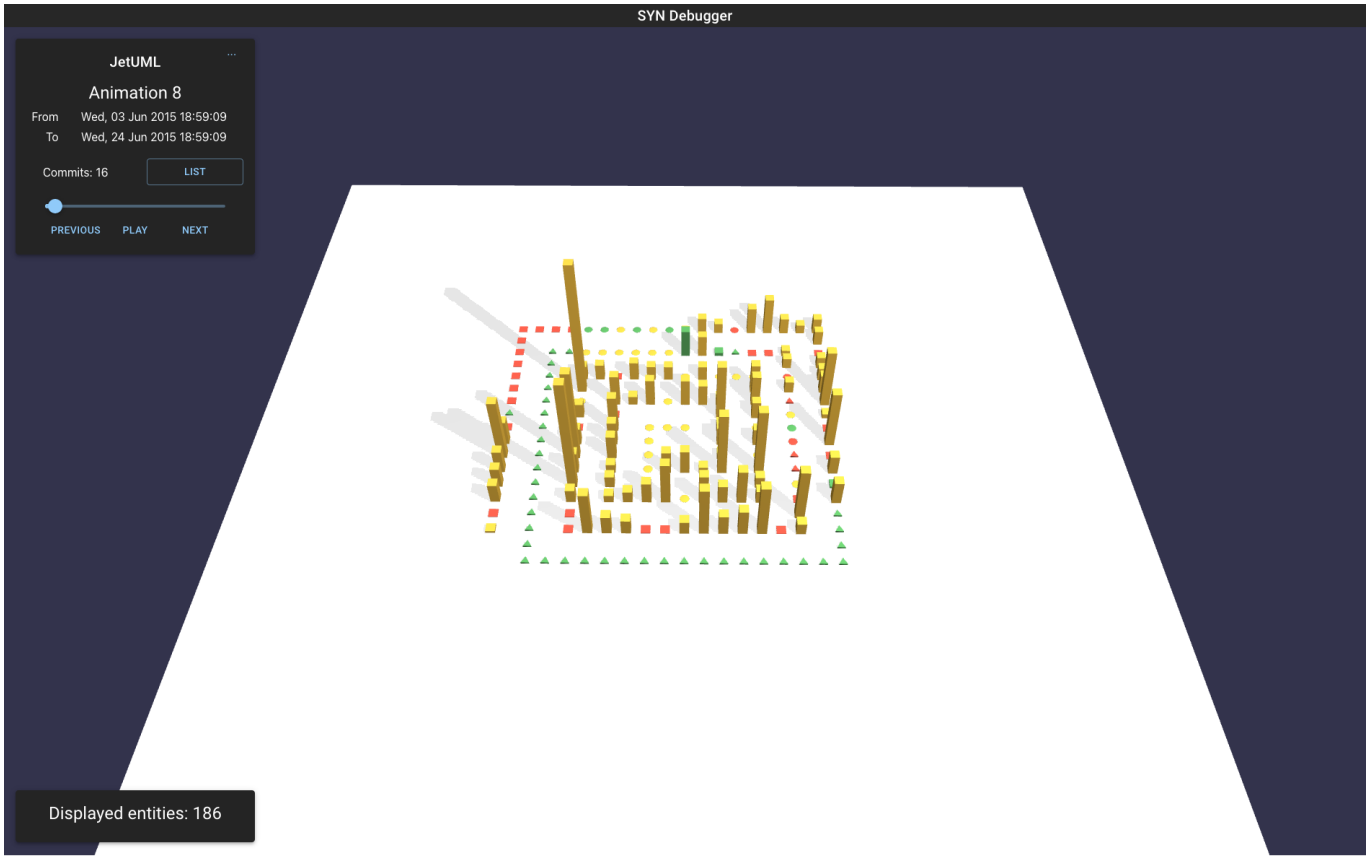


FIGURE 4.12: Visualization of JetUML with shadows, deleted entities and custom shapes for non-java files.

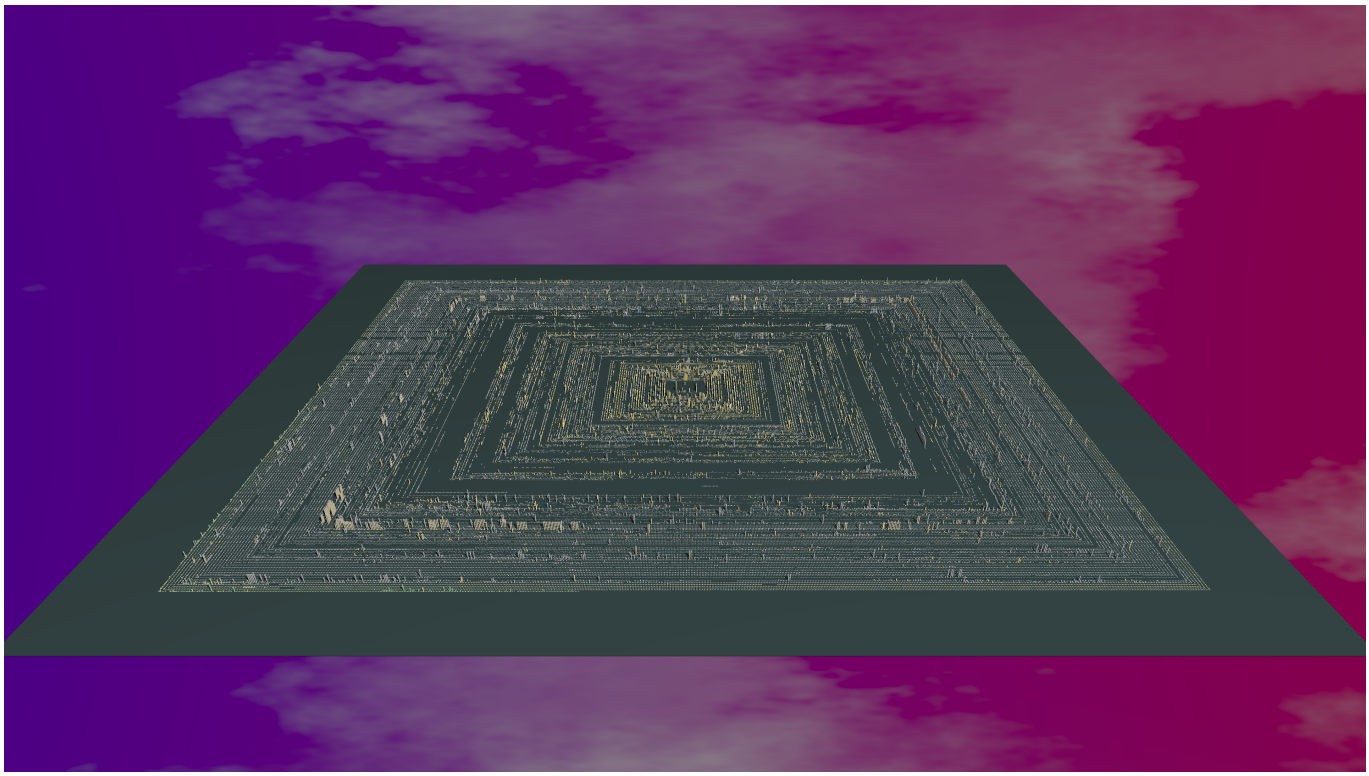


FIGURE 4.13: Example of a system rendered with POV-Ray (LibreOffice).

FIGURE 4.14: Interface of SonicPi

4.7 Audio

SYN implements the auralization approach detailed in Section 3.3 in its Core component. Its implementation uses Sonic Pi[8] to compose melodies. Sonic Pi (Figure 4.14) is a code-based creation tool written in Ruby. It enables you to write and modify code live to make music. To do that, it defines a domain-specific language to compose melodies procedurally. For example, the instruction `play 60` plays the 60th note on the piano (selected by default).

Appendix A shows the code we developed to implement this mapping. To debug our auralization approach, we developed a Python application that queries the server and controls SonicPi. The application works as follows: first, it queries the GraphQL server to retrieve the view of a project. Once the view is returned, it extracts from each AnimationFrame the needed metric values. Then these values are sent to SonicPi through a protocol called Open Sound Protocol (OSC). Finally, it renders one graph for each metric to let us associate each metric value with a sound. To do that, we used a black dot representing the current AnimationFrame.

Figure 4.15 shows the interface of the debugger. Each chart represents a metric, and the black dot (synchronized with SonicPi) represents the AnimationFrame being auralized.

[8] https://sonic-pi.net

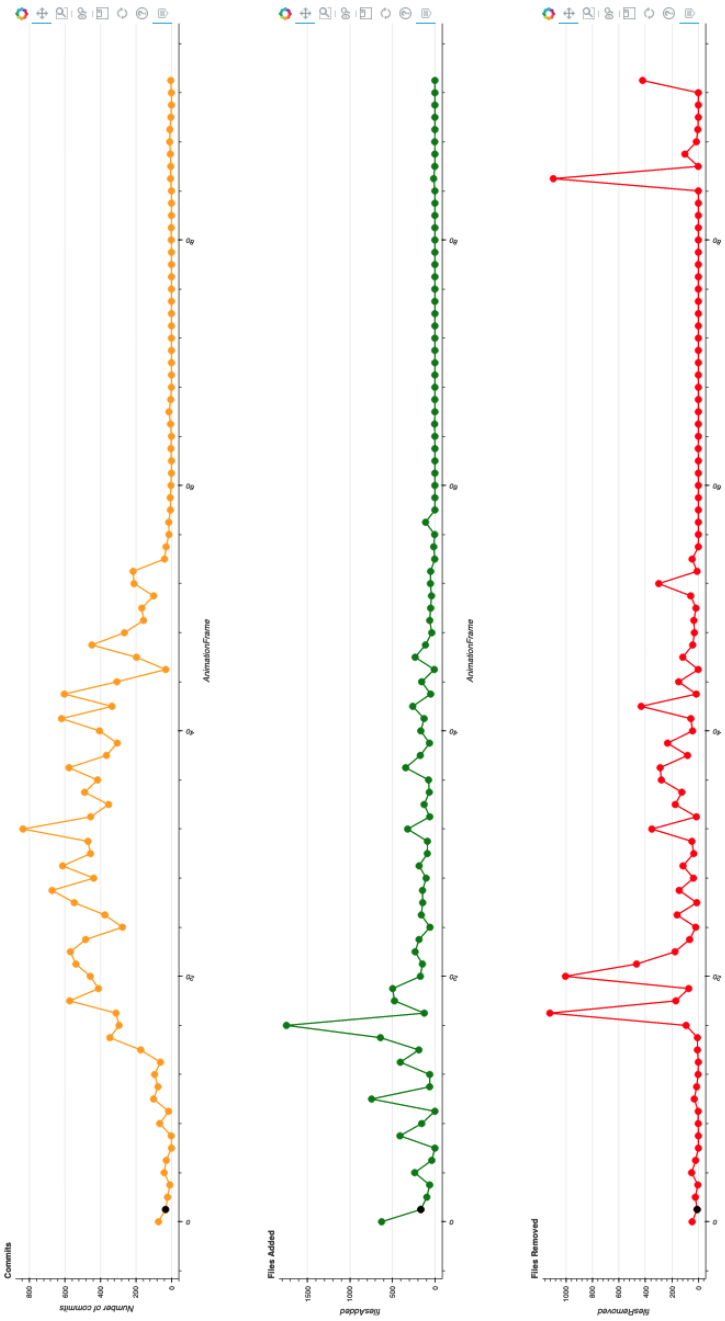


FIGURE 4.15: Charts used to debug the melody

Chapter 5

Case Studies

As a preliminary source of evaluation, this chapter presents case studies of SYN applied to open-source software systems. We analyze the evolution of JetUML, ArgoUML, Elasticsearch, LibreOffice, and Linux. Table 5.1 describes the characteristics of these systems.

Name	First Analyzed Commit	Last Analyzed Commit	# Of Commits Considered	# Of Files
JetUML	07 Jan 2015	12 May 2022	2,152	795
ArgoUML	26 Jan 1998	29 May 2021	16,672	11,129
Elasticsearch	28 Jun 2013	14 Apr 2022	34,842	41,043
LibreOffice	07 Mar 2002	18 Jun 2022	367,996	213,791
Linux	17 Apr 2005	19 Apr 2022	997,486	109,829

TABLE 5.1: List of analyzed projects

For all the visualizations in this Chapter, we decided to hide deleted entities, adopt a mapper with a LinearBucketValueStrategy with a maximum height of 20, and display files that do not have the selected metric on the visualization. Table 5.2 presents an overview of these properties.

Property	Value
showUnmappedEntities	true
mapperStrategy	LinearBucketValueStrategy
mapperStrategyOptions	max height of 20
showDeletedEntities	false

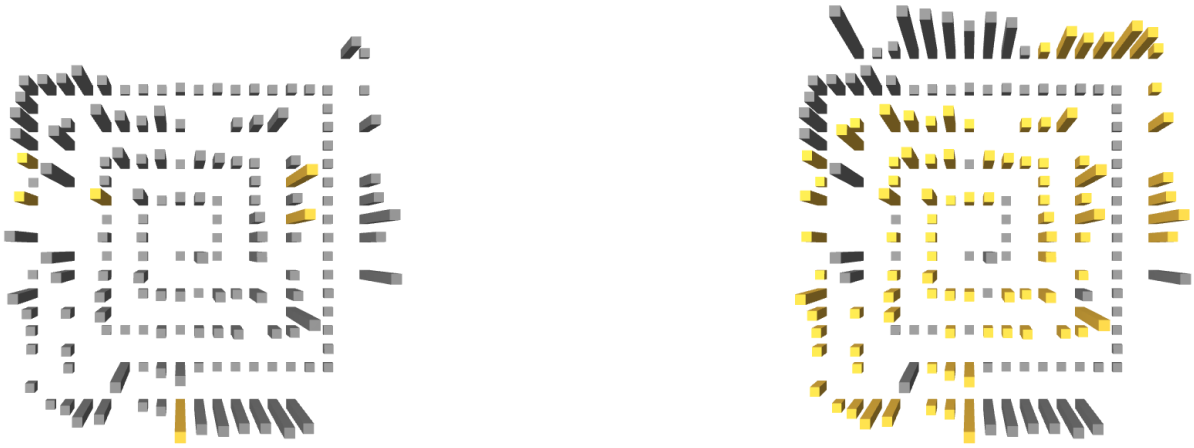
TABLE 5.2: Settings shared among all case studies

We made this choice because, during the development process, we have seen that the LinearBucketValueStrategy works better than other strategies. Of course, it needs a maximum height because otherwise, the gap between entities is too broad. Therefore we set this value to 20. Moreover, we believe that hiding the deleted entities is more intuitive for the user. On the other hand, hiding the unmapped entities might be misleading in some visualizations because the user might think these entities were deleted. For this reason, we preferred to keep them.

Deployment

We deployed two docker containers containing the Server[1] and the Visual Inspector[2] module at a publicly accessible location. We shipped these containers with the same data used in the following case studies. However, for performance issues, the Visual Inspector enables only the inspection of ArgoUML and JetUML, while data of LibreOffice, Elasticsearch, and Linux can be retrieved only through GraphQL.

[1] https://syn.si.usi.ch/api/graphql
[2] https://syn.si.usi.ch



(A) Commit that involved few entities

(B) Commit that involved numerous entities

FIGURE 5.1: Comparison of commits with a low and a high activity

5.1 JetUML – Evolution Overview

This section analyzes JetUML, an open-source desktop application to work with UML diagrams. It is hosted on GitHub.[3] Data collected in our analysis shows that the repository's history starts on January 7, 2015, and ends 2,013 commits later on May 12, 2022. The system's lifetime spans over seven years of activity. We detected 795 FileHistories and 10K FileVersions throughout these years.

Visualization goal: Visualize the entire repository commit by commit to see how the repository evolved and understand commits with high activity. To highlight them, we use a one-commit aging strategy, maximum age of two. In other words, files modified by the displayed commit have an age equal to zero, whereas others have an age equal to one. Consequently, all the files excluded in the depicted commit are painted with the base color (gray by default).

Figure 5.1 shows the graphical difference between a commit that modified only a few entities and another that changed numerous entities. Most entities on the left side are gray as the commit involved only a few entities. The commit on the right side is mainly yellow with changes in many entities. To identify highly active commits, we focused on commits that involved a considerable percentage of files.

The view specification of this visualization is shown in Table 5.3.

Results: Here we present some results that we found looking at the graphical evolution of JetUML. We report only a subset of the most active commits.

The repository's history starts on January 7, 2015, when Martin Robillard made the first commit. This commit added three files: README, a license, and a gitingore. Figure 5.2a represents it with three entities. They have the same shape because we did not adopt different shapes in the specification, but the height is different. The license file is taller than other entities because of its bigger size.

Fifteen minutes later, he pushed the initial codebase of a project named Violet, composed of 83 files and an updated version of the gitignore file. This commit is represented in Figure 5.2b, where 83 green entities represent added files, one yellow entity represents the updated gitignore files, and two gray entities, the README and the license files added before, that were not touched.

[3] https://github.com/prmr/JetUML

Property	Value
versionGroupingStrategy	commit
versionGroupingChunkSize	1
colorPalette	default
agingGroupingStrategy	commit
agingStepSize	1
agingSteps	2
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

TABLE 5.3: View Specification of JetUML – Evolution Overview

Forty minutes later, Martin Robillard made the first big refactoring of the system. His goal was to move the position of some fields in each class. This refactoring involved 49 files, and as shown in Figure 5.2c they are marked in yellow.

Three days later, after a series of delicate development tasks had been continuously made, he moved some classes from the Violet folder to a new one named Violetta. As we can notice from Figure 5.2d moved classes are represented in light blue.

Twenty-six commits later, on January 22, 2015, they moved classes from Violet and Violetta under the JetUML package. This commit (see Figure 5.2e) denotes the birth of JetUML as the first when this name was officially used.

After this first implementation, the system grew gradually, doubling its size in less than two years. During this time, they made only small commits to solve open issues. However, some exceptions were displayed in Figure 5.2f and Figure 5.2g where many files were modified because they had to update classes' copyright comments.

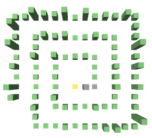
In November 2017, they removed `stg` from the path of all the Java classes. This commit is depicted in Figure 5.2h, where we can observe 162 light blue entities. They represent classes moved from the `../ca/mcgill/cs/stg/jetuml/...` path to the `../ca/mcgill/cs/jetuml/...` path. This commit also had 15 modified fields, and one added.

One last huge refactoring was done in July 2020, two years later, to change the copyright on every class. It is represented in Figure 5.2i, where we can notice many blank spots in the middle. This means that entities added in the early stages of the project have been removed and are no longer part of the system. At that time, the system had 316 entities, and 241 were affected by this commit.

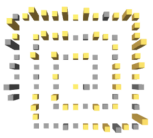
The last version of the system, created on Thu, May 12, 2022, counts 456 entities.

Conclusion: We manually inspected 2,153 frames depicting the evolution of JetUML using the autoplay feature of the Visual Inspector. Occasionally the authors of JetUML needed to make huge commits involving most files on their repository. Nonetheless, they were spotted easily, thanks to the adopted visualization settings. Most of the commits involved few entities; this means that features were implemented. With this grouping strategy, it was hard to understand the regularity and the speed of the development process. This is normal when we traverse the history by commit rather than by time.

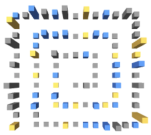
(A) 07 Jan 2015, 18:59
Initial commit



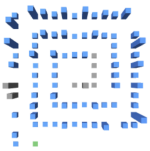
(B) 07 Jan 2015, 19:14
Initial Revision



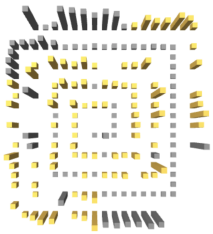
(C) 07 Jan 2015, 19:57
#1 Move all fields



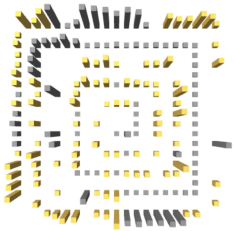
(D) 10 Jan 2015
#8 Moved to a dedicated ...



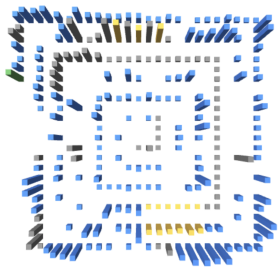
(E) 22 Jan 2015
#27 Renamed the packages



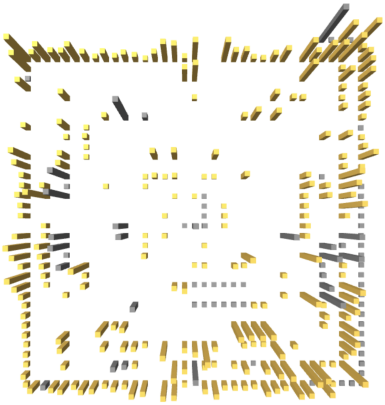
(F) 16 Oct 2015
#3 New copyright block on ...



(G) 17 Jan 2016
#146 Updated the copyright ...



(H) 26 Nov 2017
#212 Remove stg from name



(I) 22 Jul 2020
#374 Fix copyrights

FIGURE 5.2: Subset of the most significant commits in JetUML. Figure 5.2a presents the first commit. In Figure 5.2b they pushed the code of Violetta. Figure 5.2c depicts their first refactoring. In Figure 5.2d they moved some classes from Violet to Violetta. In Figure 5.2e they moved most of them under JetUML. Figure 5.2f, Figure 5.2g and Figure 5.2i present three refactoring activities to fix copyrights. In Figure 5.2h most entities are painted blue because they changed their path.

5.2 JetUML – The Beginnings

Visualization goal: With this visualization, we present how the repository evolved through its first six months. The authors of JetUML made seven pre-releases of the system during this interval of time. We want to demonstrate the effectiveness of aging by showing what a strategy of one month with ten steps looks like.

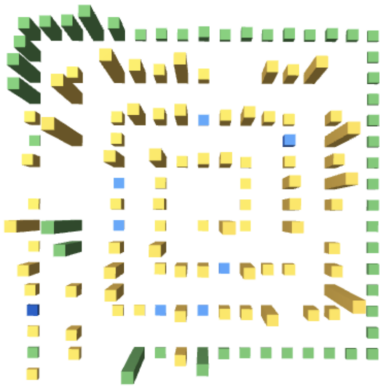
The view specification of this visualization is shown in Table 5.4.

Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	2,629,743 (1 month)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	10
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

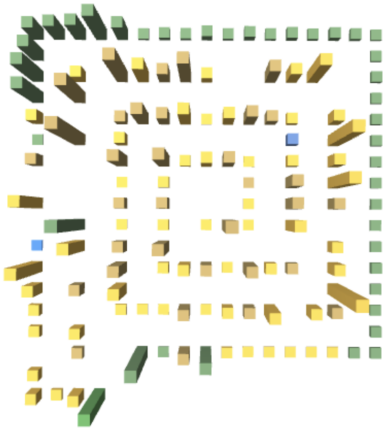
TABLE 5.4: View Specification of JetUML – The Beginnings

Results: Figure 5.3 shows the evolution of JetUML in each month of its first half-year development. Figure 5.3a shows the state of the repository after the first month. All entities are painted with the color representing the last action of the file. All the colors are bright because the age of all files is set to 0 since the time elapsed between their previous action and the last commit of the visualization is a maximum of one month. Furthermore, most of the moved files have the extension `.properties`, and they were involved in a refactoring activity since their path changed from `src/ca/mcgill/cs/stg/jetuml/...` to `src/ca/mcgill/cs/stg/jetuml/diagrams/...`. At the end of their first month of activity, they made 127 commits and worked with 135 files. The second month is depicted in Figure 5.3b. Here we can easily spot older entities as they are darker than others. This means that some files had not been touched since the first month at the end of the second month. In the third and the fourth month, more or less the same amount of files were updated. However, in the fifth month, only three entities were modified; consequently, all the others are darker because they are older. At the end of the sixth month, two entities were added and modified. However, most of the entities were still getting older.

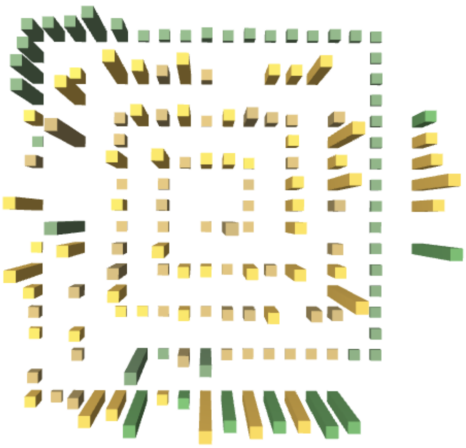
Conclusion: We have seen how different is the timestamp strategy compared to what we saw in Figure 5.2. Here, the number of frames does not depend on the number of commits but on the number of months of history. The choice of the aging strategy played a crucial role in this visualization. It highlighted the updated entities without hiding the last action made on the others. Looking at Figure 5.3f, we can see that many files were added and never touched because their color looks dark green. This inactivity on the files may be related to the file type. Java files might be more prone to be updated than configuration files such as YAML files. Nonetheless, the choice of representing all the entities with the same shape (BOX) limits our visualization because we cannot infer their type in any way.



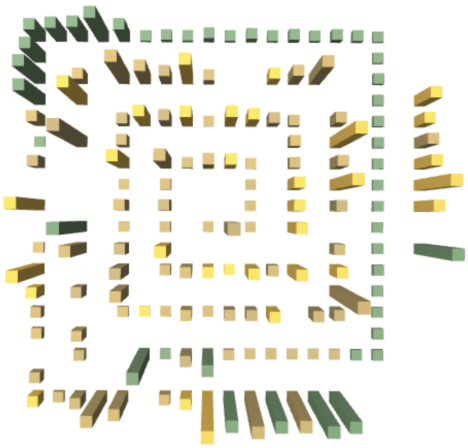
(A) 1st month - Feb 07, 2015



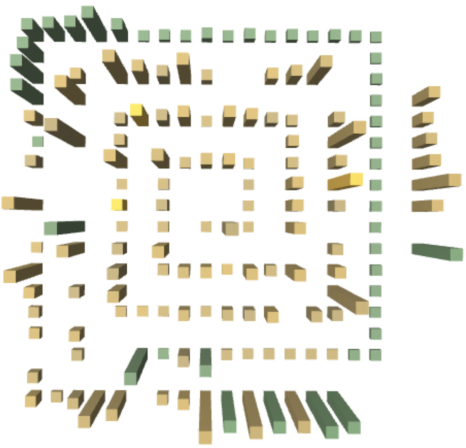
(B) 2nd month - Mar 07, 2015



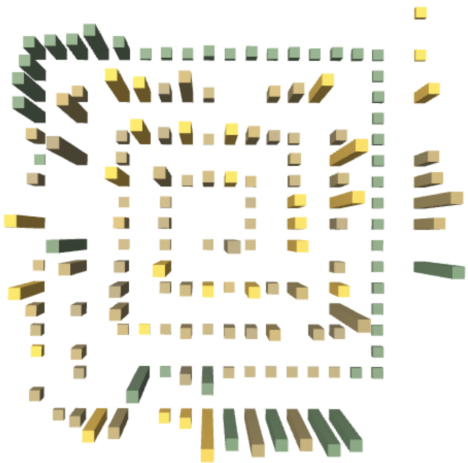
(C) 3rd month - Apr 07, 2015



(D) 4th month - May 07, 2015



(E) 5th month - Jun 07, 2015



(F) 6th month - Jul 07, 2015

FIGURE 5.3: First six months of JetUML's evolution.

5.3 JetUML – Year by Year

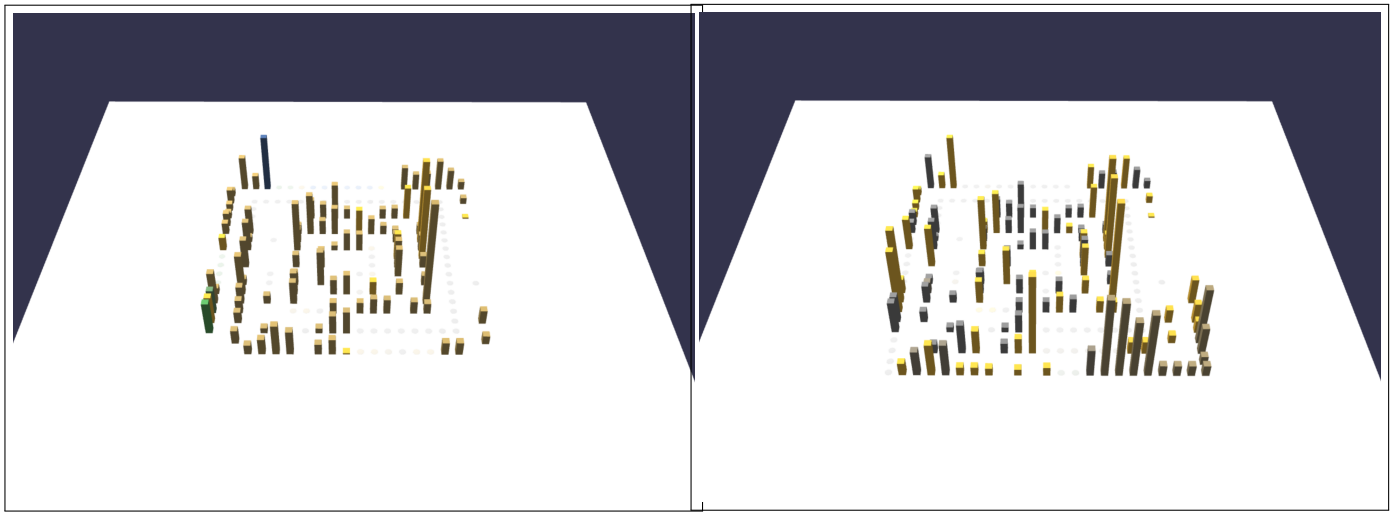
Visualization goal: In this third and last visualization of JetUML, we study the entire evolution of the system, with a focus on Java files. We use the shape and opacity property of a ViewFigure to distinguish Java files from others. We adopt a timestamp grouping strategy with one year to display the evolution. We also use a timestamp strategy for the aging, with ten steps and a one-month time window. This means that grey entities represent files that have not been updated in the last ten months from the time of the displayed AnimationFrame. The view specification of this visualization is shown in Table 5.5.

Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	31,556,926 (1 year)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	10
mapperMetricName	SIZE
fileTypeShape	Java -> BOX, OTHERS -> SPHERE
fileTypeOpacity	Java -> max, OTHERS -> low

TABLE 5.5: View Specification of JetUML – Year by Year

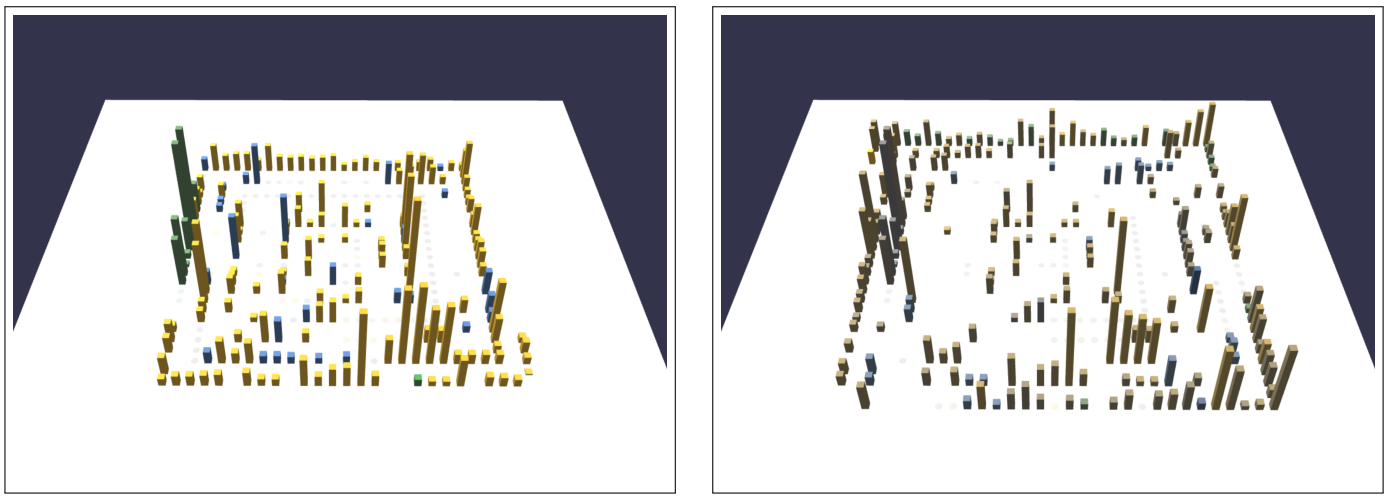
Results: The whole evolution of JetUML is depicted in Figure 5.4. As we notice, it is straightforward to distinguish Java files from others because they have different shapes (cuboids vs. hemispheres) and other files are less opaque. After the first year of development, the state of the repository is shown in Figure 5.4a. We notice that the Java files have more or less the same age. This means that all of them were updated after the second month otherwise, they would be grey. The system grew at the end of the second year, as shown in Figure 5.4b, and now we can see some grey entities. Most of them are in the middle, meaning that the "core" files, or the oldest files, were not touched after the fourteenth month. Figure 5.4c shows the state of the repository at the end of the third year. The system grew, and all the Java files were updated recently. Considering what we have seen in the first visualization, they had to change the path from all the Java files (Figure 5.2h). Perhaps this can be why all the entities were updated, and some ViewFigures are painted blue. The fourth year of development, shown in Figure 5.4d, recorded a growth in the first month; therefore, almost all the entities have a different color. However, they are dark, meaning they are old. Figure 5.4e, Figure 5.4f and Figure 5.4g show how the system evolved through the fifth, sixth, and seventh years. The fifth year recorded an intense activity compared to what we saw in the fourth year. Lots of files were added, and most of the classes of the system were modified. Furthermore, the system's center is slowly disappearing, meaning old Java files have been replaced. In the sixth and seventh years, non-Java files were added, and some Java classes were updated outside the system's core. Finally, the last year, represented by Figure 5.4h, did not record such a huge activity on the system. It became inactive as almost all the entities are grey, meaning that they were not touched in the last ten months of evolution. However, in this frame, the two tallest entities of the system appeared. Two Java files were added (they were not present the year before) and rapidly became the biggest files in the repository.

Conclusion: With this visualization, we have seen JetUML from another point of view. The year visualization works well with systems like JetUML with a large history. In fact, with just eight AnimationFrames, we depicted the evolution of the last nine years. In addition, we identified moments in the past when the development process was more intense.



(A) 1st year

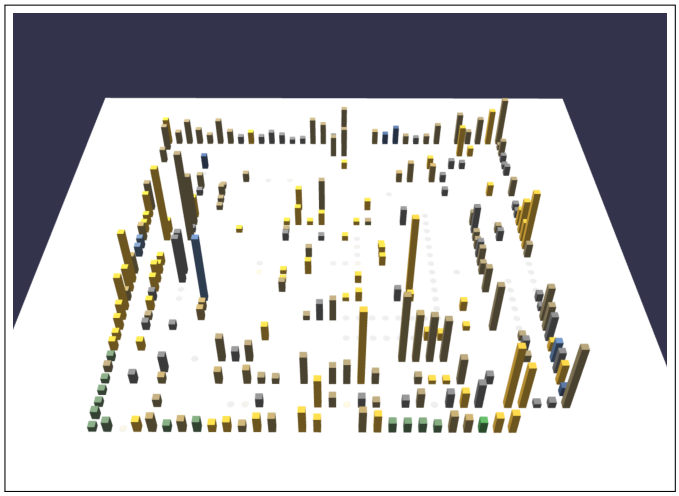
(B) 2nd year



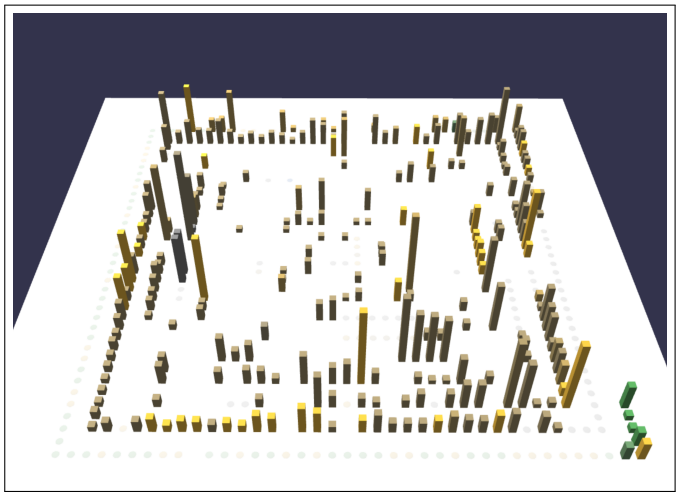
(C) 3rd year

(D) 4th year

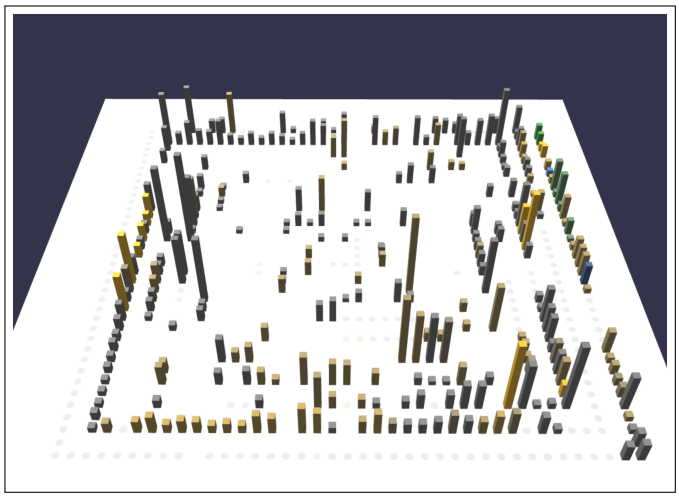
FIGURE 5.4: Evolution of JetUML



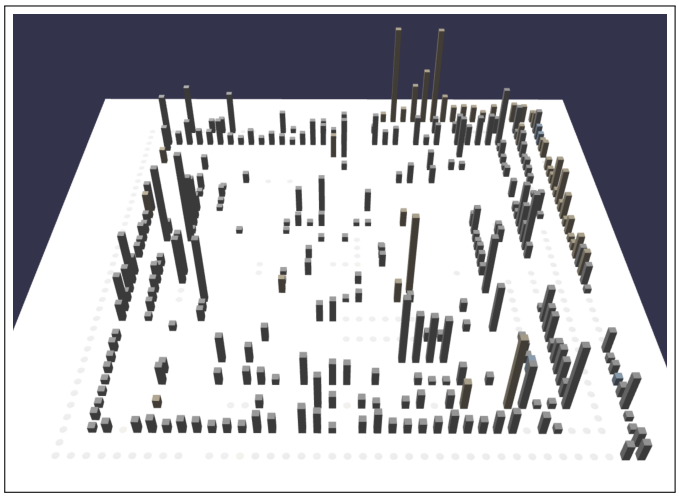
(E) 5th year



(F) 6th year



(G) 7th year



(H) 8th year (last)

FIGURE 5.4: Evolution of JetUML

5.4 ArgoUML – 23 Years of History

This section analyzes ArgoUML, a leading open source UML modeling tool. Hosted on GitHub.[4] This repository was created in January 1998, converted from CVS to Subversion in 2006, and finally moved to git in 2019. It has more than 23 years of history. Our analysis considered 16,672 commits, 11,129 FileHistories, and more than 90K FileVersions.

Visualization goal: This visualization aims to understand how the repository evolved through these 23 years. Therefore we adopt a commit grouping strategy based on a time window of one year and an aging strategy of one month with a total of 12 steps. Hence, grey entities represent files that have not been updated for more than 12 months.

The view specification of this visualization is shown in Table 5.6.

Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	31,556,926 (1 year)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	12
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

TABLE 5.6: View Specification of ArgoUML – 23 Years of History

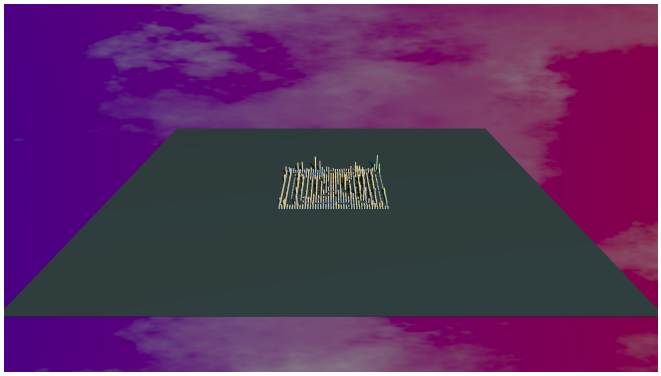
Results: Here we present an interesting summary of ArgoUML's evolution. The full evolution is depicted in Appendix B. Figure 5.4 shows the hotspots during the system's evolution. Figure 5.5a shows the system's state after the first year of development. As we can notice, almost all the entities have a color different than gray. This means they were updated between April 1998 and January 1999 otherwise, they should be darker or gray. The size of the system was constantly increasing in 2003. As Figure 5.5b shows, the system's core was entirely rewritten. Three years later, it had another important refactoring. As shown in Figure 5.5c there are empty rings around the center and a bigger size than the previous year. Our interpretation is that an entire set of consequently added files was removed and maybe later added. It is important to underline that these files were deleted, not moved, or renamed because otherwise, SYN would treat them as the same logical entity. The same event happened four years later when the system became ten years old. 2010 was the last truly active year for the repository, as depicted in Figure 5.5e. Almost all the repository files were modified, and we can notice taller entities, meaning that the overall size of the files increased. The system did not grow anymore after that year. Sometimes some files were updated until 2014, when they became almost inactive. In 2019 they made a commit that removed lots of files.[5] As a result, in January 2020, as represented in Figure 5.5f, the visualization has more blank spots. Since there was no activity, except for a few files, almost all the entities are painted gray. In June 2022, the system is nearly identical to its version in 2020.

Conclusion: The visualization shows the composition of JetUML and its development pace. Thanks to the spiral layout, where entities are added based on their timestamp, we can see that many older files were removed in the final version. They had to be rewritten elsewhere because the system would have fewer features than before. Moreover, thanks to the aging strategy, we observed breaks in the development of the

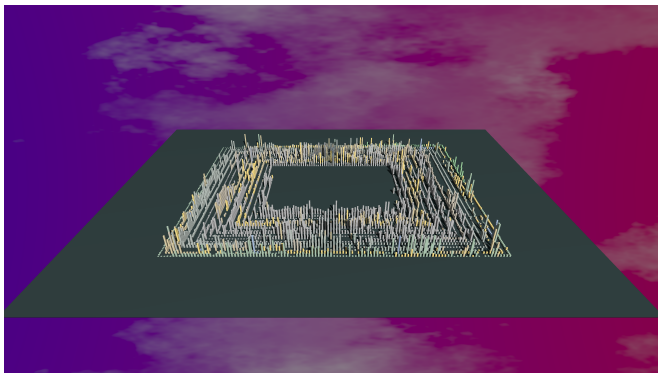
[4] https://github.com/argouml-tigris-org/argouml

[5] https://github.com/argouml-tigris-org/argouml/commit/62d5393eeaf08b749b17107faef255a22da55ce5

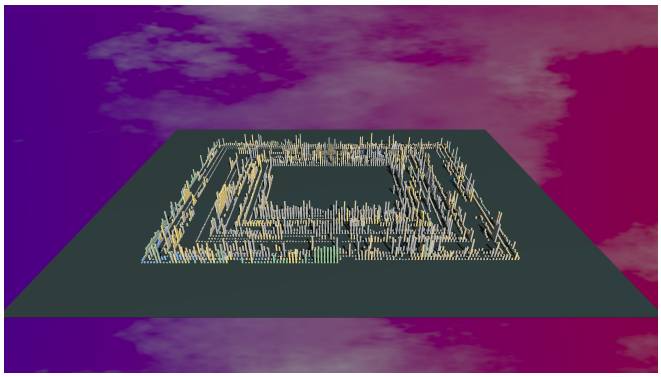
repository. Nonetheless, some of them are not shown in Figure 5.5, they can be found in Appendix B. The system became almost inactive in 2014. Since that year, most files were never updated.



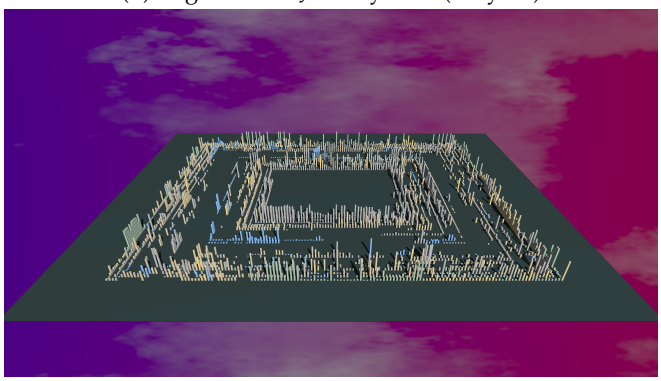
(A) ArgoUML in January 1999 (1st year)



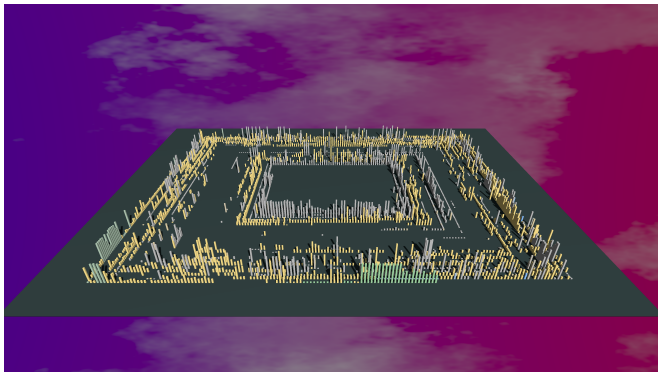
(B) ArgoUML in January 2003 (5th year)



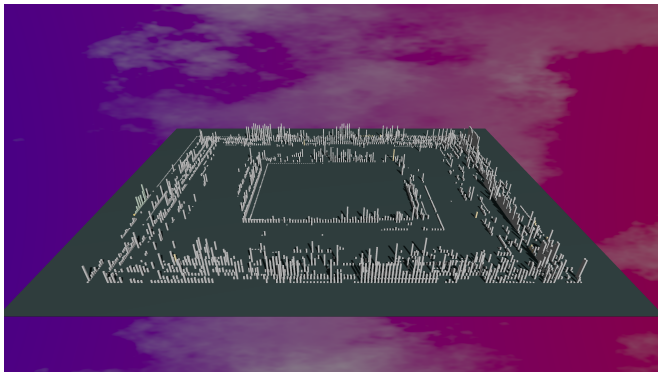
(C) ArgoUML in January 2006 (6th year)



(D) ArgoUML in January 2008 (10th year)



(E) ArgoUML in January 2010 (12th year)



(F) ArgoUML in January 2020 (22th year)

FIGURE 5.5: Hot spots in the evolution of ArgoUML

5.5 Elasticsearch – Forty Thousand File Histories

This section analyzes Elasticsearch, a distributed, RESTful JSON-based search, and analytics engine. The project is open-source and hosted on GitHub.[6] The first commit was made in June 2013, even though it was not the real first one because the project started before being migrated to GitHub, in 2009. Our analysis includes almost nine years of history, in which there are 34,843 ProjectVersions, 41,043 FileHistories, and more than 300K FileVersions.

Visualization goal: This visualization aims to see how the repository evolved through its last nine years. We adopt a commit grouping strategy based on a time window of one year and an aging strategy of one month with 12 steps. Hence, grey entities represent files that have not been updated for more than 12 months.

The view specification of this visualization is shown in Table 5.7.

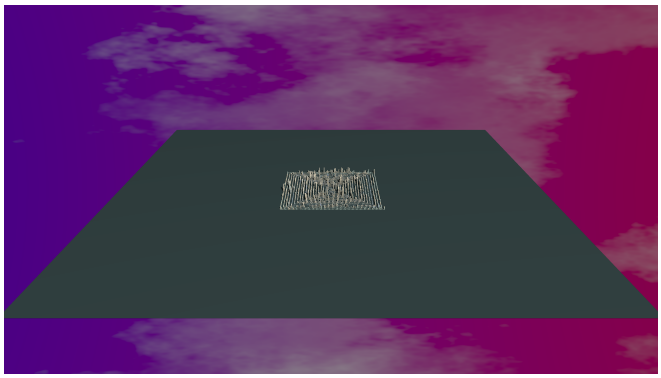
Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	31,556,926 (1 year)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	12
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

TABLE 5.7: View Specification of Elasticsearch – Forty Thousand File Histories

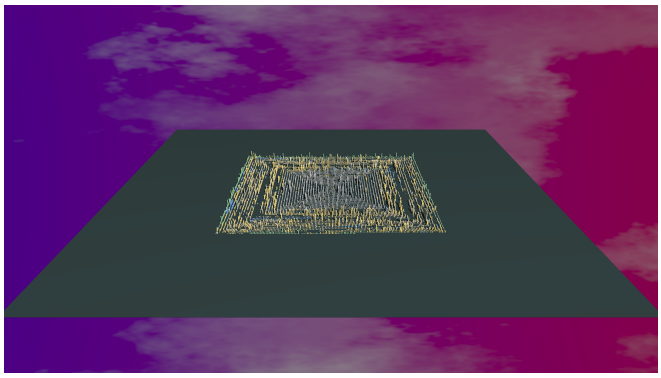
Results: Here we present only the relevant aspects of ArgoUML's evolution that we have found and present its full story in Appendix C. Figure 5.6 shows the hot spots during the system's evolution. Figure 5.6a shows the initial size of the system when the system became open source and was officially moved on GitHub. After one year of inactivity, the size of the system in June 2016 was almost triple of two years before. In June 2014, it had 2,904 files; in June 2016, it had 9,537 files. However, in Figure 5.6b, we notice that only the entities around the center of the system were touched. This means that the initial implementation of Elasticsearch remained untouched and that not many of the files added in 2013 were modified. Perhaps the code was stable enough that it did not require much maintenance, and only new features were added before June 2016. After a one-year break, in June 2018, they had another considerable increment in the system, as shown in Figure 5.6c. It doubled its size again. Interestingly, the core remained untouched, except for very few changes. Moreover, the height of entities present in Figure 5.6b became higher. This not only means that they were modified, but it might also mean that new features were added to these classes. We recorded the same growth pattern later until June 2022. As depicted in Figure 5.6d, Figure 5.6e, and Figure 5.6f the system incremented the number and size of files gradually. The most important thing is that, except for the center, most of the files are painted with a bright color, meaning they are maintained.

Conclusion: This analysis highlighted interesting aspects of Elasticsearch's evolution. The core of the system, once developed, was never touched, except for very few entities. As we can notice in Figure 5.6, the core is always depicted in gray, even when files around it are modified in the last year of evolution.

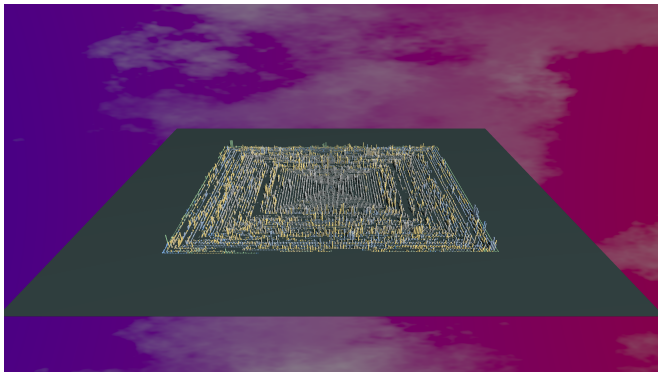
[6] https://github.com/elastic/elasticsearch



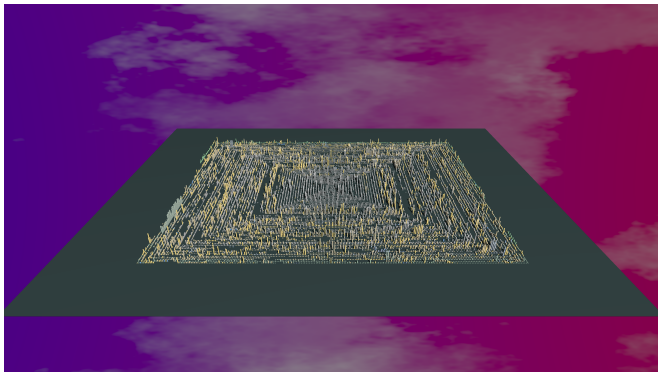
(A) Elasticsearch in June 2014 (1st year)



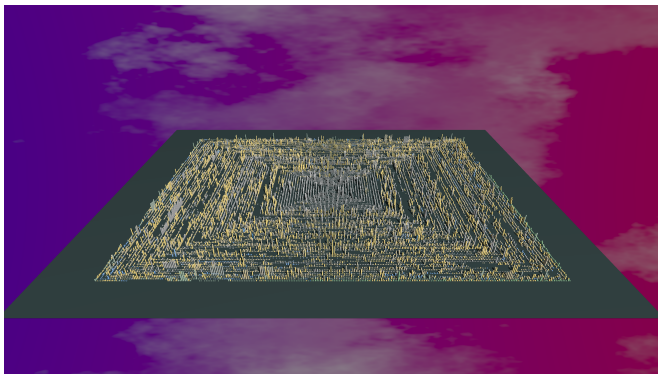
(B) Elasticsearch in June 2016 (3rd year)



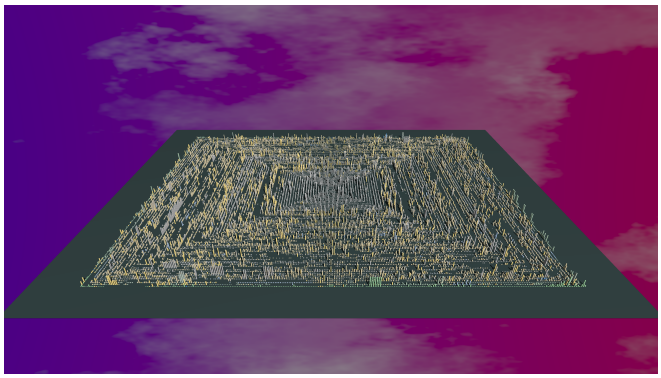
(C) Elasticsearch in June 2018 (5th year)



(D) Elasticsearch in June 2019 (6th year)



(E) Elasticsearch in June 2021 (8th year)



(F) Elasticsearch in June 2022 (9th year)

FIGURE 5.6: Hot spots during the evolution of Elasticsearch

5.6 LibreOffice – 200K Files in 20 Years

LibreOffice is an integrated office suite compatible with most document formats and standards. Its first commit,[7] made on March 7, 2002, was an import of the original codebase developed by Sun Microsystems. The repository's history is 20 years long, with 327,996 commits, 213,791 FileHistories, and almost 2M FileVersions.

Visualization goal: This visualization aims to see how the repository evolved through its last 20 years. We adopt a commit grouping strategy based on a time window of one year and an aging strategy of one month with 12 steps. Hence, grey entities represent files that have not been updated for more than 12 months.

The view specification of this visualization is shown in Table 5.8.

Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	31,556,926 (1 year)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	12
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

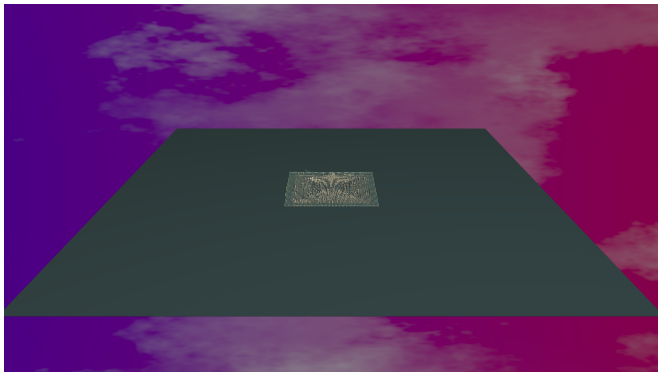
TABLE 5.8: View Specification of LibreOffice – 200K Files in 20 Years

Results: We assess only the relevant aspects of the evolution that we have found. The full evolution is depicted in Appendix D. Figure 5.7 shows the hot spots during the system's evolution. The initial composition of the system is shown in Figure 5.7a. Between 2003 and 2011, the system grew substantially. In March 2003, it had 13,500 files, and in March 2011, it had 57,520 files with an increment of over 40K files. Figure 5.7b shows the state of the repository in March 2011. Few entities are painted with a color, meaning that only a few parts of the codebase were touched. Figure 5.7c shows the state in March 2012. Closely to that date, the repository underwent a big refactoring where most of the recently added files were moved. In this picture, we initially start to see empty rings like in the analysis of ArgoUML. Another important aspect of this picture is that the core was entirely modified. As we can see, it is yellow, and the entities are taller than the year before. The work on the repository was constant until June 2022. In all the AnimationFrames, the center was always painted with yellow colors. This could mean that the core still needs maintenance after 20 years, and recently developed features heavily rely on the core implementation. In the last 12 years, as can be seen in Figure 5.7d, Figure 5.7e and Figure 5.7f the system doubled its size. In June 2022, it counted 135,793 files. Empty rings around the core became thicker, suggesting the deletion of a significant part of the system, which they probably reimplemented later.

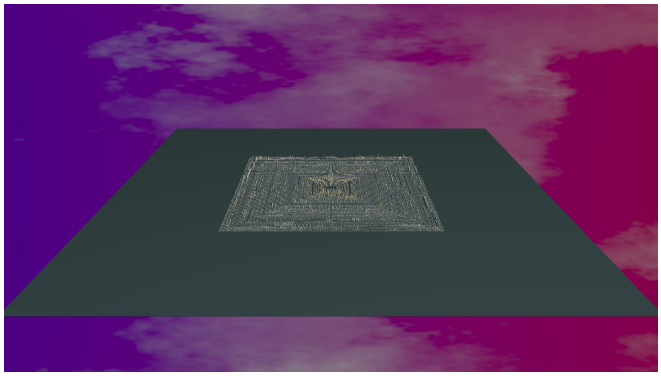
Conclusion: The evolution of LibreOffice was ongoing throughout its 20 years of history. Thanks to the aging concept of SYN, we understood that they heavily rely on code that was added 20 years ago. The core of the system was regularly updated, and the overall height of entities grew, especially in the middle. The empty rings around the core represent a pattern that we have already recognized in ArgoUML. It means that a set of files were removed one after another. Perhaps these files might be in an entire folder that contained a feature that was later readded. One issue that this visualization raised is regarding the size of the system. It is a limitation of this static approach because it is hard to infer the color of some entities

[7] https://github.com/LibreOffice/core/commit/5acbb755f3f8dedf51904b711ff78f5e1aa498cf

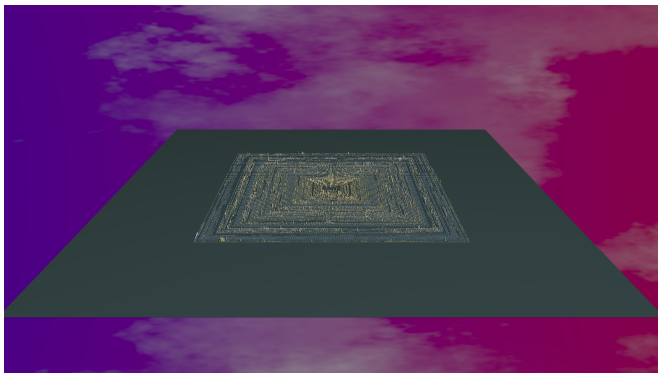
since they are too small to be individually identified. An interactive tool like the Visual Inspector module (Section 4.6) would not have this limitation because it allows zooming.



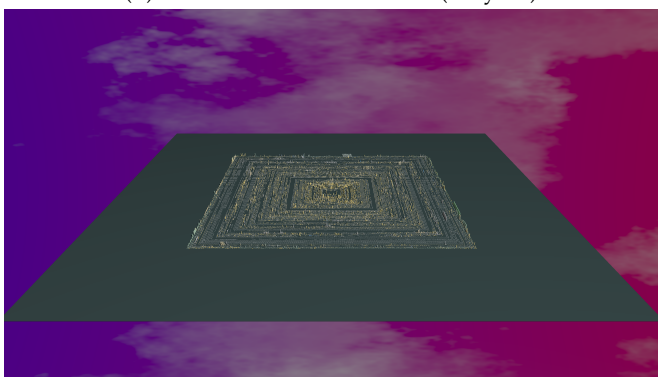
(A) LibreOffice in March 2003 (1st year)



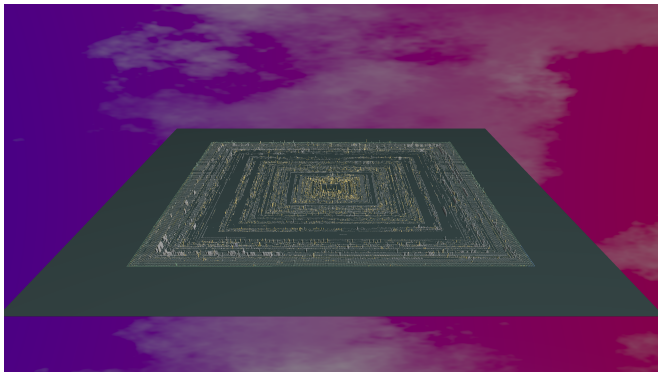
(B) LibreOffice in March 2011 (9th year)



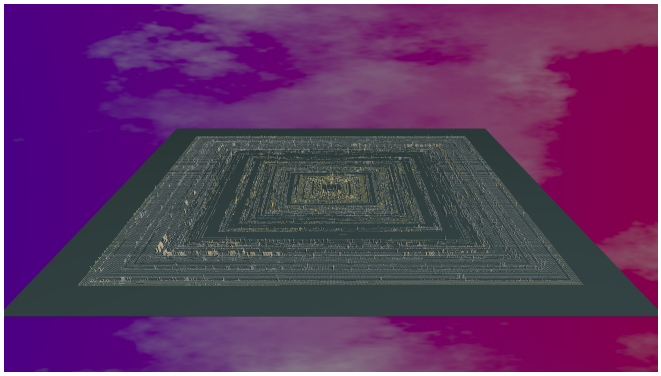
(C) LibreOffice in March 2012 (10th year)



(D) LibreOffice in March 2013 (11th year)



(E) LibreOffice in March 2019 (17th year)



(F) LibreOffice in June 2022 (21th year)

FIGURE 5.7: Hot spots during the evolution of LibreOffice

5.7 Linux – Over a Million Commits

Linux is an open-source kernel widely used and known for its efficiency and reliability. Its code is published on GitHub.[8] Its development was started in 1991 by Linus Torvalds. The first commit pushed in its git repository is dated 17 April 2005. The reason is explained in the message of the first commit:[9]

> *"Initial git repository build. I'm not bothering with the full history, even though we have it. We can create a separate "historical" git archive of that later if we want to, and in the meantime it's about 3.2GB when imported into git - space that would just make the early git days unnecessarily complicated, when we don't have a lot of good infrastructure for it."*

He never created a repository containing the entire history of Linux. The analysis of this repository detected almost 1M commits, with 109,829 FileHistories and more than 2M FileVersions.

Visualization goal: This visualization aims to see how the repository evolved through its last 17 years. We decided to adopt a commit grouping strategy based on a time window of one year and an aging strategy of one month with 12 steps. Hence, grey entities represent files that have not been updated for more than 12 months.

The view specification of this visualization is shown in Table 5.9.

Property	Value
versionGroupingStrategy	timestamp
versionGroupingChunkSize	31,556,926 (1 year)
colorPalette	default
agingGroupingStrategy	timestamp
agingStepSize	2,629,743 (1 month)
agingSteps	12
mapperMetricName	SIZE
fileTypeShape	all BOX
fileTypeOpacity	all 1

TABLE 5.9: View Specification of Linux – Over a Million Commits

Results: We present only a subset of Linux's significant evolution aspects. The full depiction is shown in Appendix E. Figure 5.9. Figure 5.9a shows the system's state in April 2006, after its first year on GitHub. At that time, the repository had 19,709 files. The development of the repository was active for the next four years. In April 2009, shown in Figure 5.9b, almost all the files were touched. Moreover, we can start to see a ring around the core. 12 years after the first commit, the system has grown to three times its initial size. Figure 5.9c shows its state at that time. We can start to notice that the spiral's center has fewer files than the edge. This is a good thing it means that old files are rewritten into newer ones. Intense development activity on Linux continued uninterruptedly until April 2022. As we can see from Figure 5.9d, Figure 5.9e and Figure 5.9f during its recent growing process, the center slowly became less dense. This is a sign that older files are now rewritten into newer ones.

[8] https://github.com/torvalds/linux
[9] https://github.com/torvalds/linux/commit/1da177e4c3f41524e886b7f1b8a0c1fc7321cac2

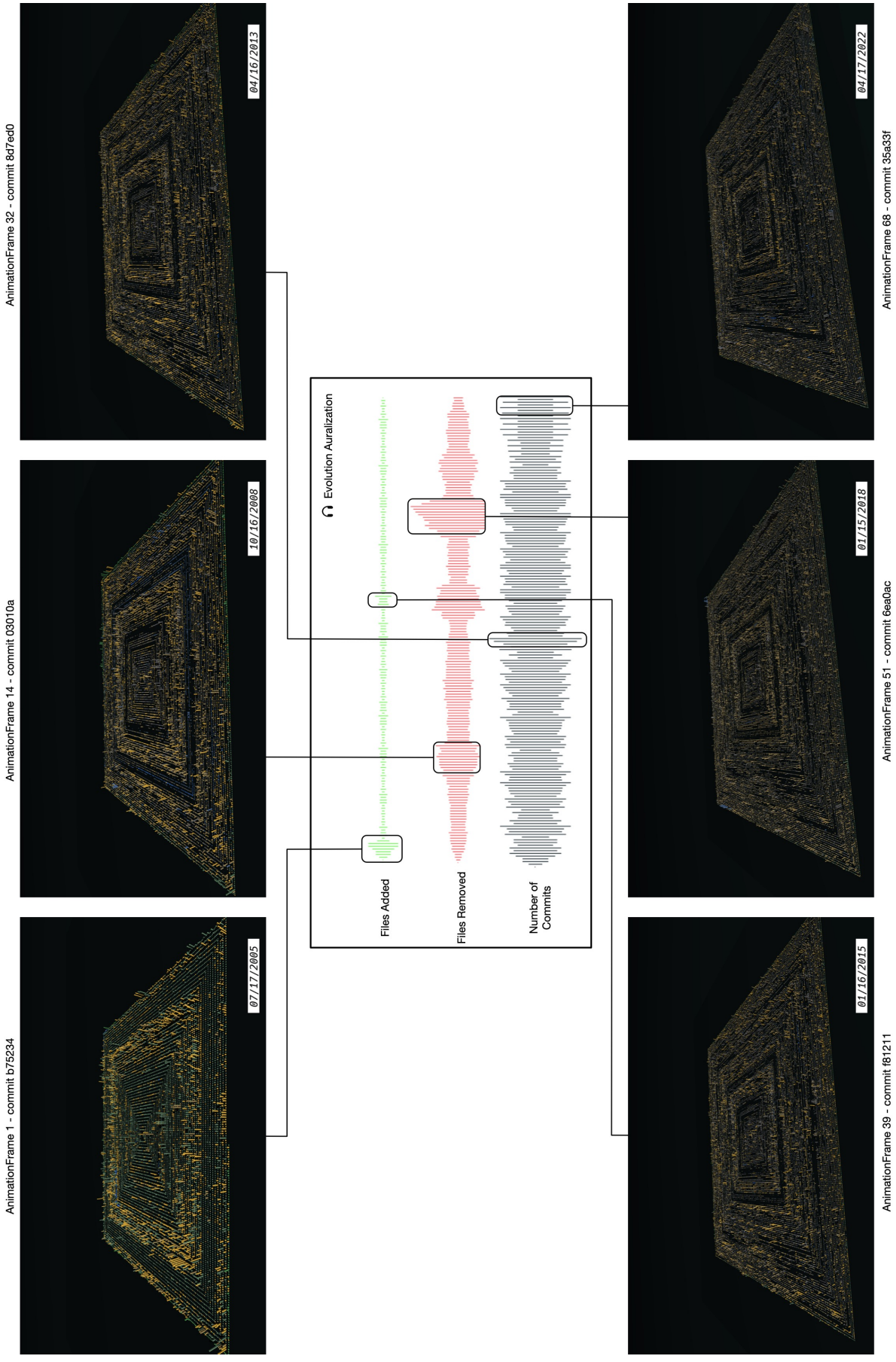


FIGURE 5.8: Frames of Linux's Evoltution

Auralization: We combined our visualization and auralization approach in a video depicting the evolution of Linux.[10] The video shows how the Linux repository evolved in 17 years between April 2005 and April 2022.

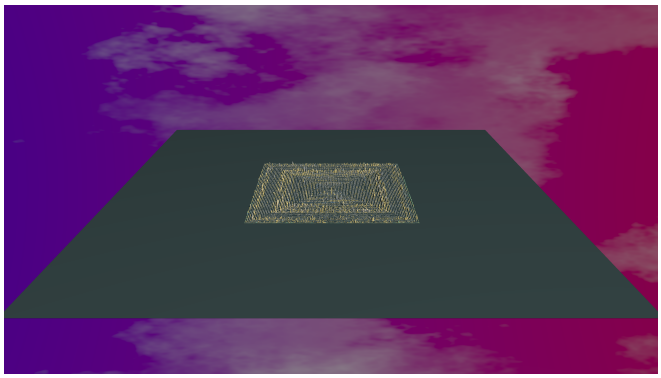
Figure 5.8 shows a visual representation of sounds generated from Linux, with selected frames of its evolution. The video starts with a peak on the number of files added. As we can see from AnimationFrame 1, the size of the system became massive in the first three months of evolution. Linus Torvalds, the author of Linux and the maintainer of the repository in April 2005, created a new repository to host the Linux codebase instead of converting the existing one to Git. He explained the reasons for this choice in the first commit.[11]

In October 2008, as shown in AnimationFrame 14, there was a peak in the number of removed files. Consequently, the prevalence of the sound representing deleted files is high. In this video frame, the center of the spiral starts to become sparse. As highlighted in AnimationFrame 32, the development activity was constant and very high during this seventeen-year interval. This frame has many files painted with a bright color, meaning they were updated slightly before the frame's date. In the end, from AnimationFrame 68, we can understand the actual size of the system.

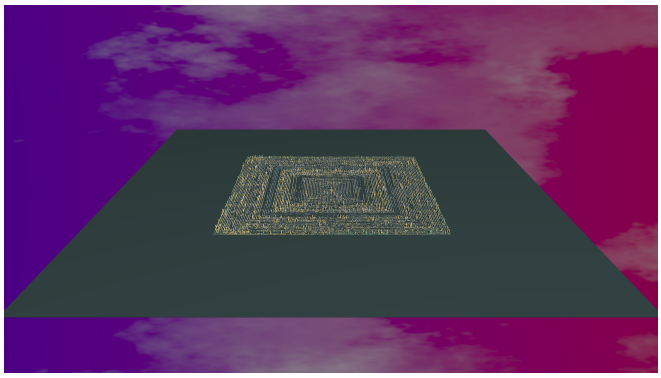
Conclusion: We saw how the codebase of Linux evolved since their move to git. The development activity was constant throughout these 17 years. The codebase started with 19,705 files and reached 77,183 in April 2022, almost four times its original size. During this time, they evolved the kernel to develop new features and started a process to slowly remove the files that were added 17 years ago. As we can see, the center of the spiral began to become sparse. Nonetheless, it still holds many files, which means that the current version of Linux relies on files written more than 17 years ago.

[10] https://www.youtube.com/watch?v=iXGQoO7WAA0

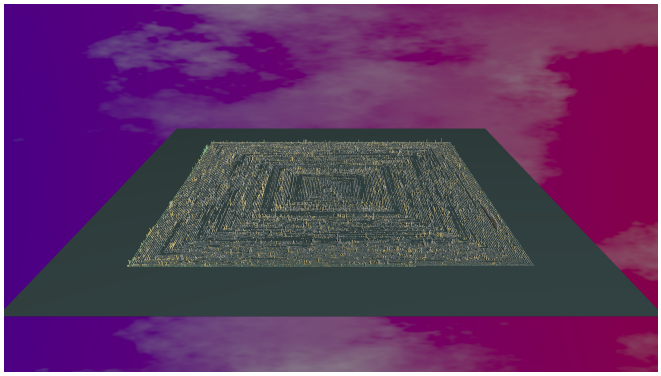
[11] https://github.com/torvalds/linux/commit/1da177e4



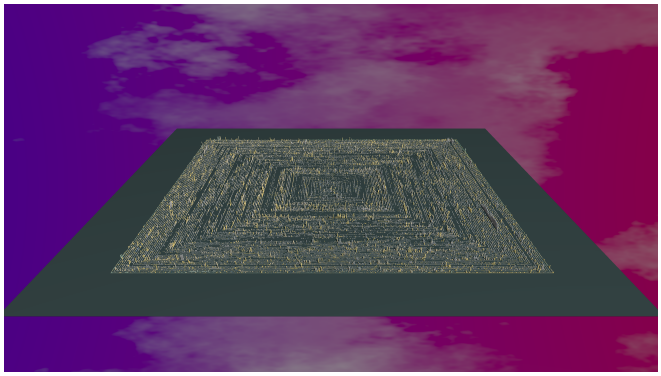
(A) Linux in April 2006 (1st year)



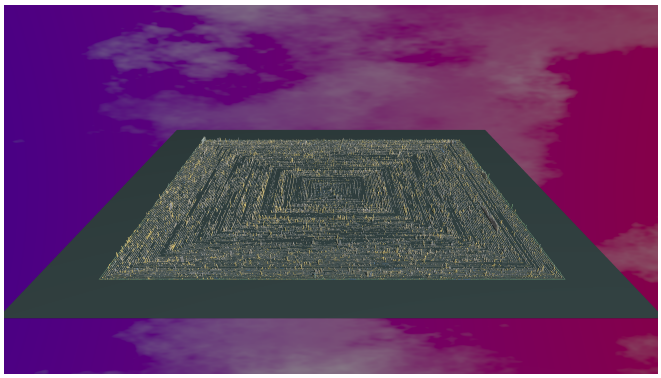
(B) Linux in April 2009 (4th year)



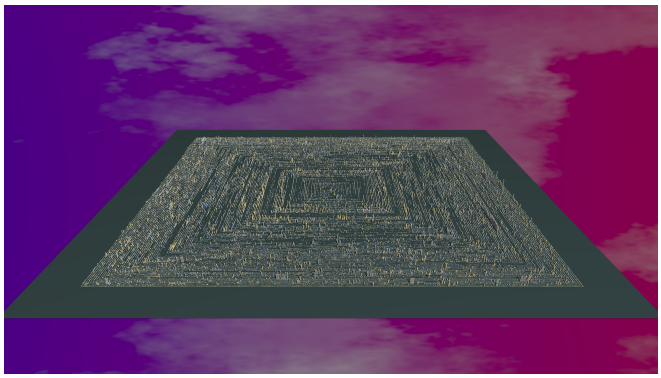
(C) Linux in April 2017 (12th year)



(D) Linux in April 2019 (14th year)



(E) Linux in April 2020 (15th year)



(F) Linux in April 2022 (17th year)

FIGURE 5.9: Hot spots during the evolution of Linux

5.8 Summary

To give the reader an idea about the size of the systems, we took the last AnimationFrame of each project and put them next to each other in Figure 5.10. Even though Linux was initially bigger than LibreOffice, the latter grew sharply in the last year, overtaking the biggest repository on GitHub. Furthermore, we can observe each system's different development paths. ArgoUML is the only repository that removed all the old files. Elasticsearch seems to rely on its core heavily. As we can see, it is dense, and we also saw during its evolution that it was updated frequently. Of course, these two projects have different histories. ArgoUML has commits dated 1998, and the first commit of Elasticsearch was made in 2013. Equally important is the state of Linux and LibreOffice. Both of them started to move some files from the core to the edges. However, both work with files older than 17 years.

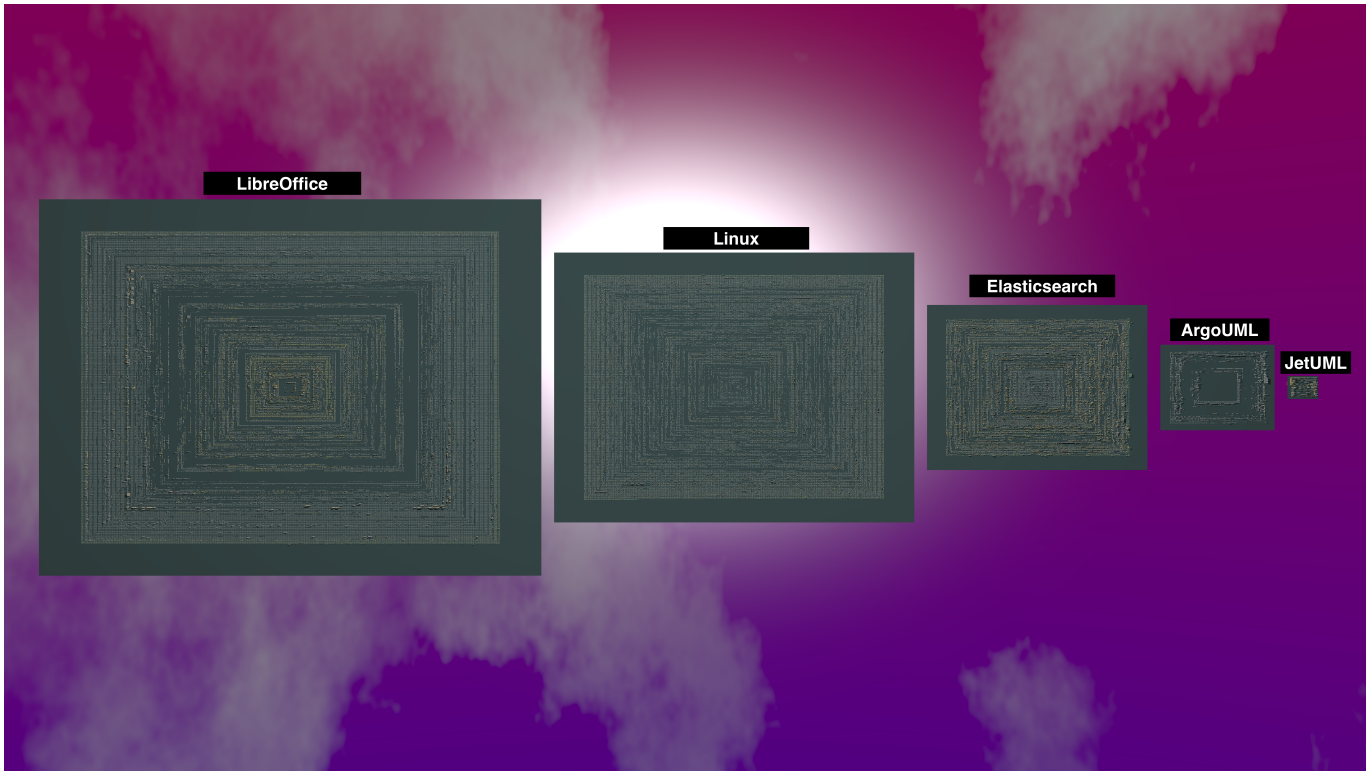


FIGURE 5.10: Comparison of the projects' state in June 2022.

Chapter 6

Conclusion

In this chapter, we summarize the contributions of this thesis, discuss our approach's benefits, and outline possible future work.

6.1 Contributions

This thesis presents three new approaches to mine, visualize, and auralize large software repositories. The goal of this thesis was to explore new solutions to represent software evolution. We cover all the stages required to reconstruct the history of a git repository, from the historical collection of information to the graphical data representation and visualization. We developed it with a language-agnostic approach. Therefore, an extension of the system is needed to collect and visualize specific kinds of information. Our contributions can be summarized as follows:

1. **The mining and modeling of large git repositories**. We present an approach to rebuilding the history of a git repository by traversing the repository's history. It starts from the first commit and analyzes all the subsequent commits until the end. For each commit, we extract information about the modified files, parse it and finally serialize it in a file on the local storage. The root element of our model is the history of a project that holds a set of and a set of files. An action recorded by a commit made on a file is also represented. The main benefits provided are the following:

    - Logical tracking of files, each file's history is represented by a separate object in the model. When a file is moved or renamed, the model keeps track of this action.
    - The analysis process is automatic.
    - The analysis process can be scaled horizontally.

2. **The sensorial software evolution visualization**. It exploits synesthesia, the production of a sense impression relating to one sense by stimulating another. It represents the evolutionary process through an interactive visual depiction of evolving software artifacts. It enables the comprehension of both the structural and the evolutionary prospectives. Moreover, SYN, the tool that implemented this approach, allows the user to customize visualization properties. This approach provides the following benefits:

    - It provides an interactive way to comprehend software evolution.
    - It is a scalable approach. Therefore it works with large git repositories like Linux and LibreOffice.
    - It allows the user to customize many graphical properties such as colors, shapes, and height. Therefore, the user can leverage a customized experience to trigger as many collateral cognitive paths as possible.
    - It provides two strategies to traverse the repository history: by commit or by time. This way, the comprehension of repositories with a very large history is not undermined.

- Thanks to the aging concept, the user can infer how old a file is. Age differences can be expressed both with commits or with time and are useful to understand how many moments have passed since the last action.
- The height of a file is mapped to the value of a pre-selected metric. We provided multiple mapper strategies.
- The shapes and the opacity can be customized to highlight a file type over others.
- The positioning strategy allows the user to understand the additional order of files immediately. This way, they can understand if the repository still holds legacy files.

3. **Auditive portrayal of the evolution**. It consists of a guideline suggesting how to compose audio sounds representing the evolution of a project. The proposed methodology was tested with one system and presented with the Linux case study analysis. The main benefit provided by this approach is the support of the visualization. It provides additional information without displaying them. Consequently, the listener can infer additional aspects of the analysis, such as the number of commits and the number of files added/removed from each AnimationFrame.

6.2 Future Work

In this section, we discuss some possible future directions for our approaches:

- *Auditorial implementation*: the auditorial implementation is not written as an internal module of SYN. However, we believe that a module able to automatically instrument Sonic Pi, or any other compositional tool, could help bring the audio into the interactive visualization.
- *Computational efficiency*: we built our app within a web environment. Although it worked very well with regular repositories, it was impossible to use it with large repositories such as LibreOffice or Linux. We had to move the render process into a desktop environment losing the interaction feature.
- *Extract more git information*: more git information can be extracted from commits. For example, we can retrieve the author of a commit and map it to a color in the visualization. As a result, we can create a new kind of visualization to understand how many authors contributed to the systems and which part of the system is covered by the development team.
- *New metrics*: the initial set of metrics we defined can be extended to comprehend metrics that require a source code analysis.
- *New positional layouts*: at the moment, we implemented only one positional layout that works with an outward spiral. A future implementation could take care of the system's architecture, displaying additional information about the file's package, for example.
- *Scalable analysis implementation*: Currently, the analysis is meant to work on a single machine. However, the approach does not have this requirement because analysis results are generated independently. We can scale this process horizontally with multiple devices to significantly cut the analysis time.
- *New solution for persistent storage*: at the moment, all the information is stored in JSON files. This means that they need to be deserialized every time we need them. With a database, the access time can be significantly improved.
- *Perform other case studies*: analyzing different software systems allows us to evaluate our methodology on other systems and study the growth model of other open-source systems.

6.3 Epilogue

In this thesis, we have proposed an approach based on synesthesia to ease the comprehension task of evolving software systems with an interactive visual depiction of software artifacts complemented by an auditive portrayal of the evolution. We claim that its main benefit is understanding the structure and the development lifecycle of a git project.
The broad applicability of our approach (on GitHub, there are more than 20M open-source systems) allows us to study the evolution of software systems and, thanks to the gained experience, improve and specialize the techniques presented in this thesis.

Appendix A

Sonic Pi for system evolution auralization

```
commits = [4776, 4883, 7710, 6996, 6, ..., 2]
additions = [18046, 867, 1032, 696, 794, ..., 0]
deletions = [148, 269, 310, 238, 170, ..., 0]

moments = commits.length
print("moments", moments)

bpm_min = 60
bpm_max = 200

bpm = 60

commit_min = commits.min
commit_max = commits.max

add_min = additions.min
add_max = additions.max

del_min = deletions.min
del_max = deletions.max

measure = 60.0 / bpm

tick_reset(:my_tick)

########################################
### Auxiliary functions ###############
########################################
define :rescale do | value, min, max, newMin, newMax |
  result = (value-min+0.0)/(max-min+0.0)*(newMax-newMin+0.0)+newMin
  return result
end

########################################
### Continuously play the beat ########
########################################
in_thread do
  loop do
```

```
    sample :bd_sone, amp: 1
    sleep 60.0 / bpm
  end
end

in_thread do
  moments.times do

    index = tick(:my_tick)

    ##########################################################
    ### Update the BPM with the current value of commits ###
    ##########################################################
    bpm = rescale commits[index], commit_min, commit_max, bpm_min, bpm_max

    ########################################
    ### Play additions ###################
    ########################################
    addTempo = rescale additions[index], add_min, add_max, 1, 8
    addAmp = rescale additions[index], add_min, add_max, 0.2, 0.85
    in_thread do
      addTempo.times do
        sample :elec_blip2, amp: addAmp
        sleep measure / 8
      end
    end

    ########################################
    ### Play deletions ###################
    ########################################
    delTempo = rescale deletions[index], del_min, del_max, 1, 4
    delAmp = rescale deletions[index], del_min, del_max, 0.1, 0.8
    in_thread do
      delTempo.times do
        sample :bass_trance_c, amp: delAmp
        sleep measure / 4
      end
    end
    sleep measure
  end
end
```

Appendix B

Evolution of ArgoUML

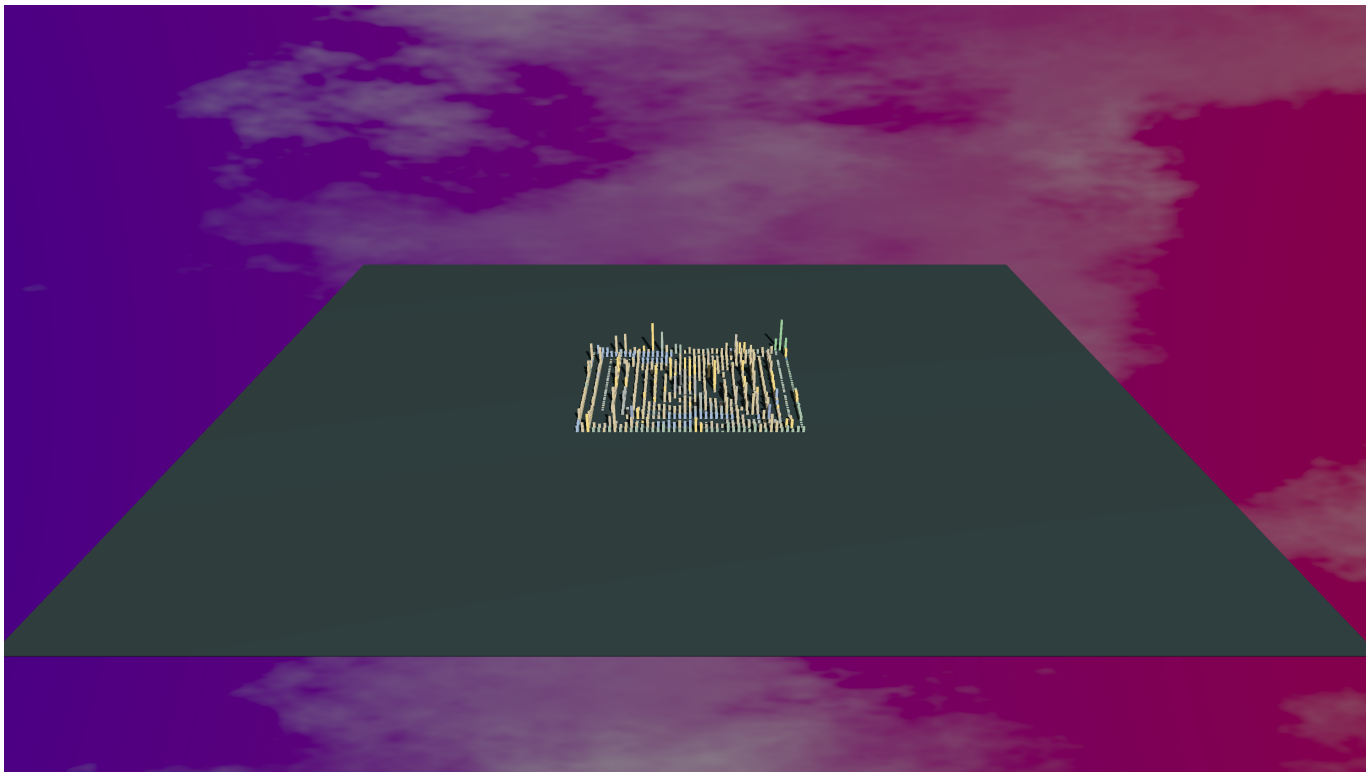


FIGURE B.1: ArgoUML in January 1999

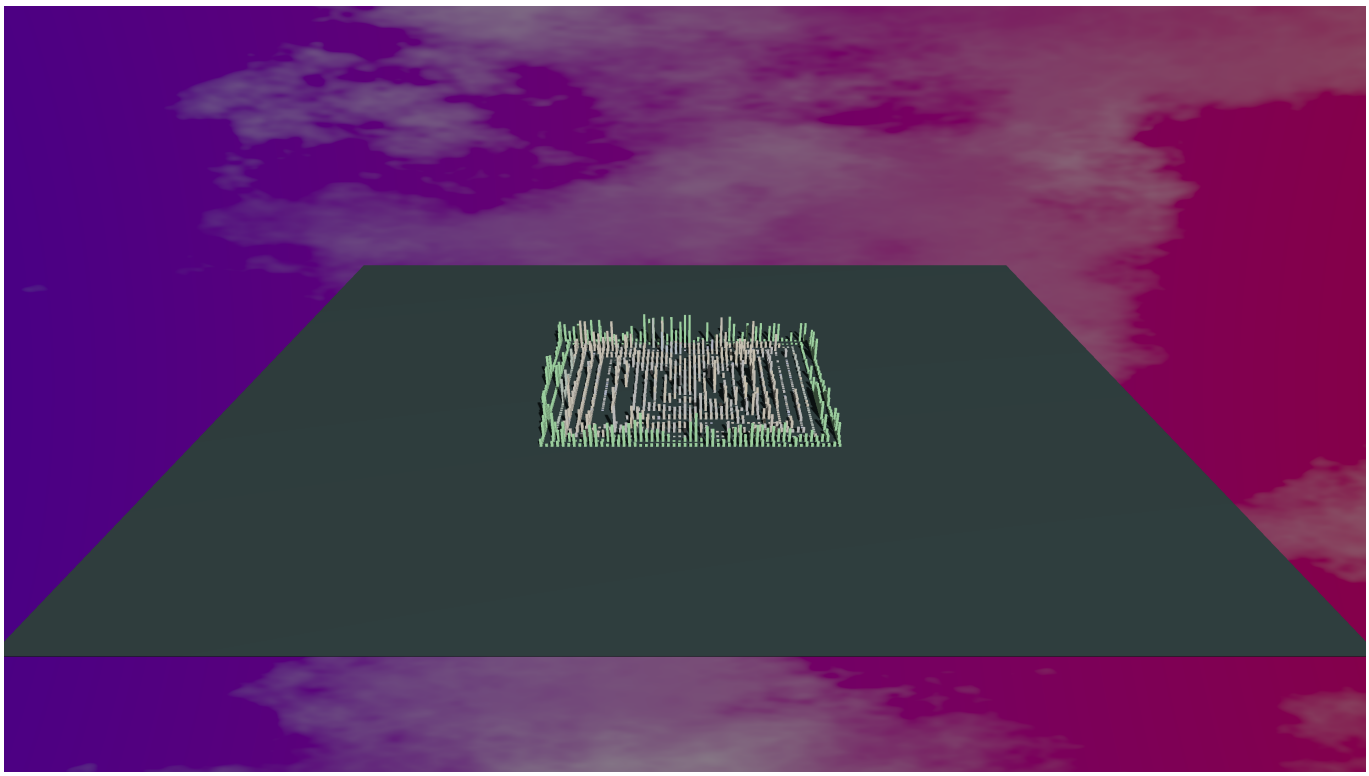


FIGURE B.2: ArgoUML in January 2000

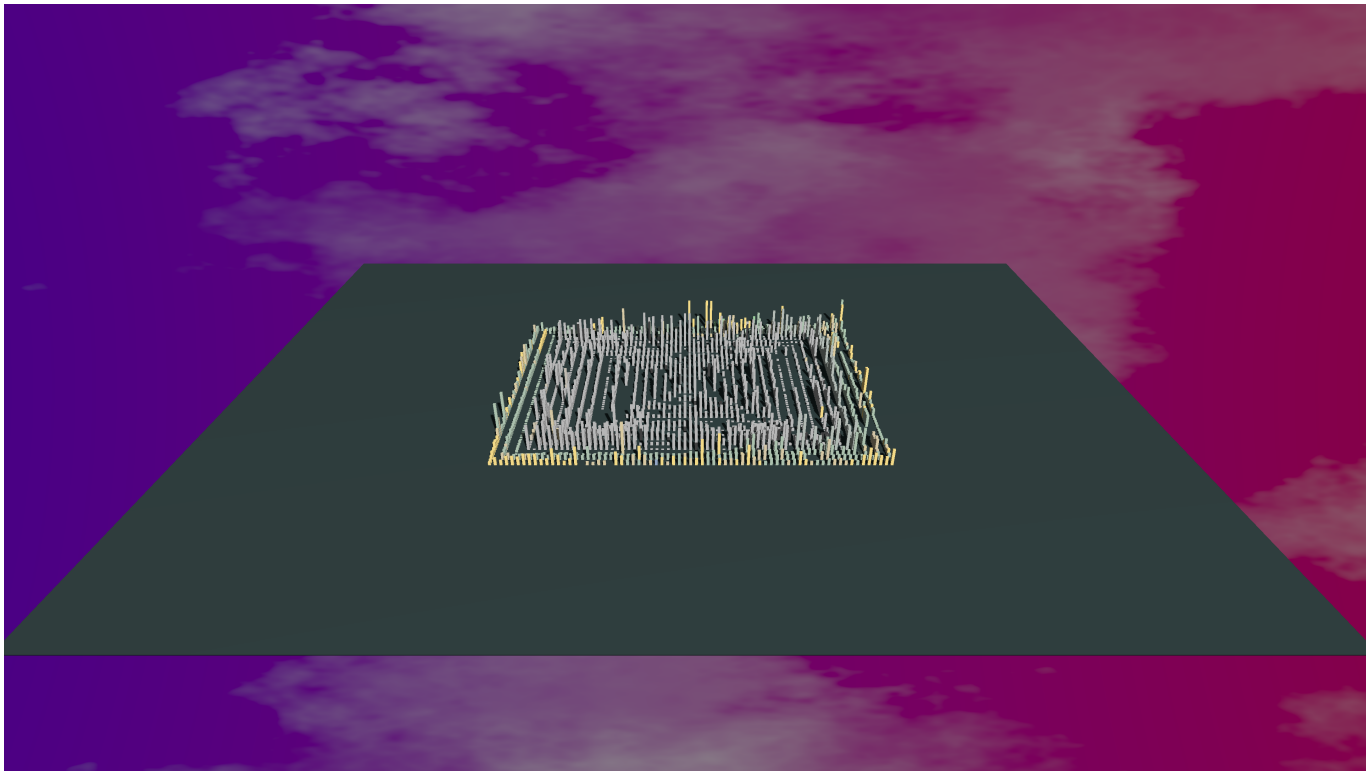


FIGURE B.3: ArgoUML in January 2001

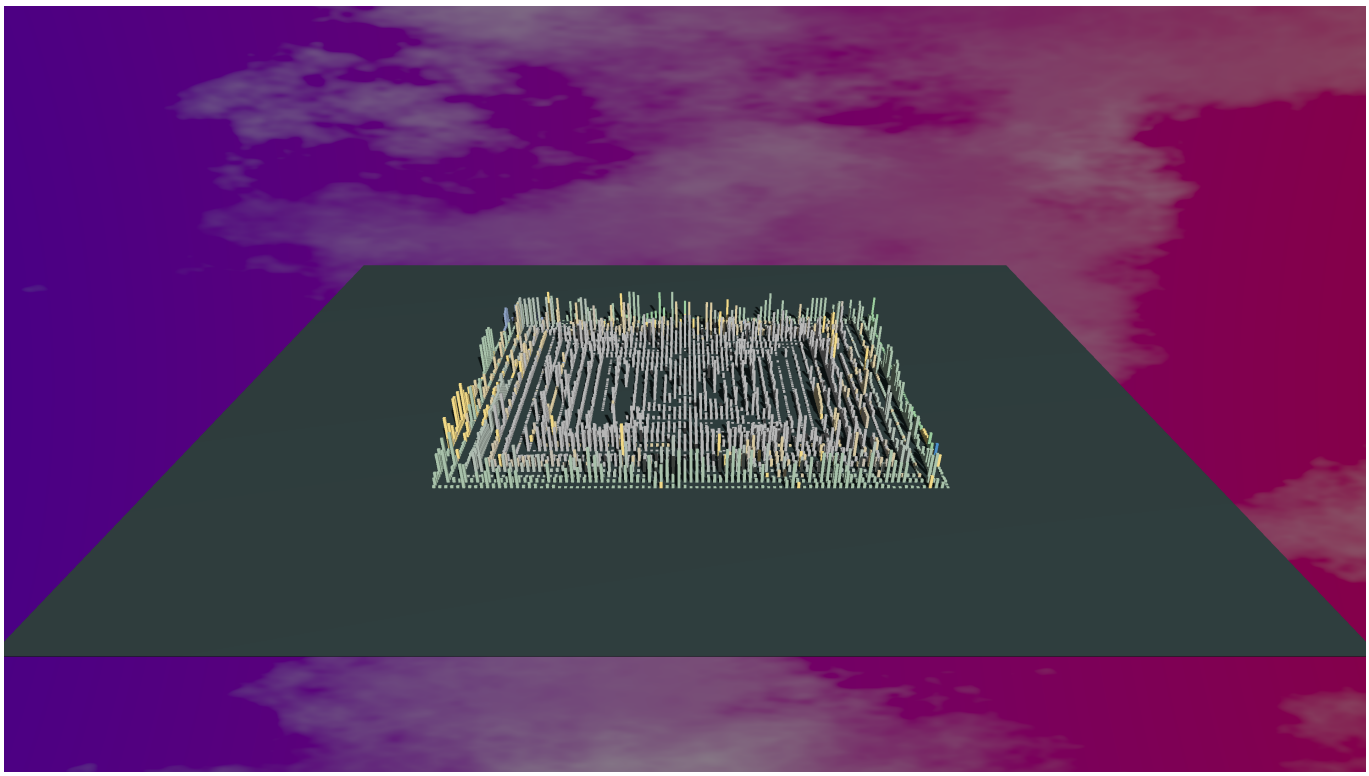


FIGURE B.4: ArgoUML in January 2002

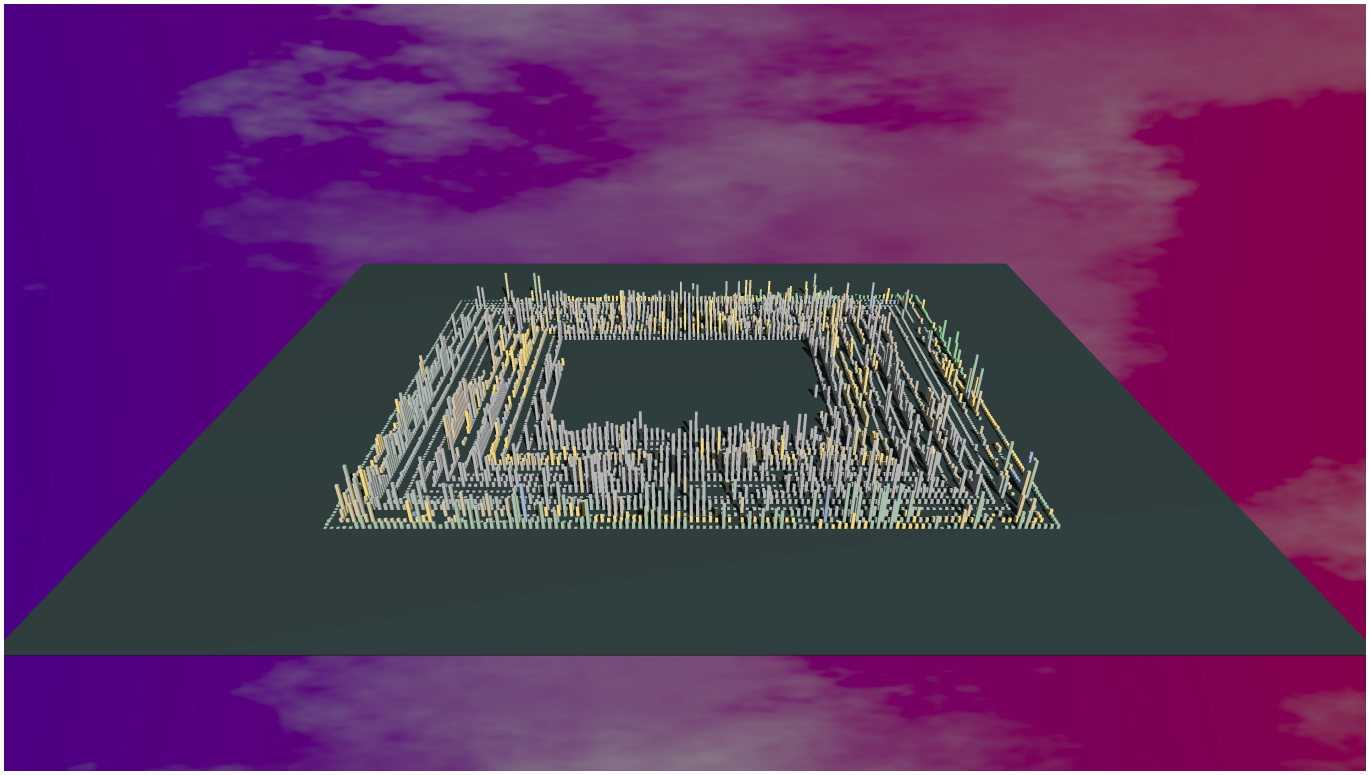


FIGURE B.5: ArgoUML in January 2003

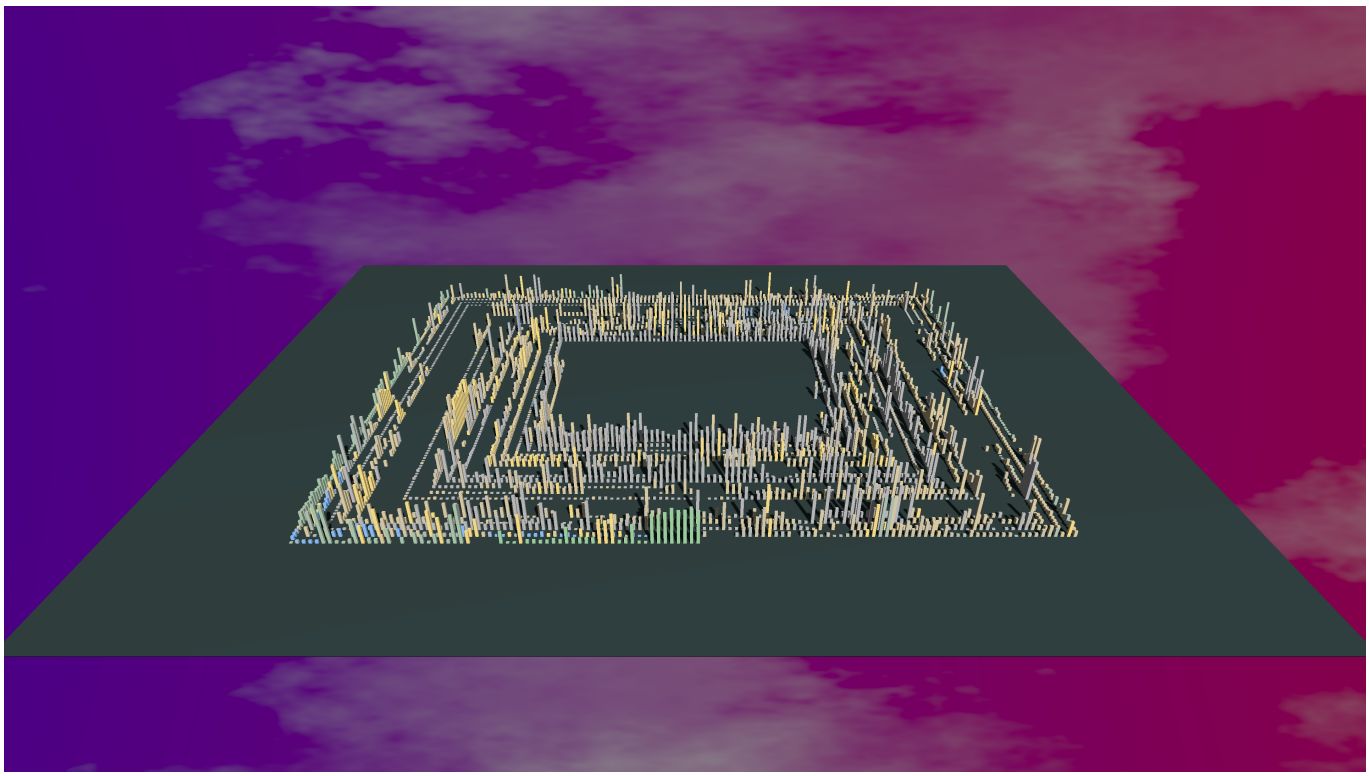


FIGURE B.6: ArgoUML in January 2004

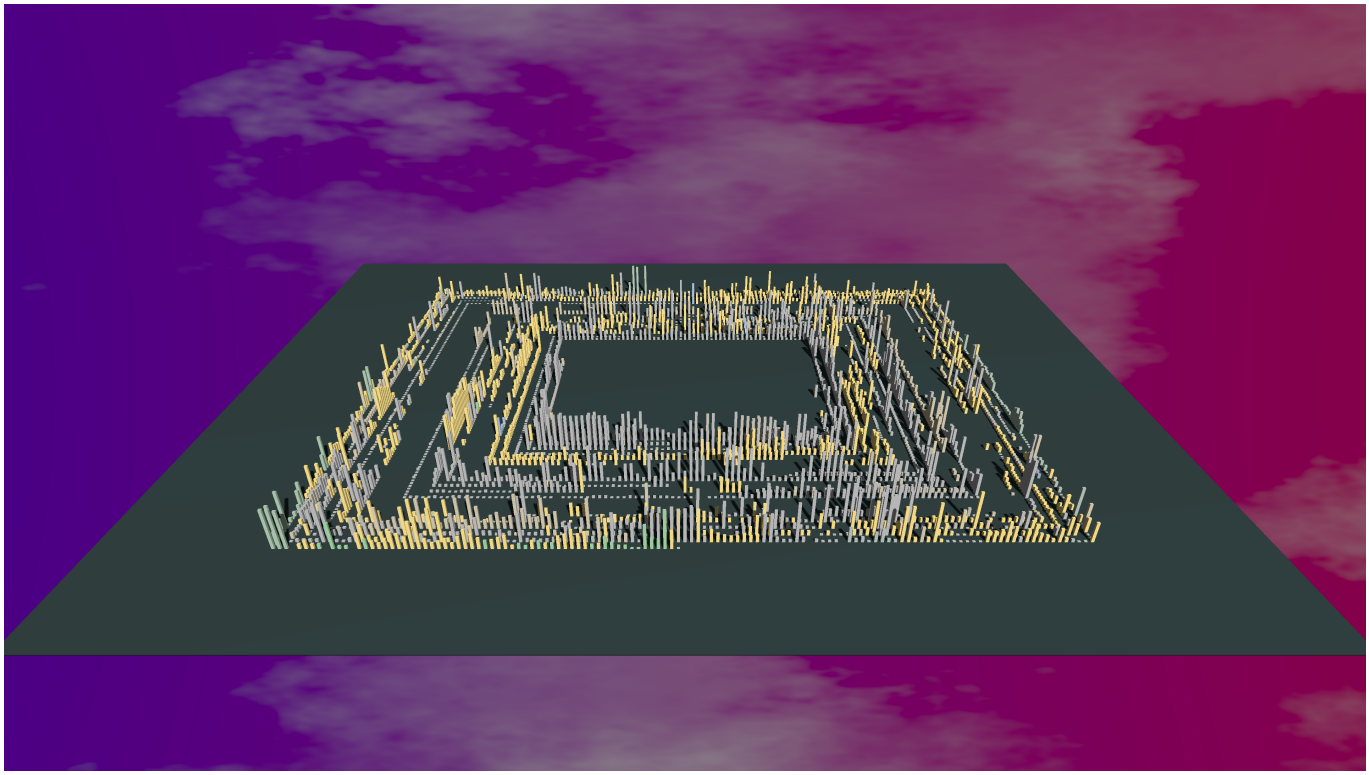


FIGURE B.7: ArgoUML in January 2005

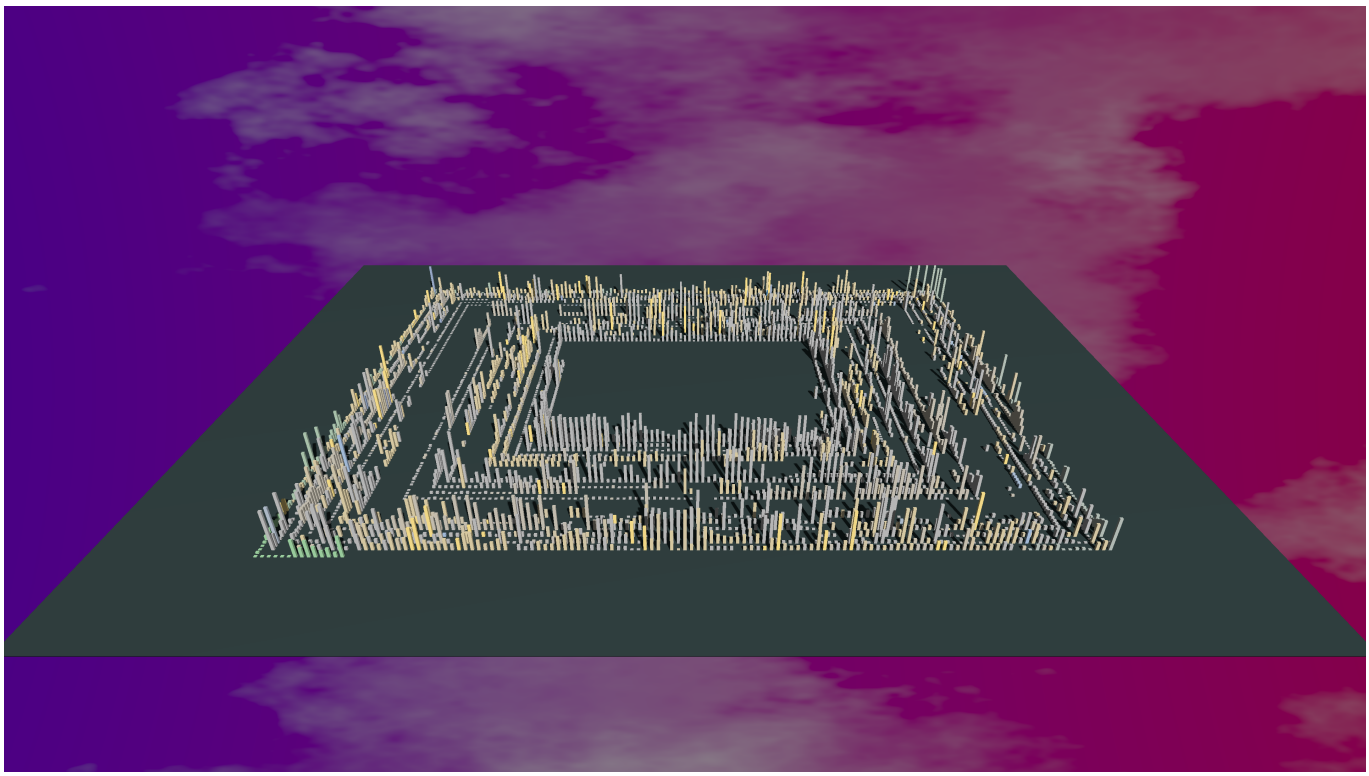


FIGURE B.8: ArgoUML in January 2006

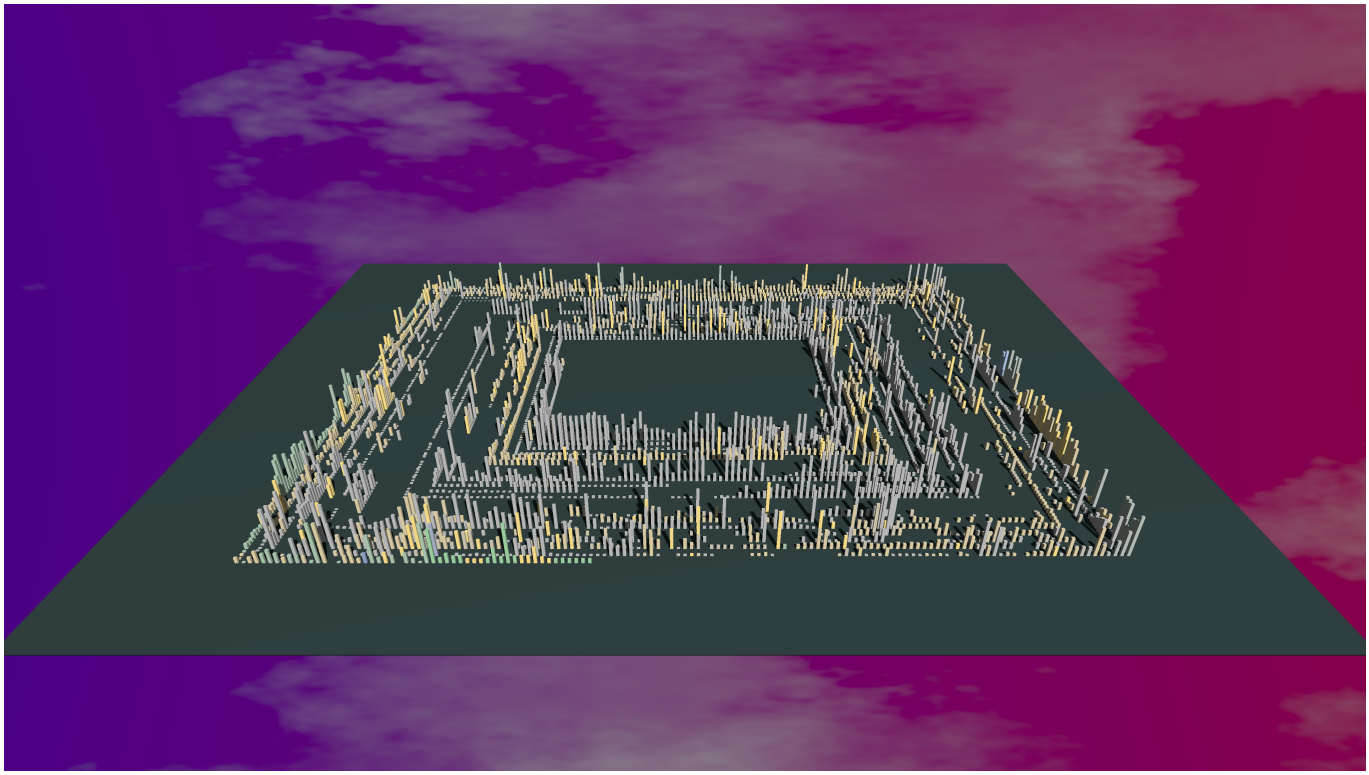


FIGURE B.9: ArgoUML in January 2007

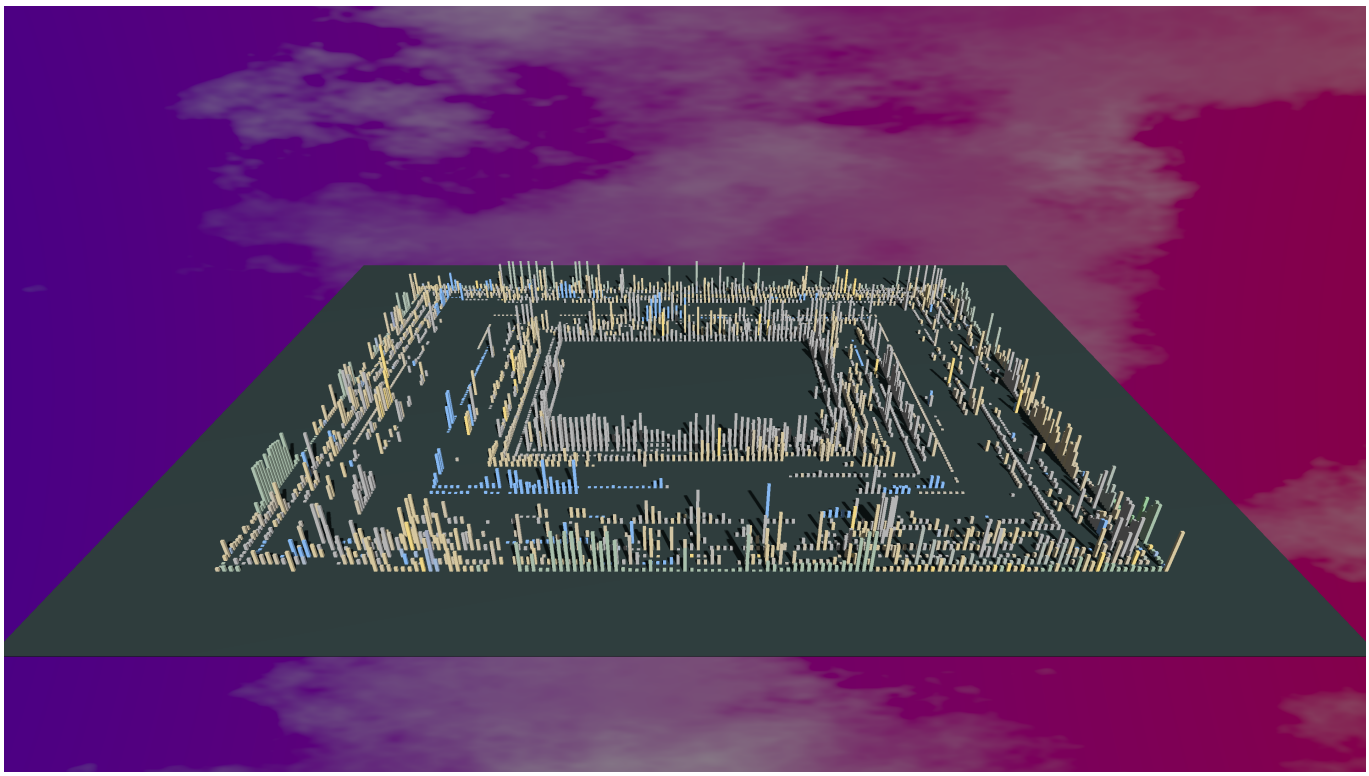


FIGURE B.10: ArgoUML in January 2008

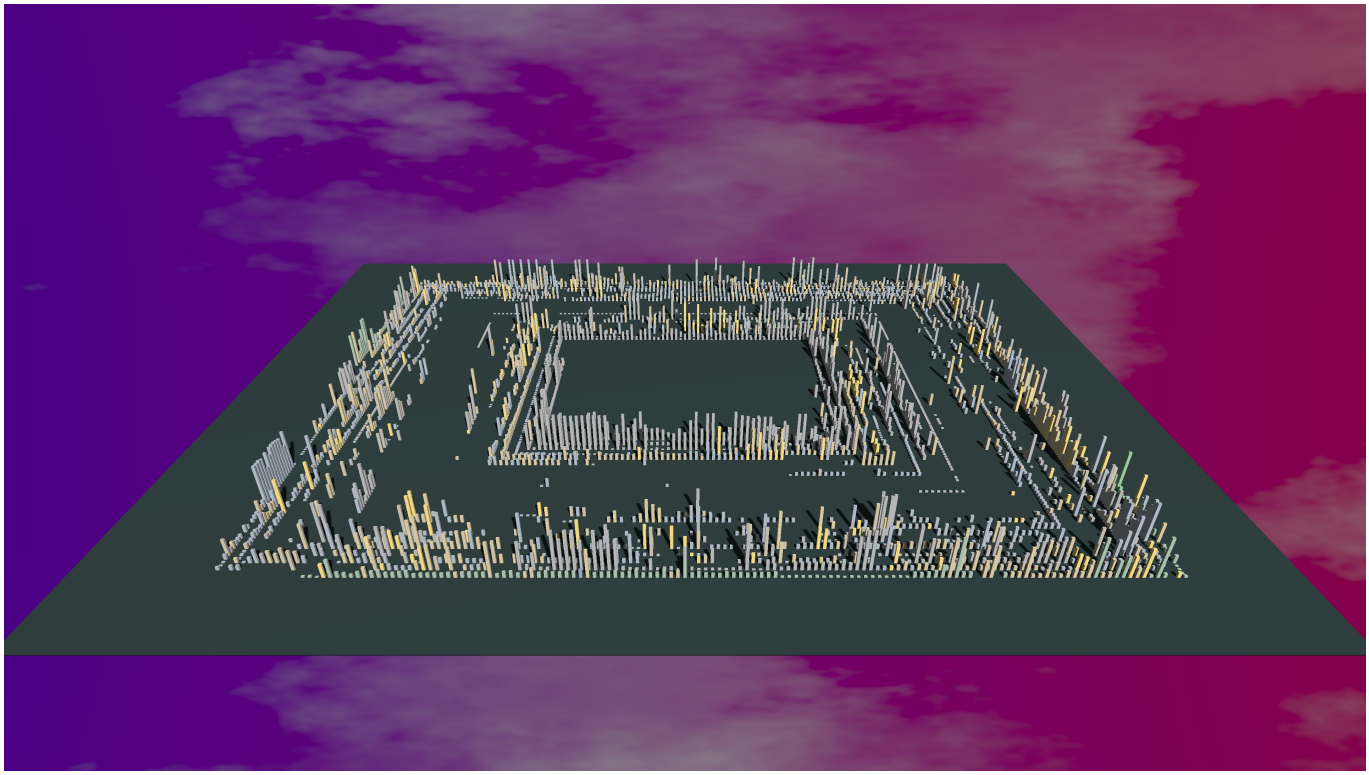


FIGURE B.11: ArgoUML in January 2009

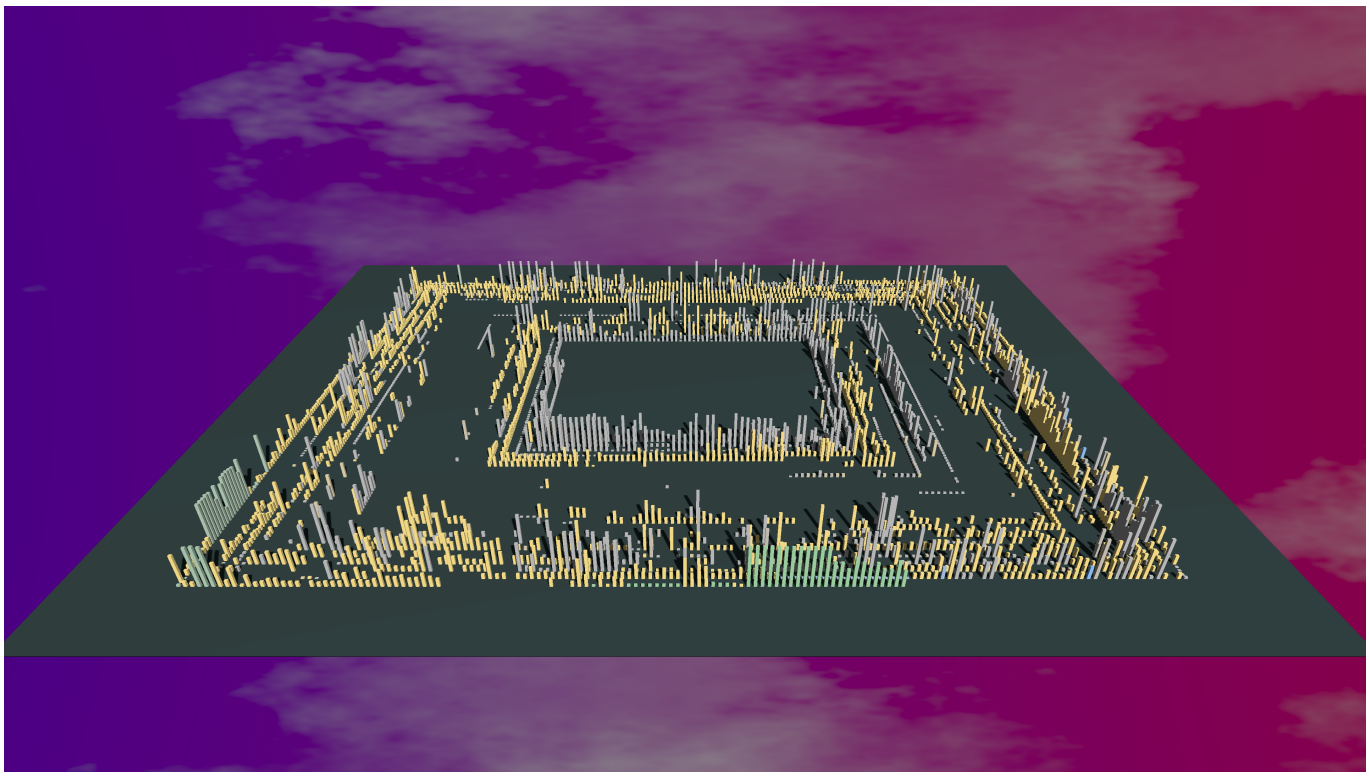


FIGURE B.12: ArgoUML in January 2010

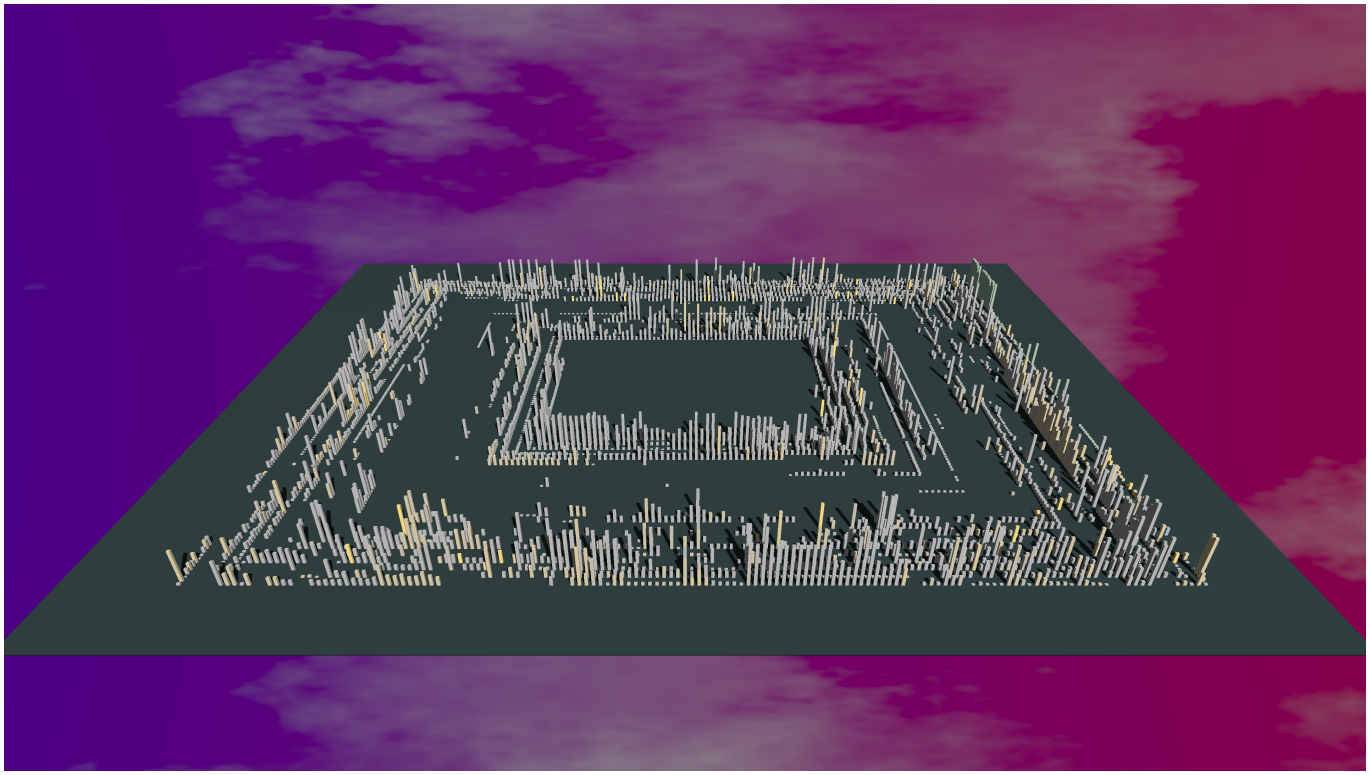


FIGURE B.13: ArgoUML in January 2011

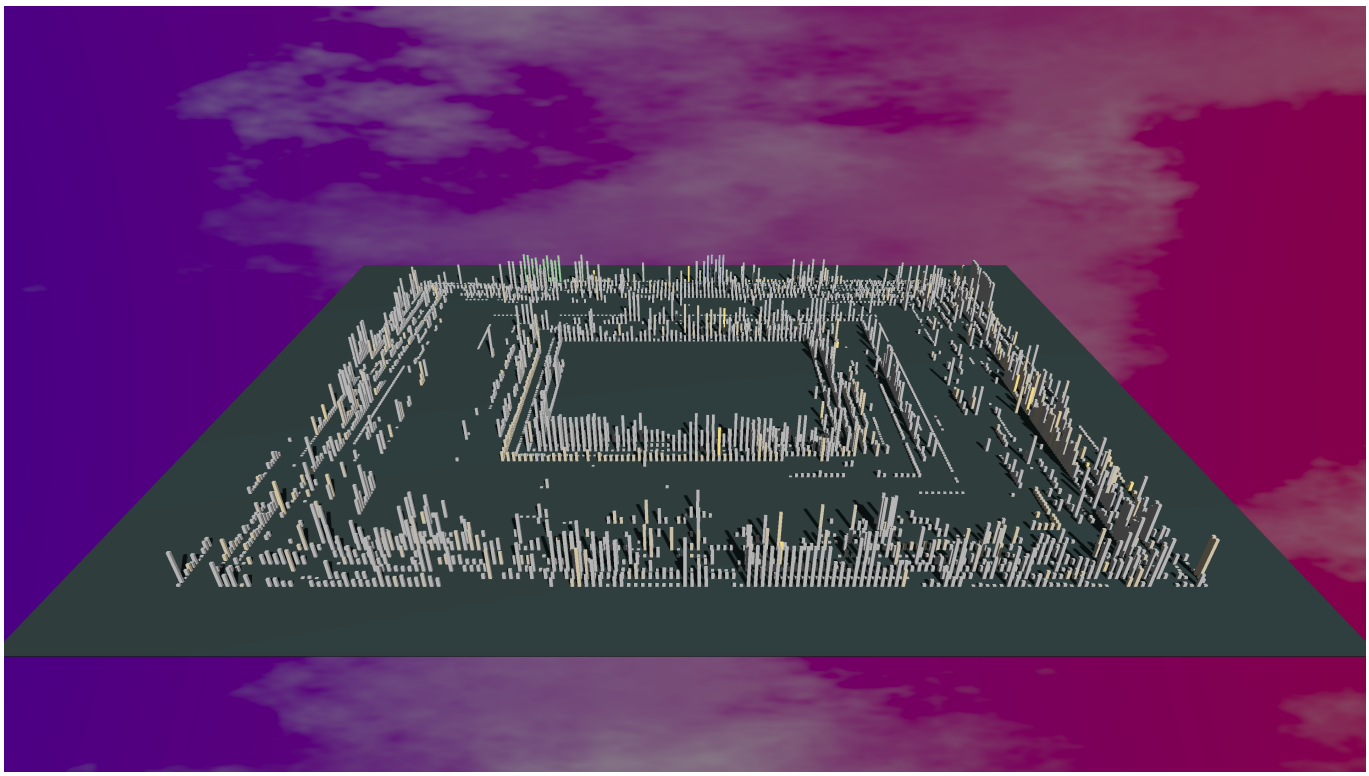


FIGURE B.14: ArgoUML in January 2012

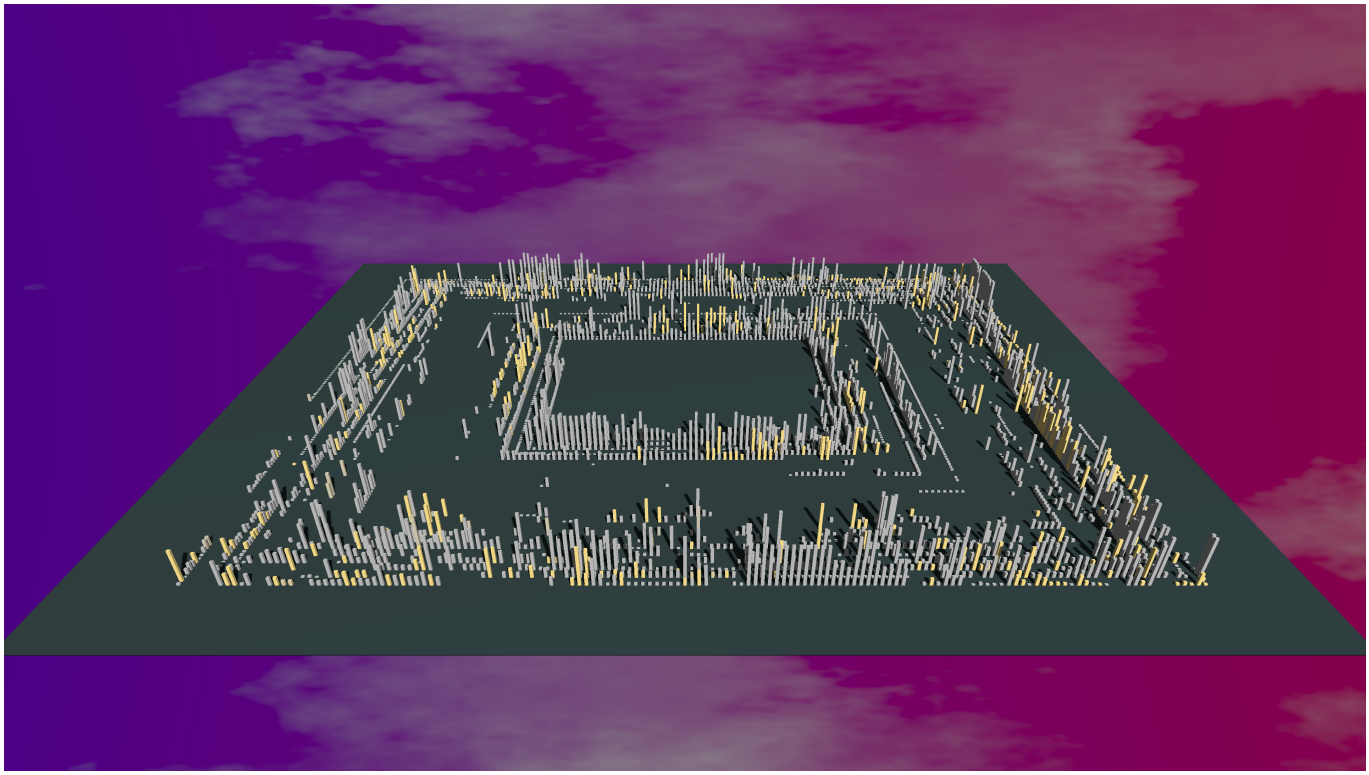


FIGURE B.15: ArgoUML in January 2013

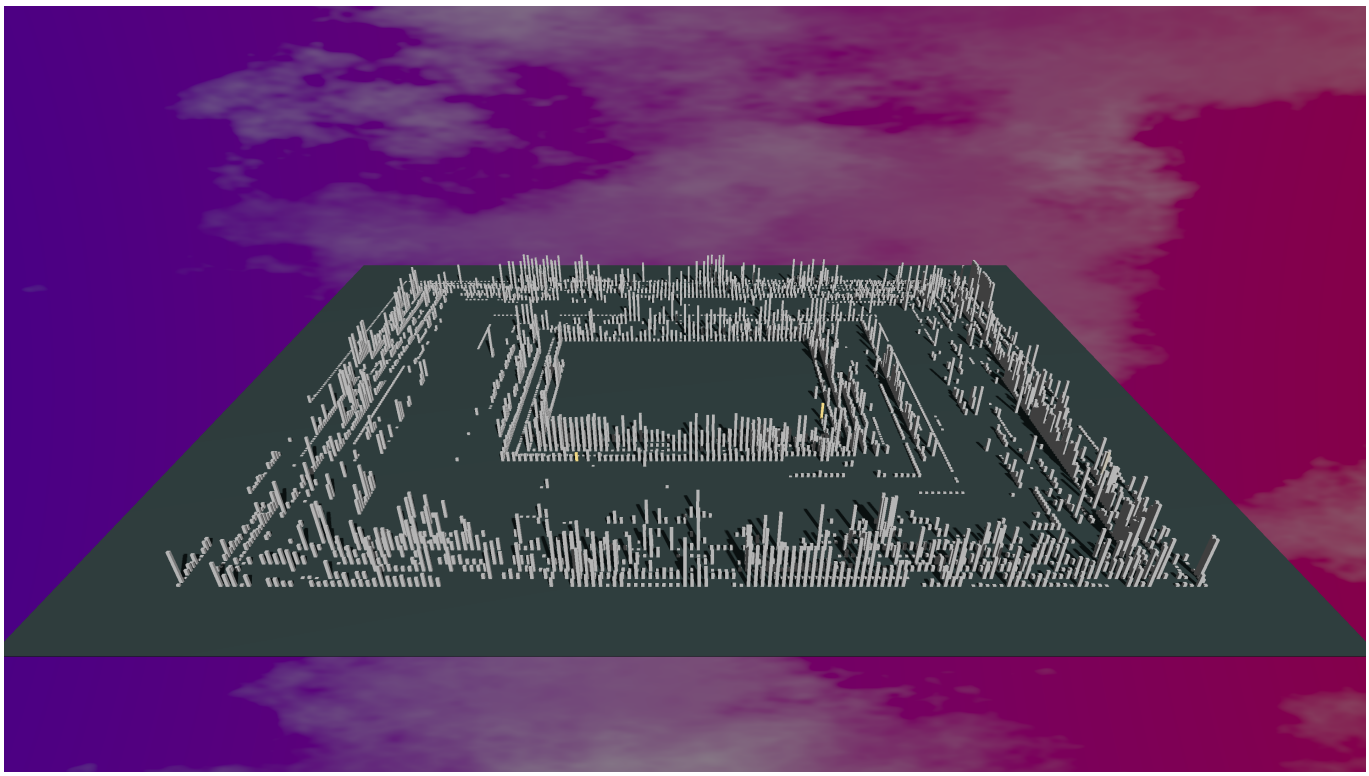


FIGURE B.16: ArgoUML in January 2014

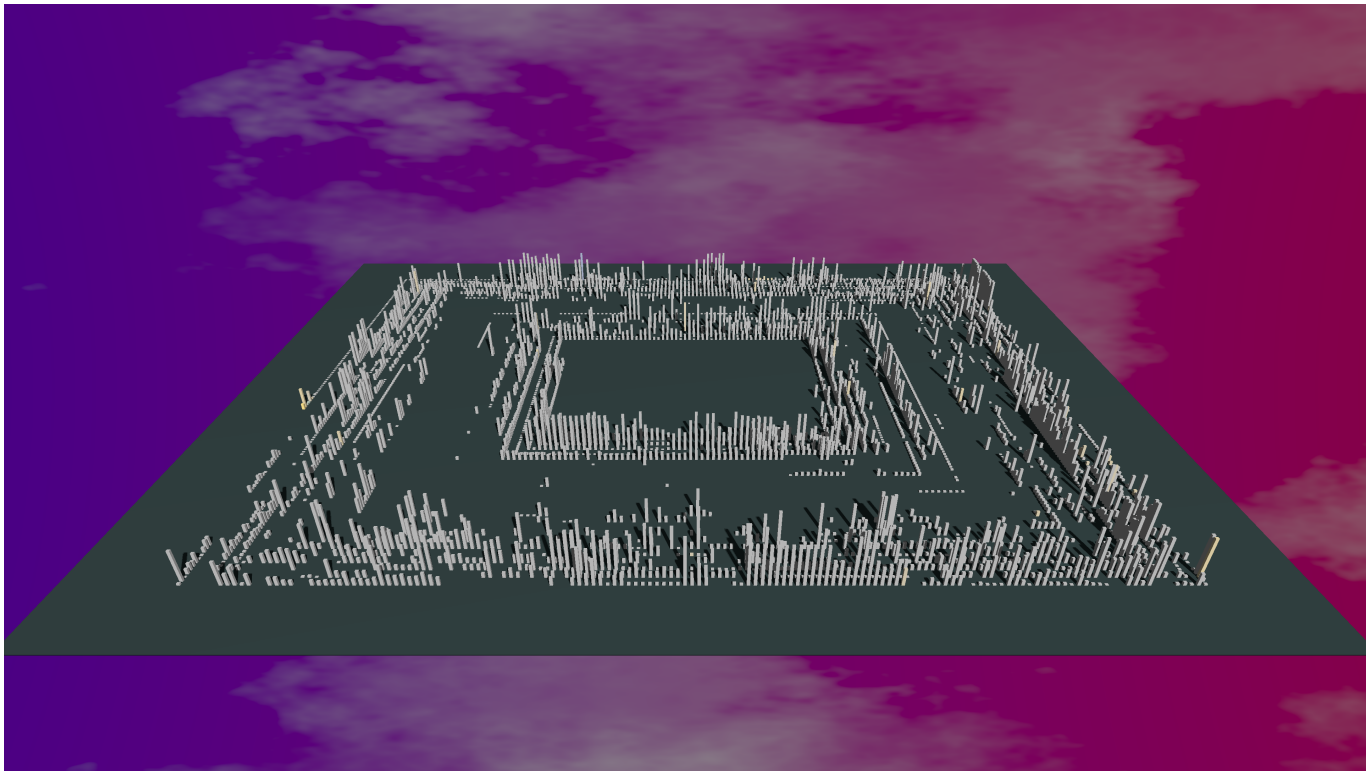


FIGURE B.17: ArgoUML in January 2015

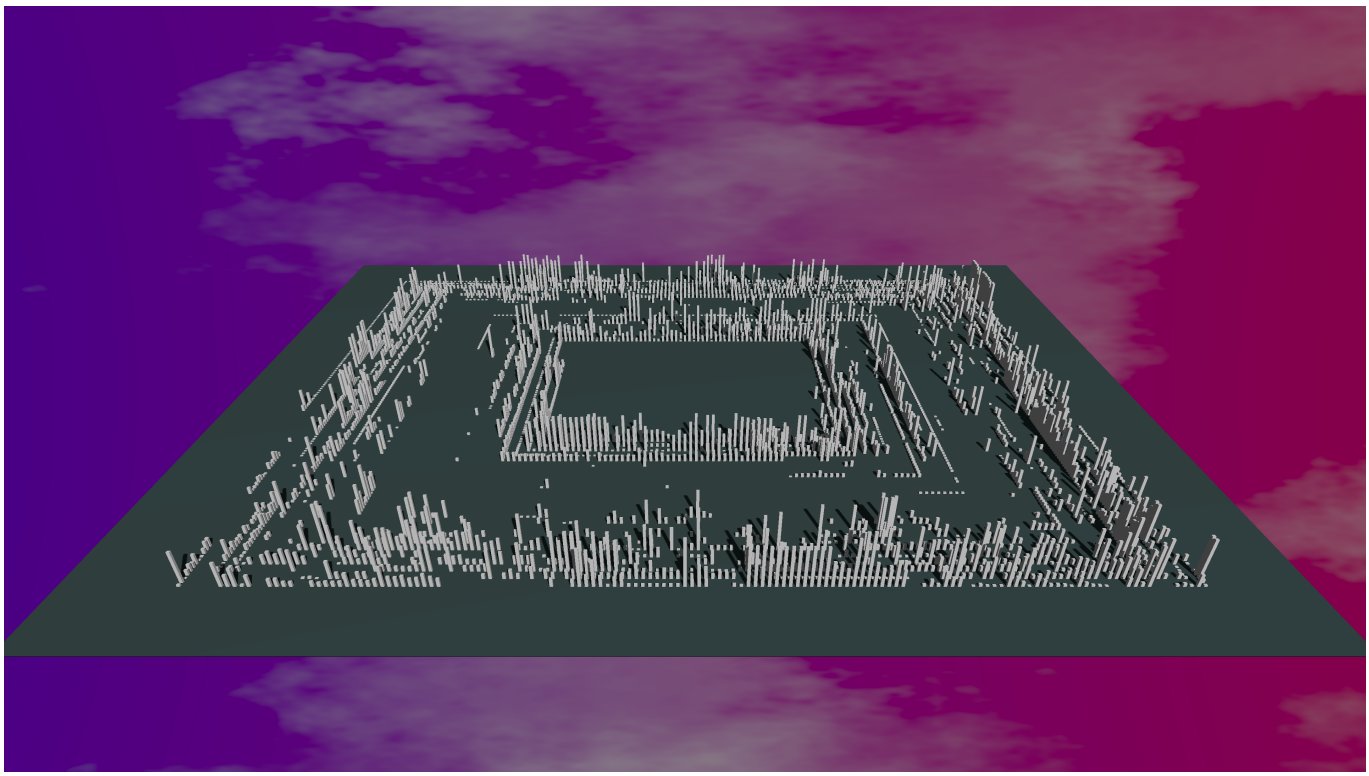


FIGURE B.18: ArgoUML in January 2016

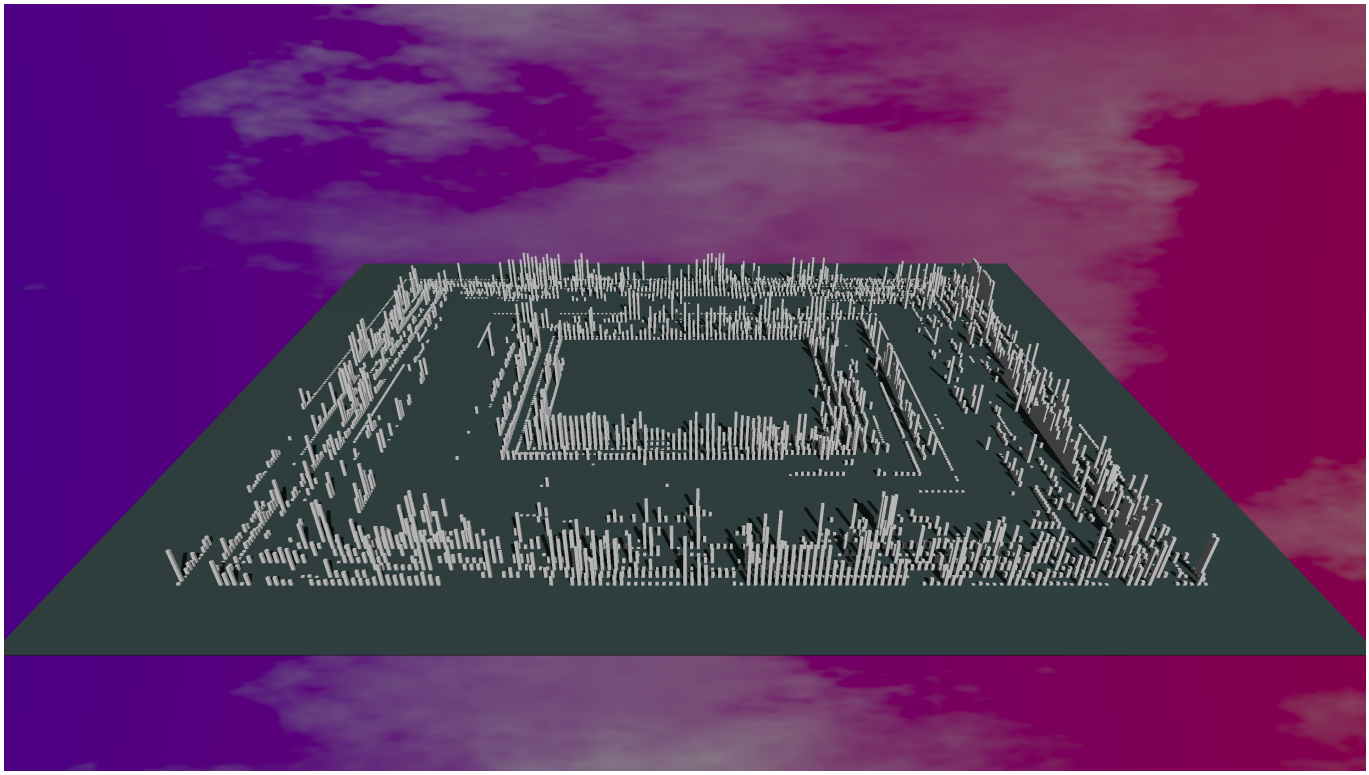


FIGURE B.19: ArgoUML in January 2017

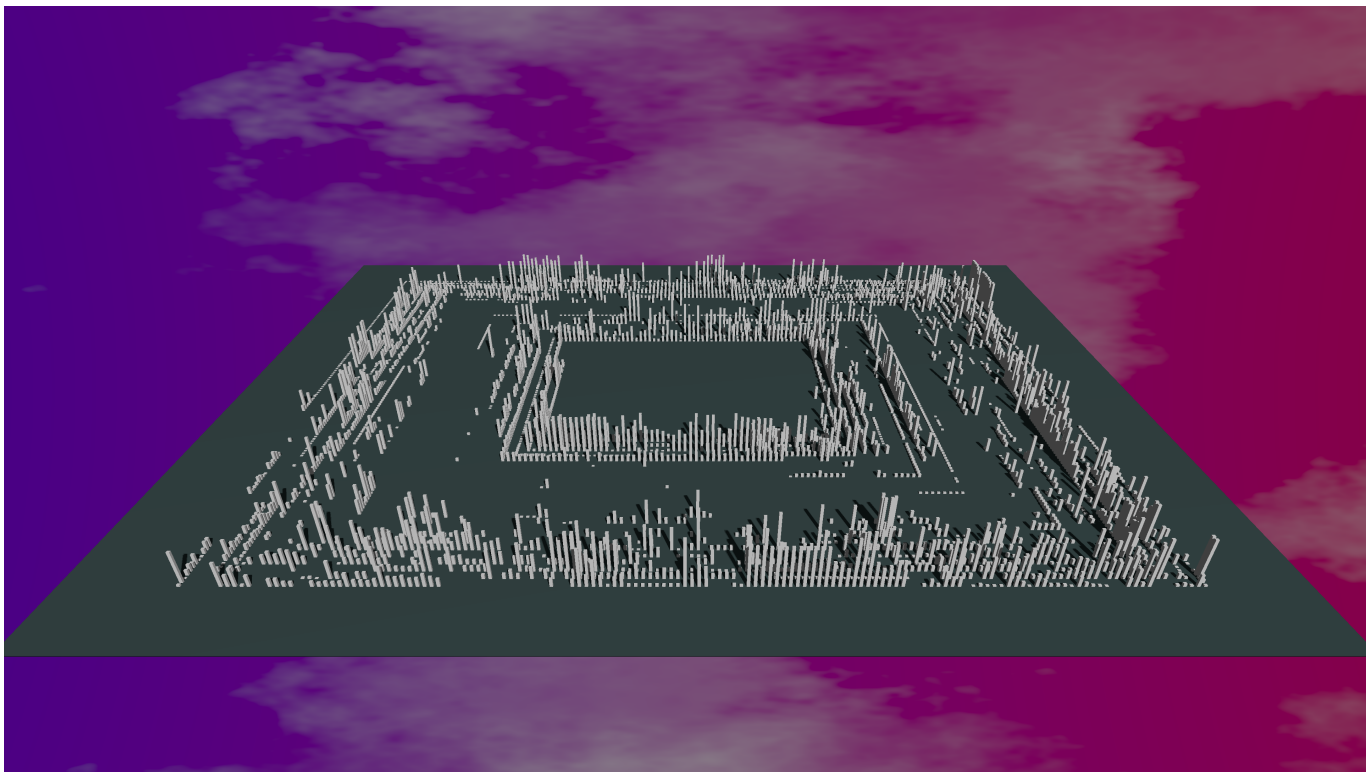


FIGURE B.20: ArgoUML in January 2018

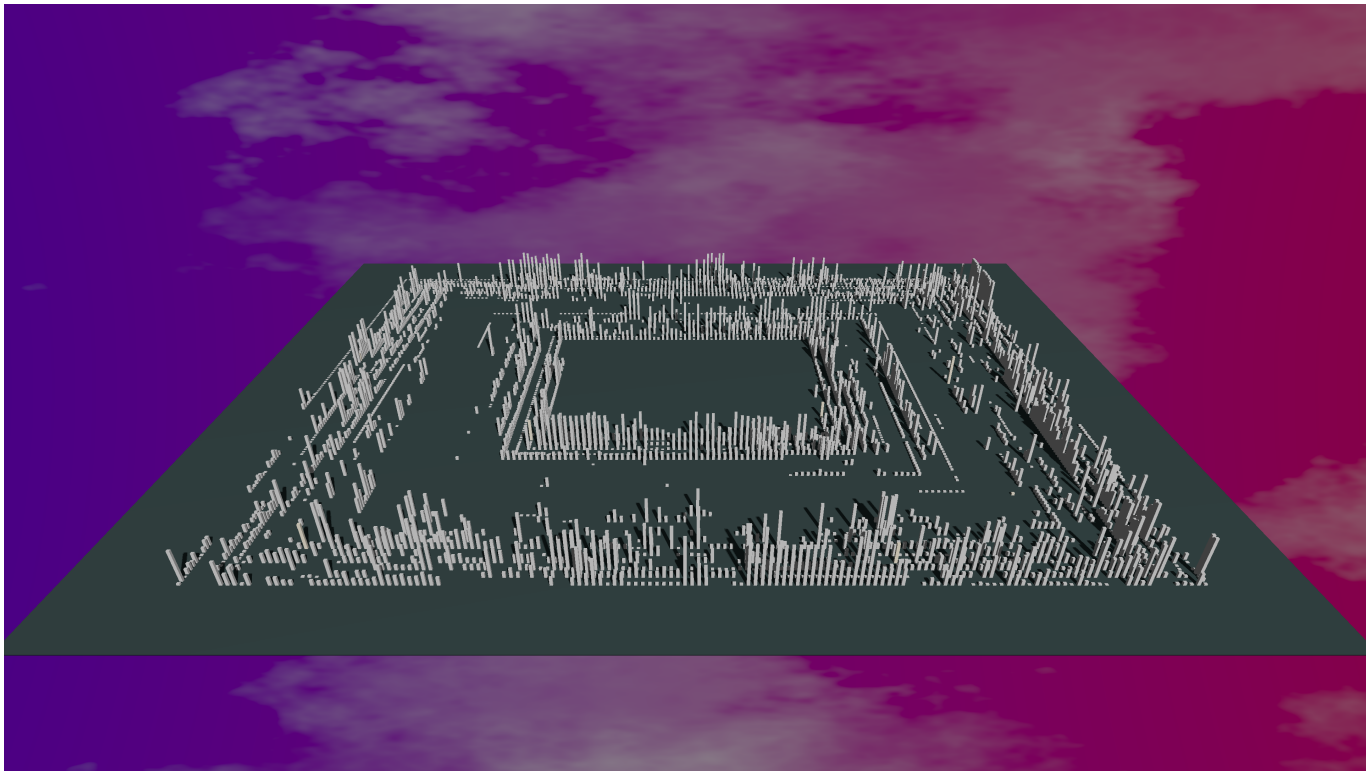


FIGURE B.21: ArgoUML in January 2019

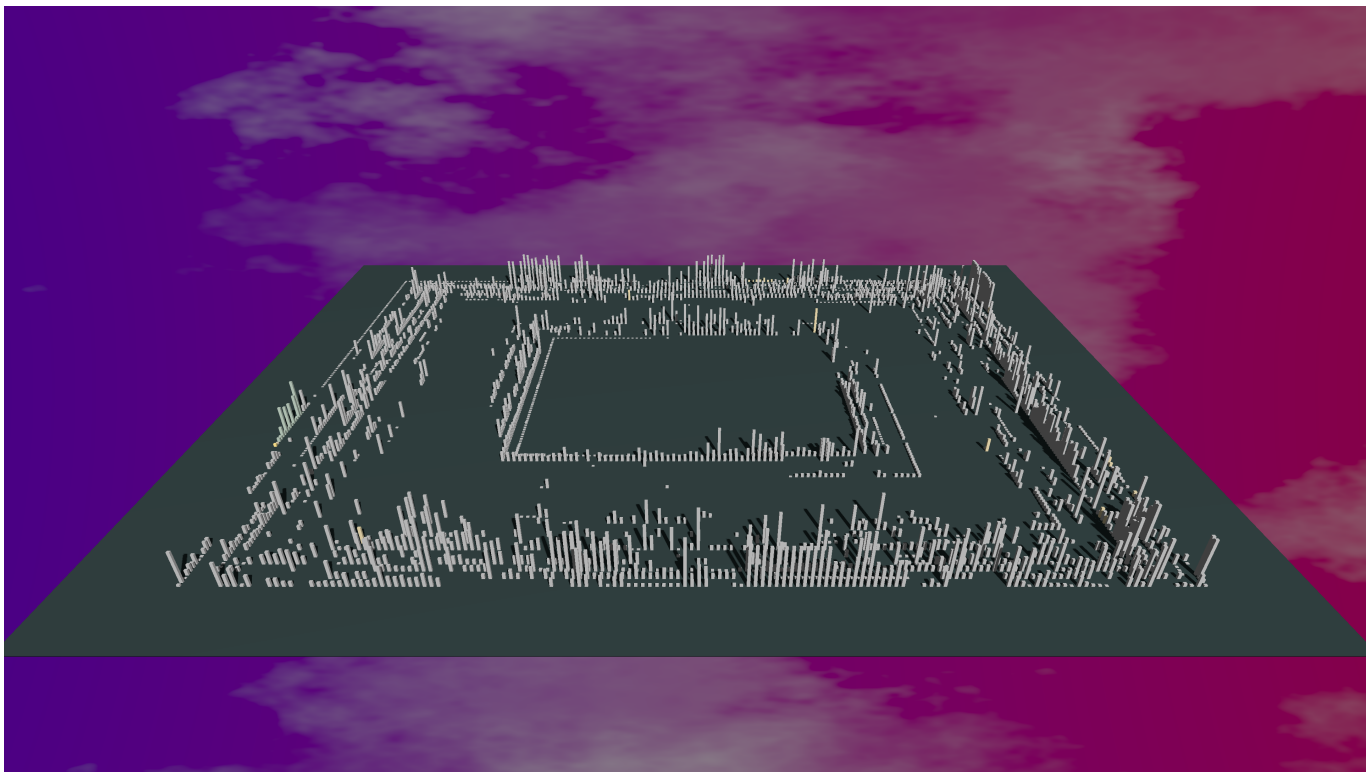


FIGURE B.22: ArgoUML in January 2020

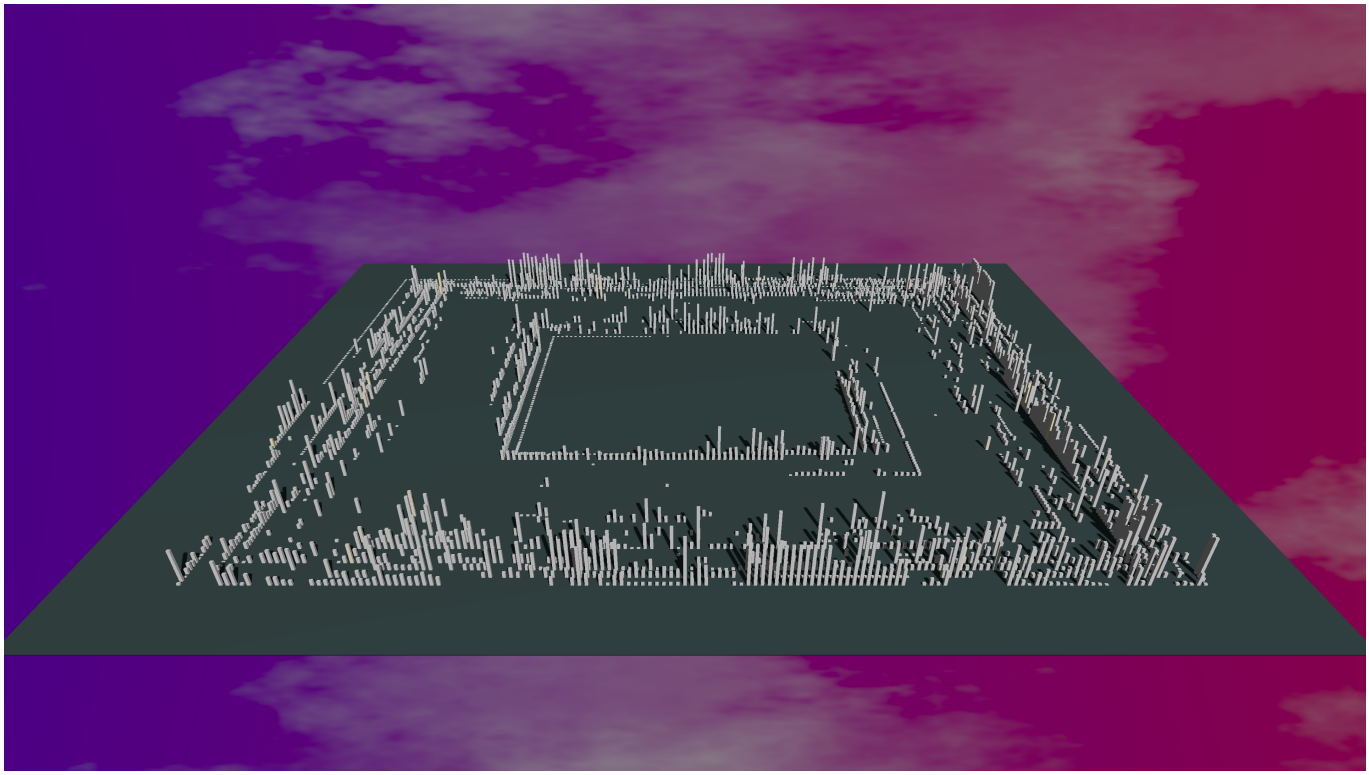


FIGURE B.23: ArgoUML in January 2021

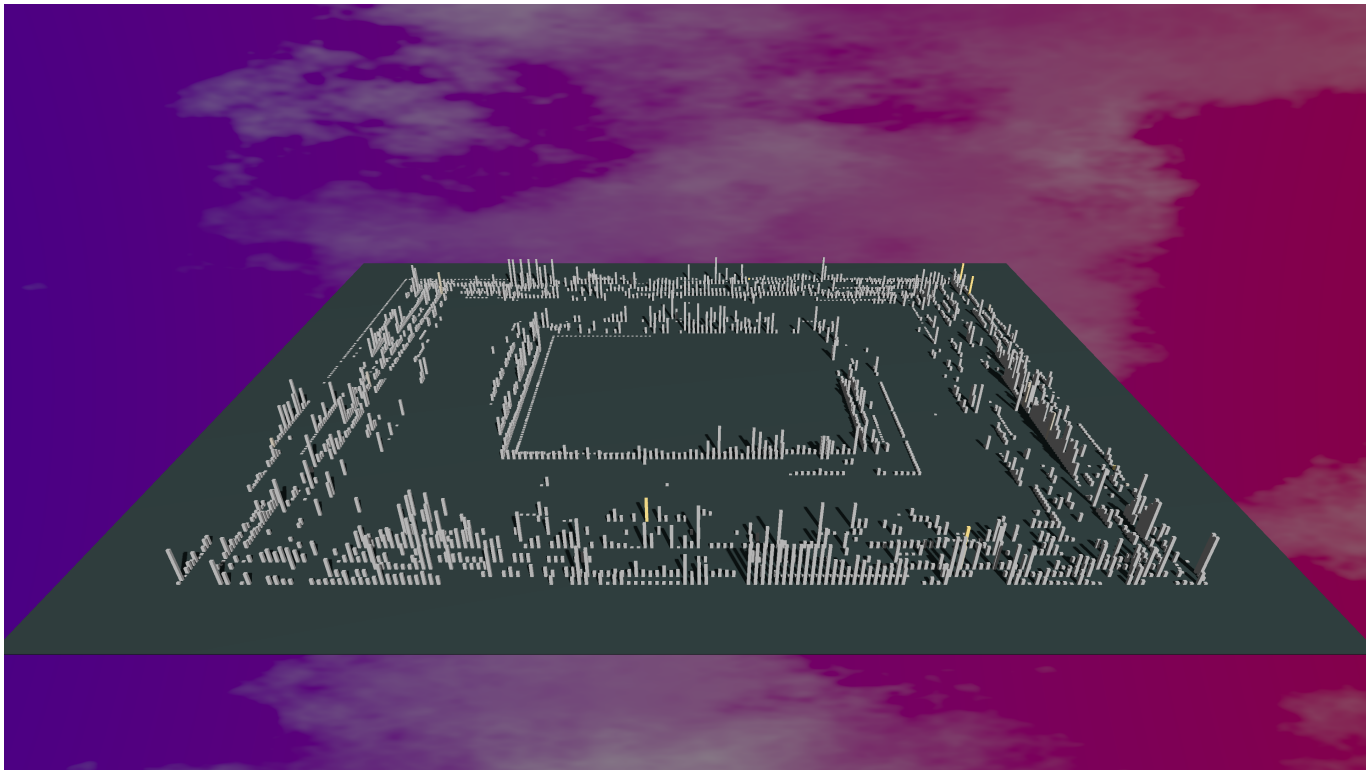


FIGURE B.24: ArgoUML in January 2022

Appendix C

Evolution of Elasticsearch

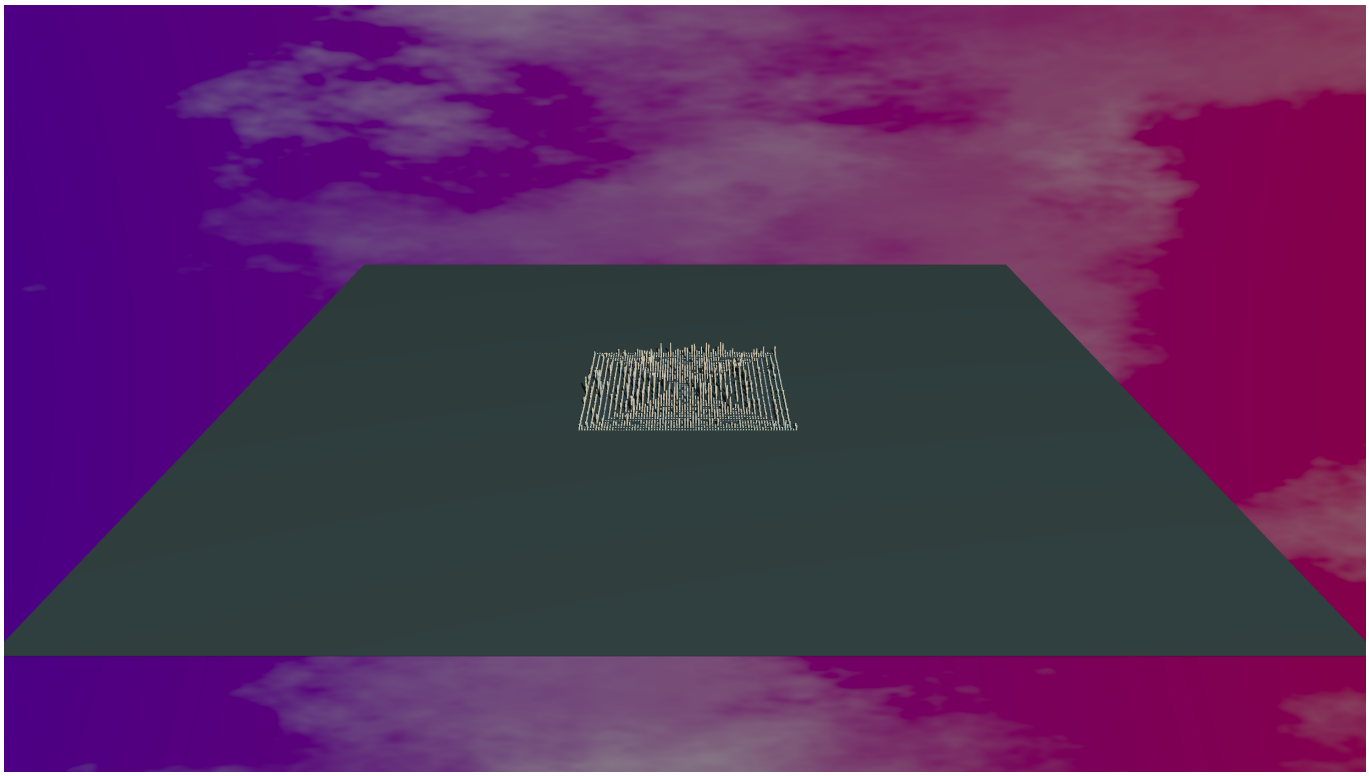


FIGURE C.1: Elasticsearch in June 2014

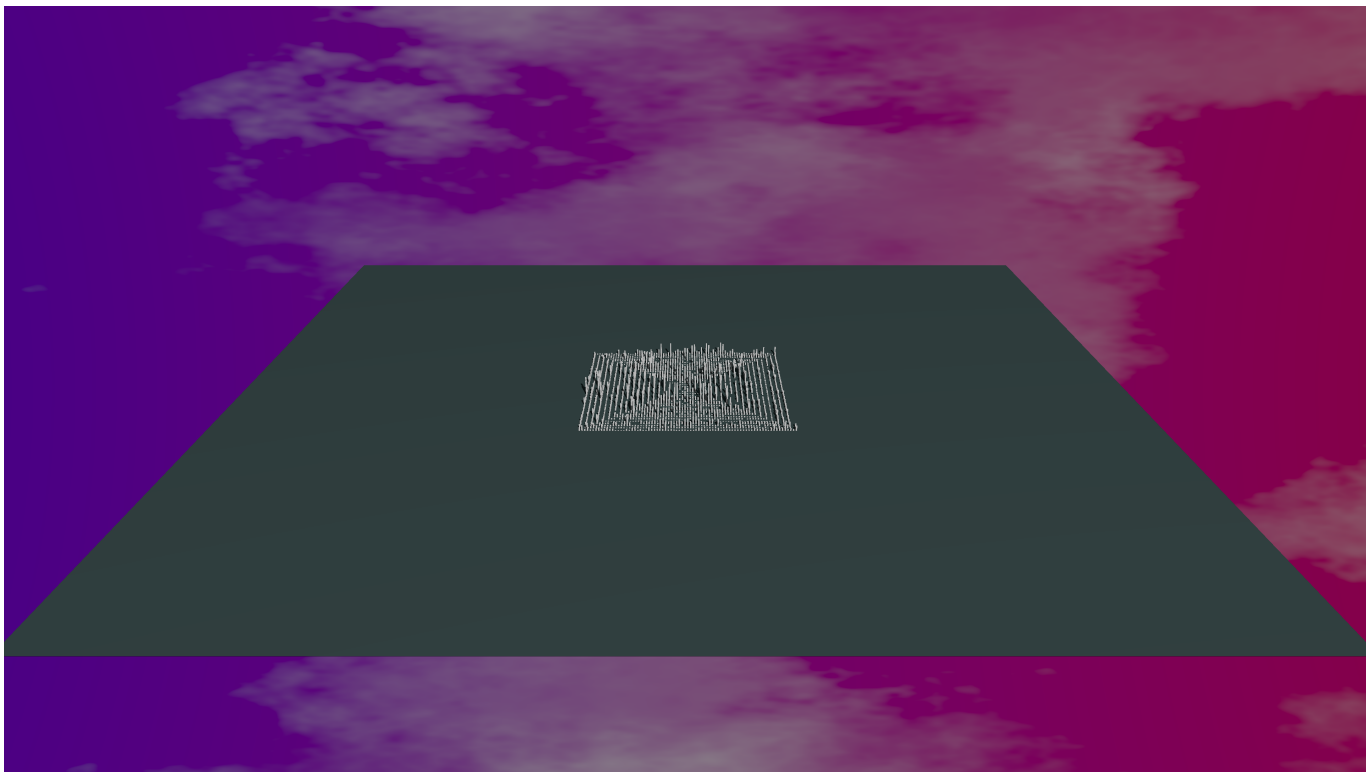


FIGURE C.2: Elasticsearch in June 2015

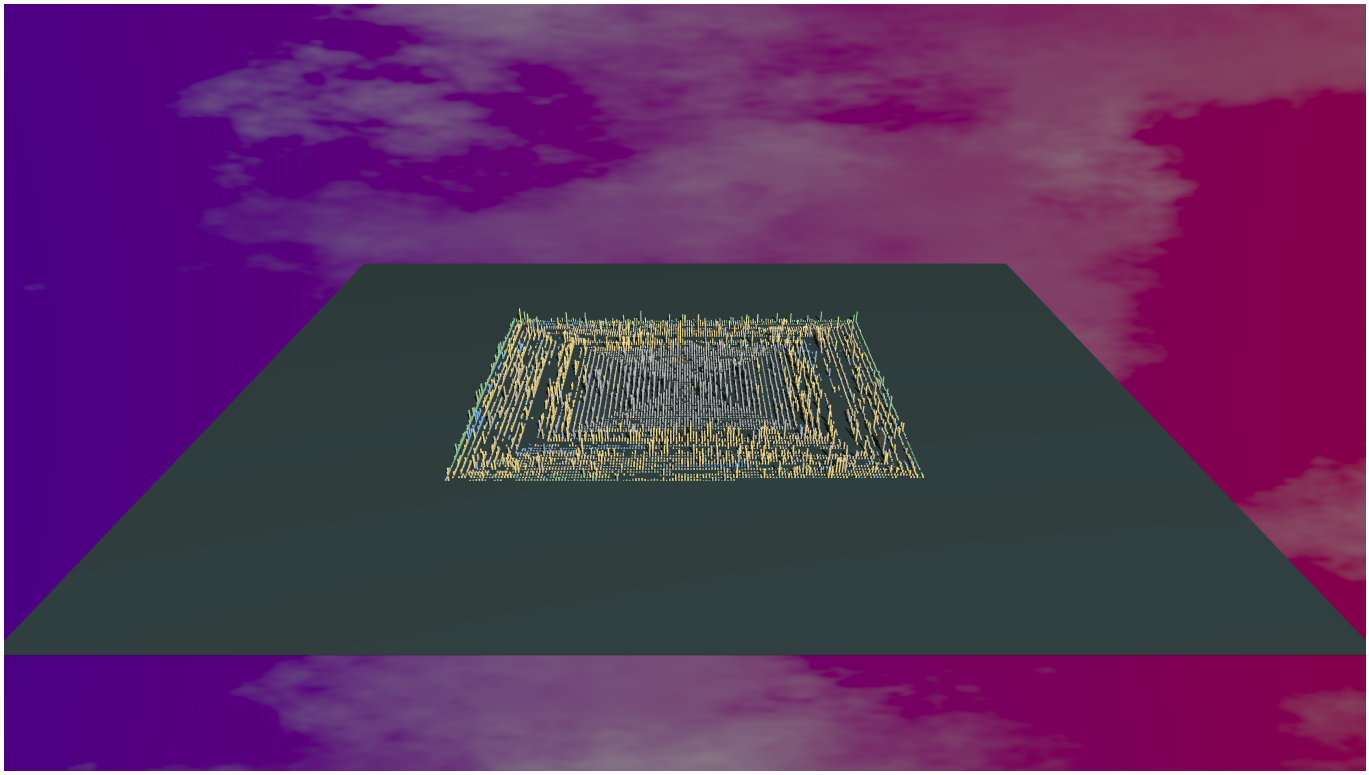


FIGURE C.3: Elasticsearch in June 2016

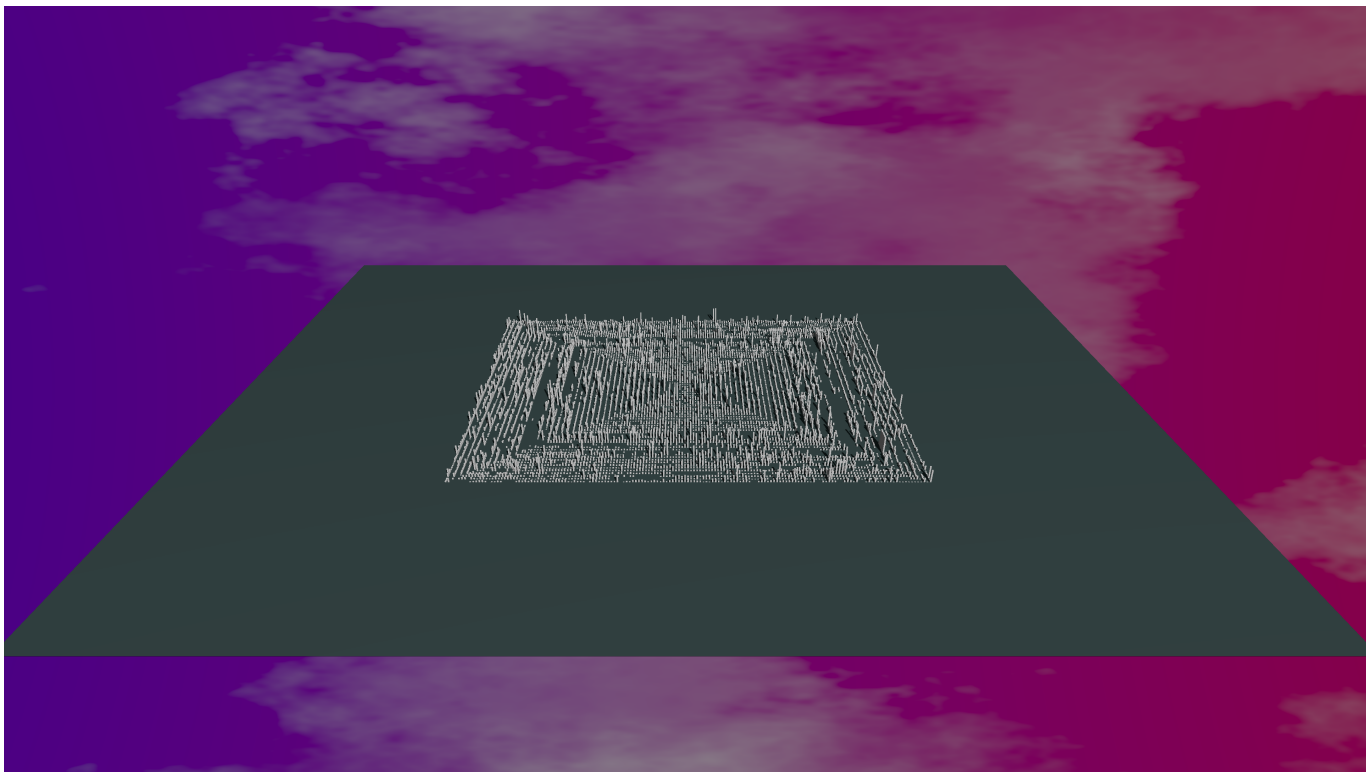


FIGURE C.4: Elasticsearch in June 2017

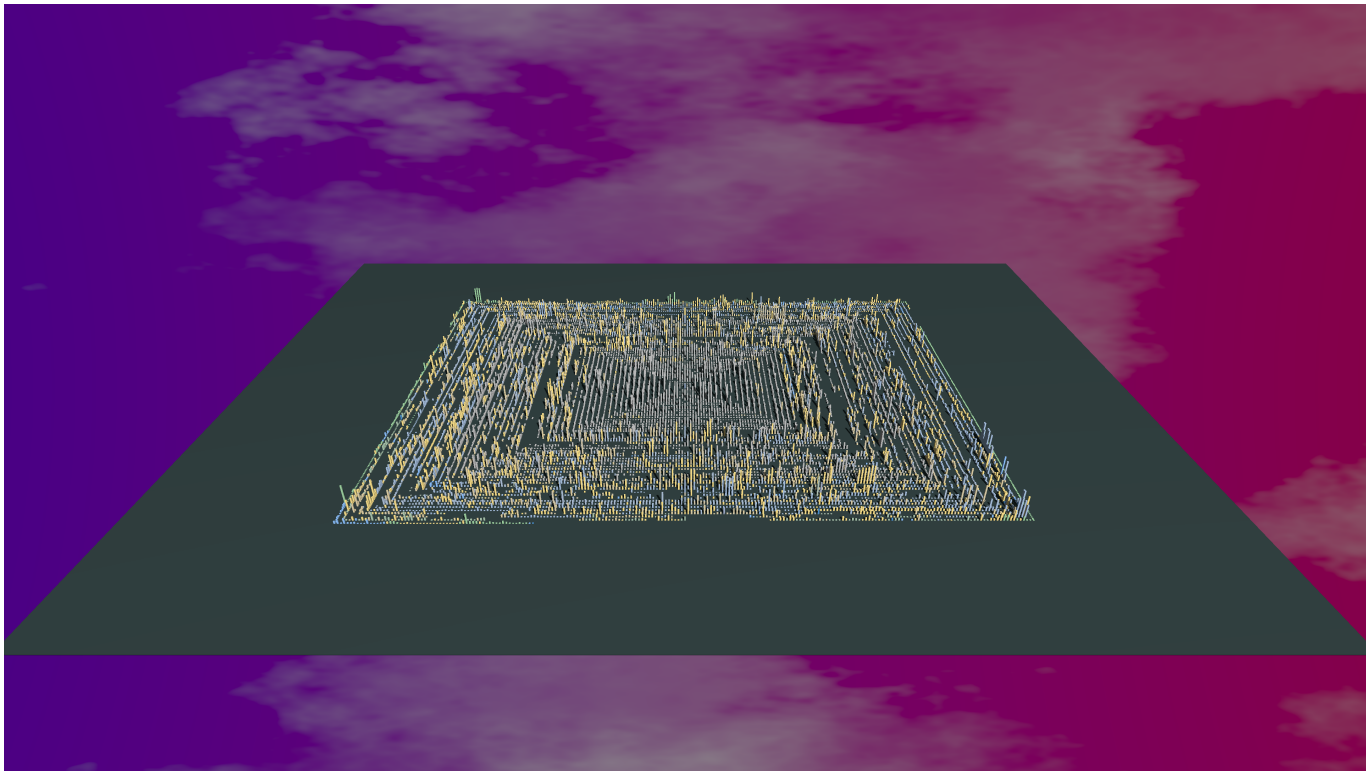


FIGURE C.5: Elasticsearch in June 2018

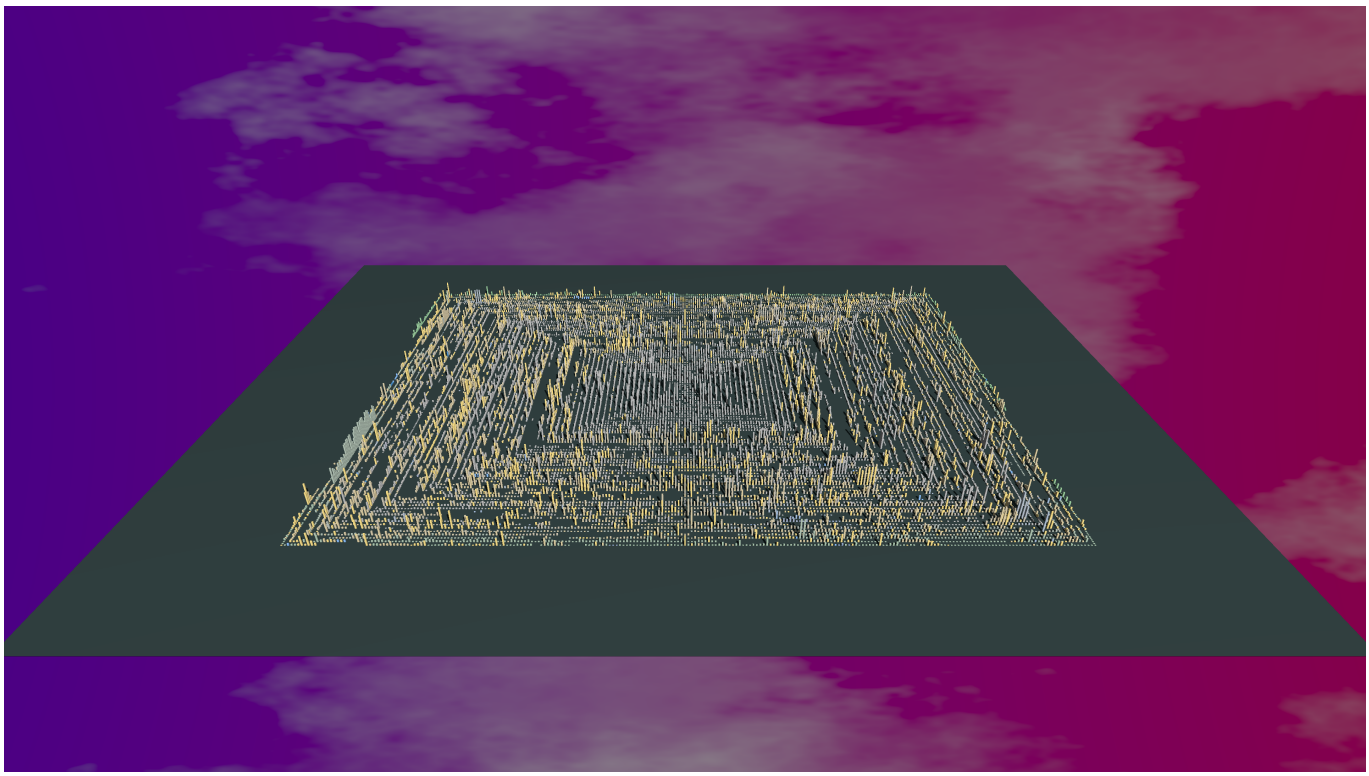


FIGURE C.6: Elasticsearch in June 2019

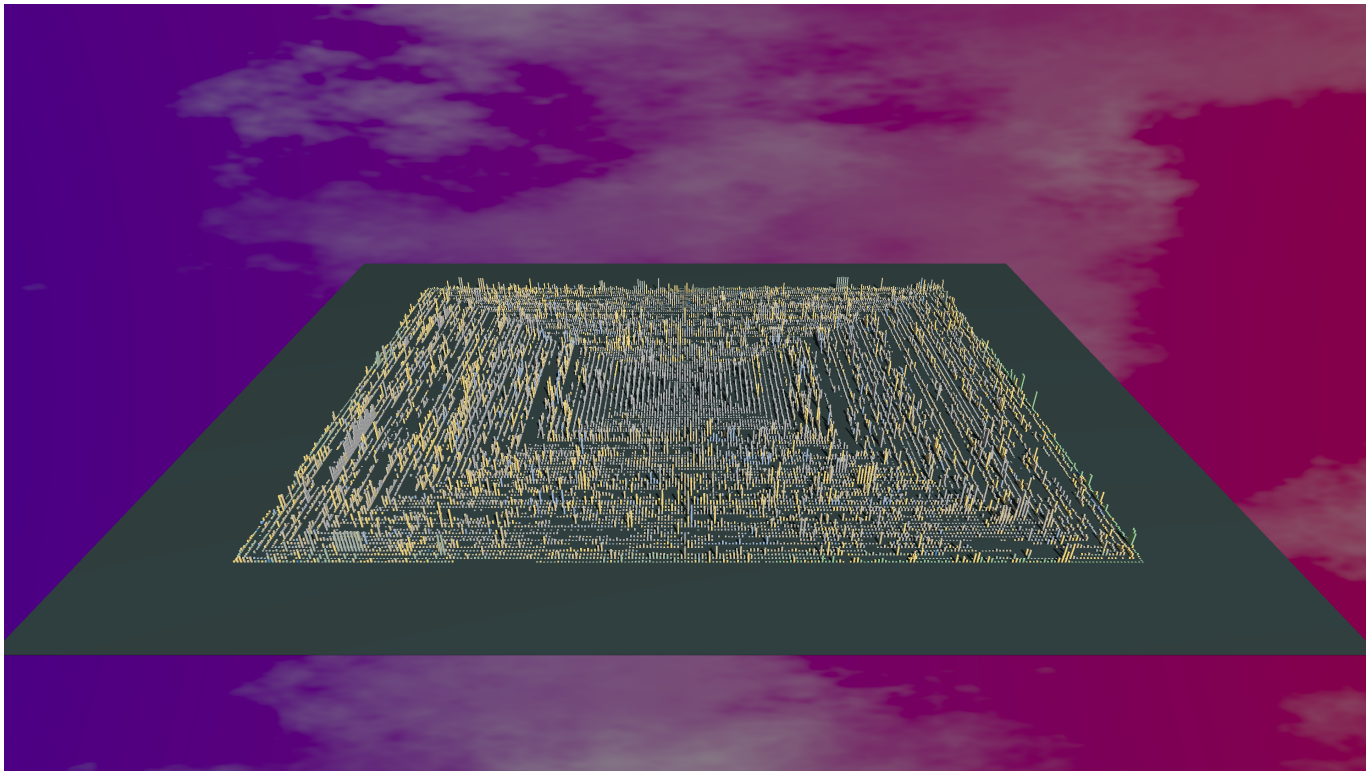


FIGURE C.7: Elasticsearch in June 2020

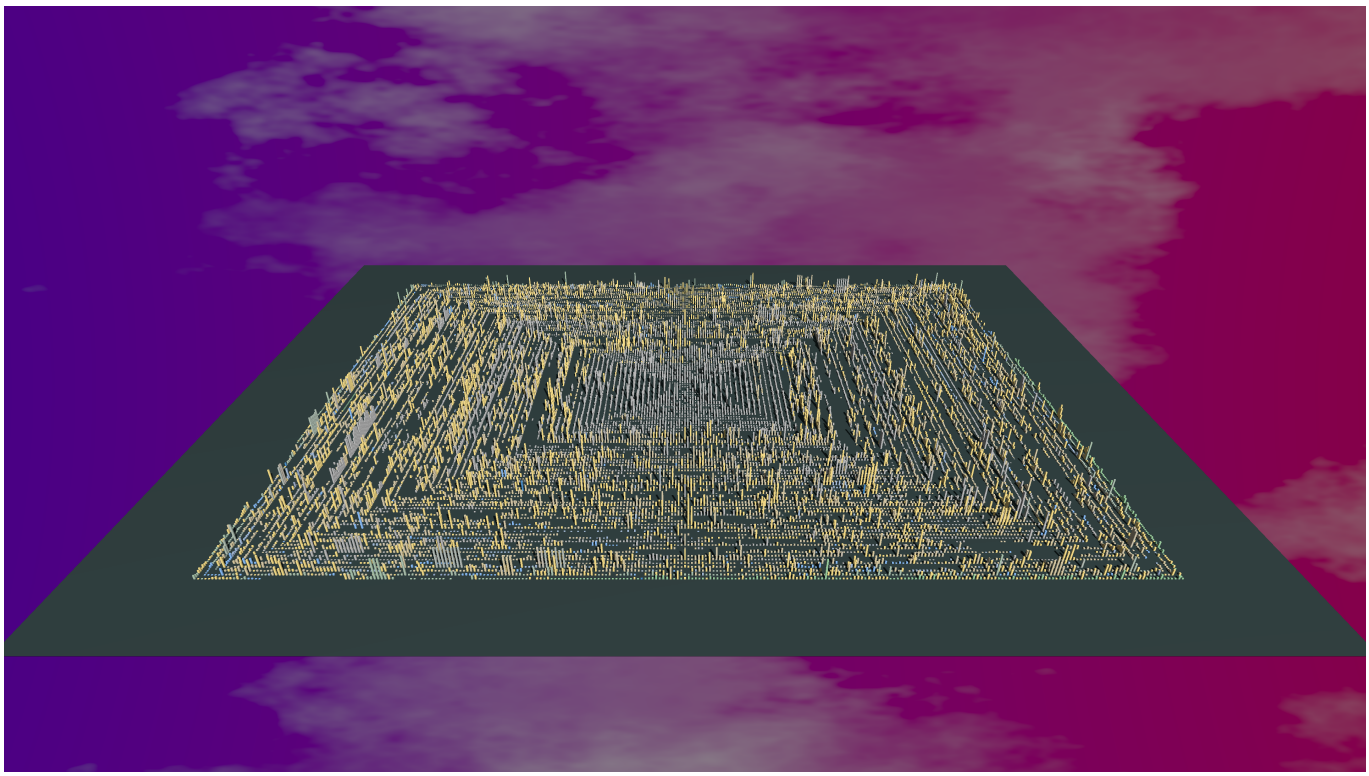


FIGURE C.8: Elasticsearch in June 2021

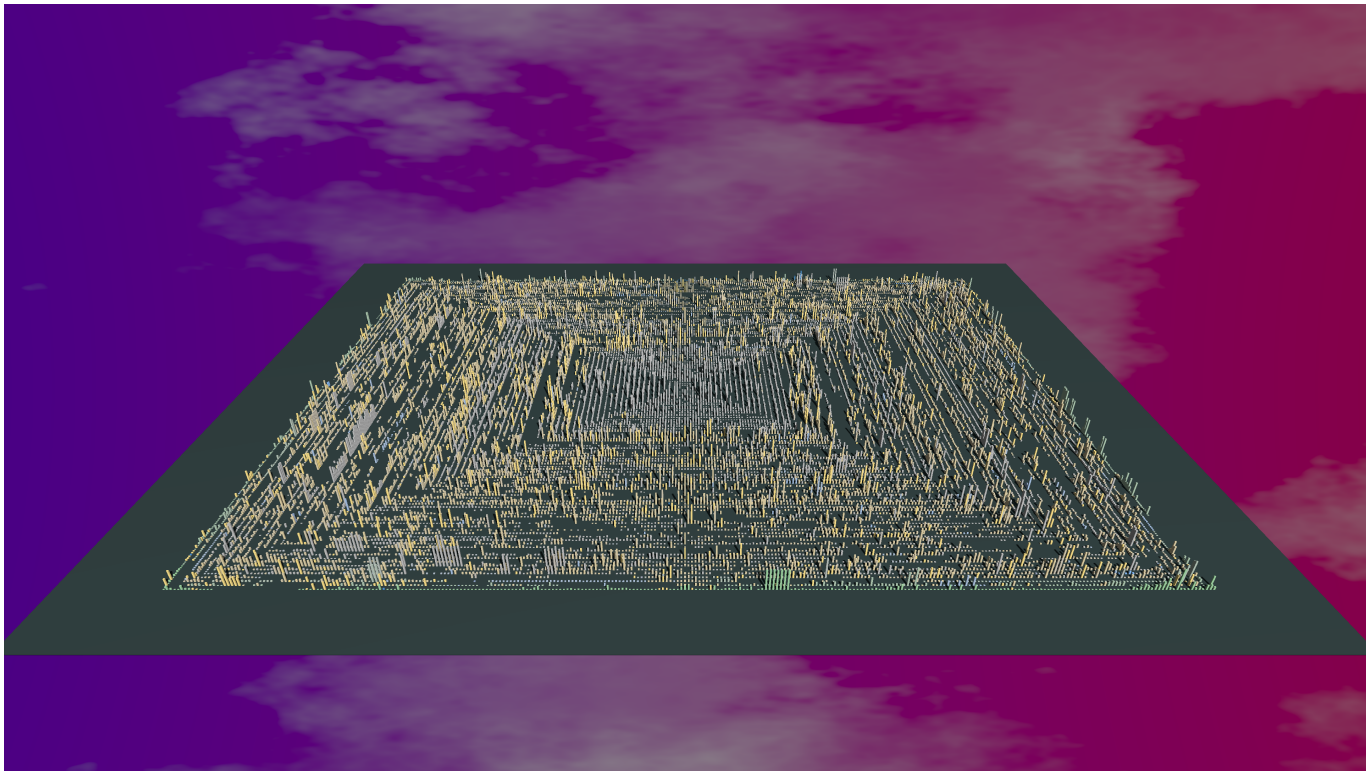


FIGURE C.9: Elasticsearch in June 2022

Appendix D

Evolution of LibreOffice

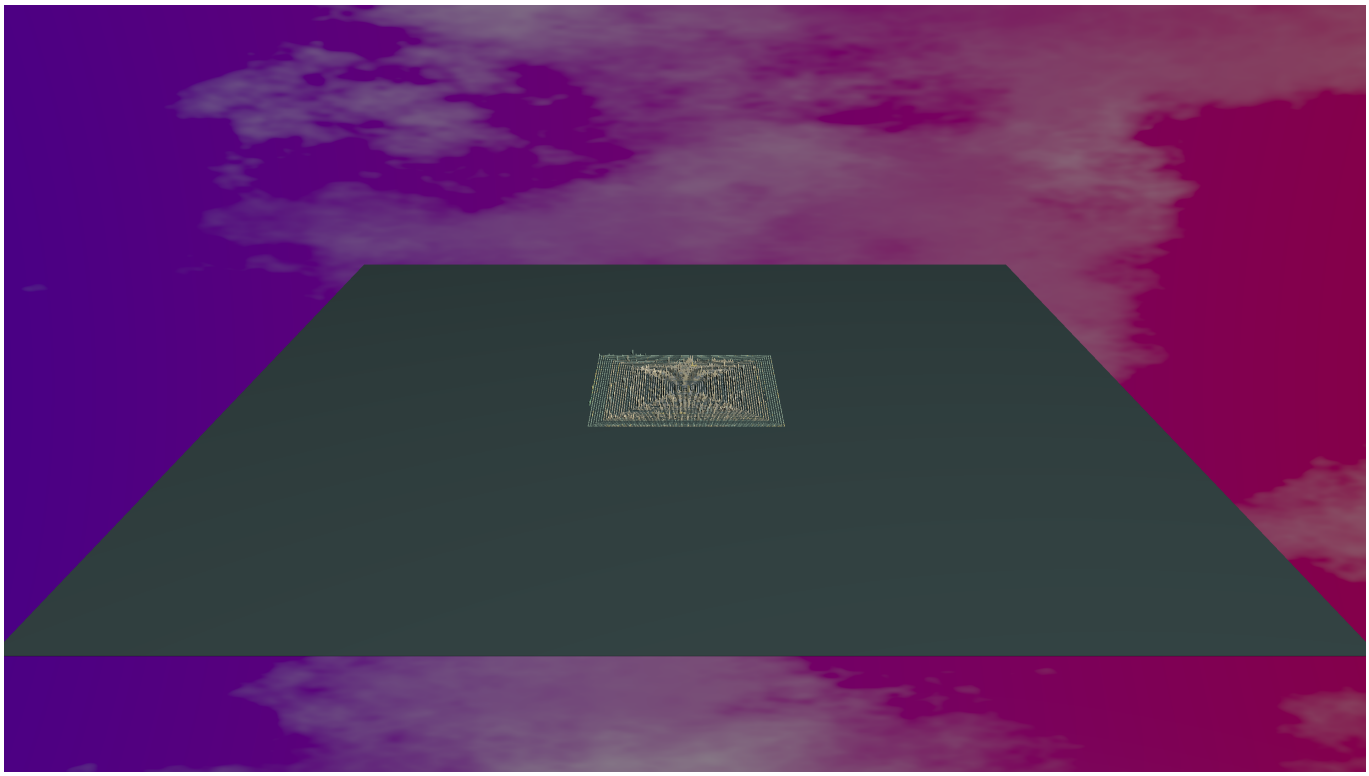


FIGURE D.1: LibreOffice in March 2003

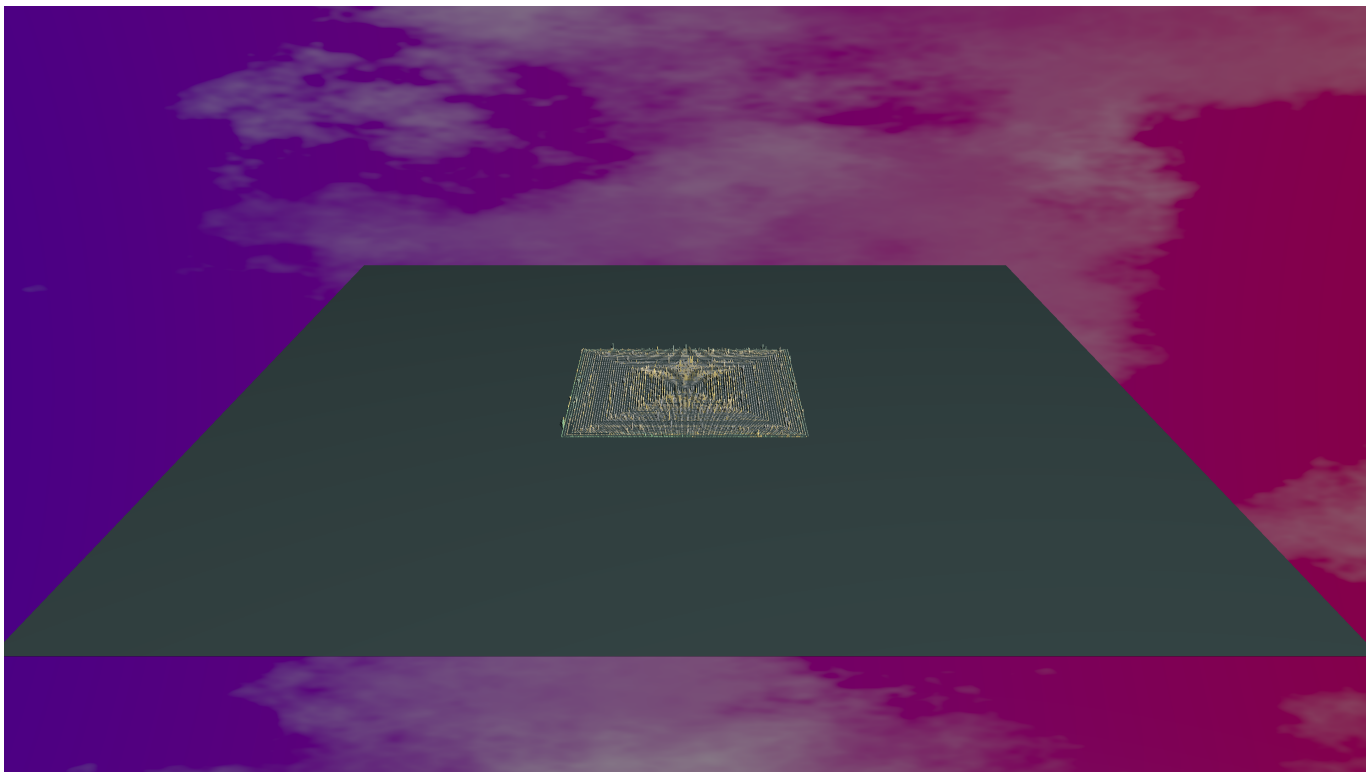


FIGURE D.2: LibreOffice in March 2004

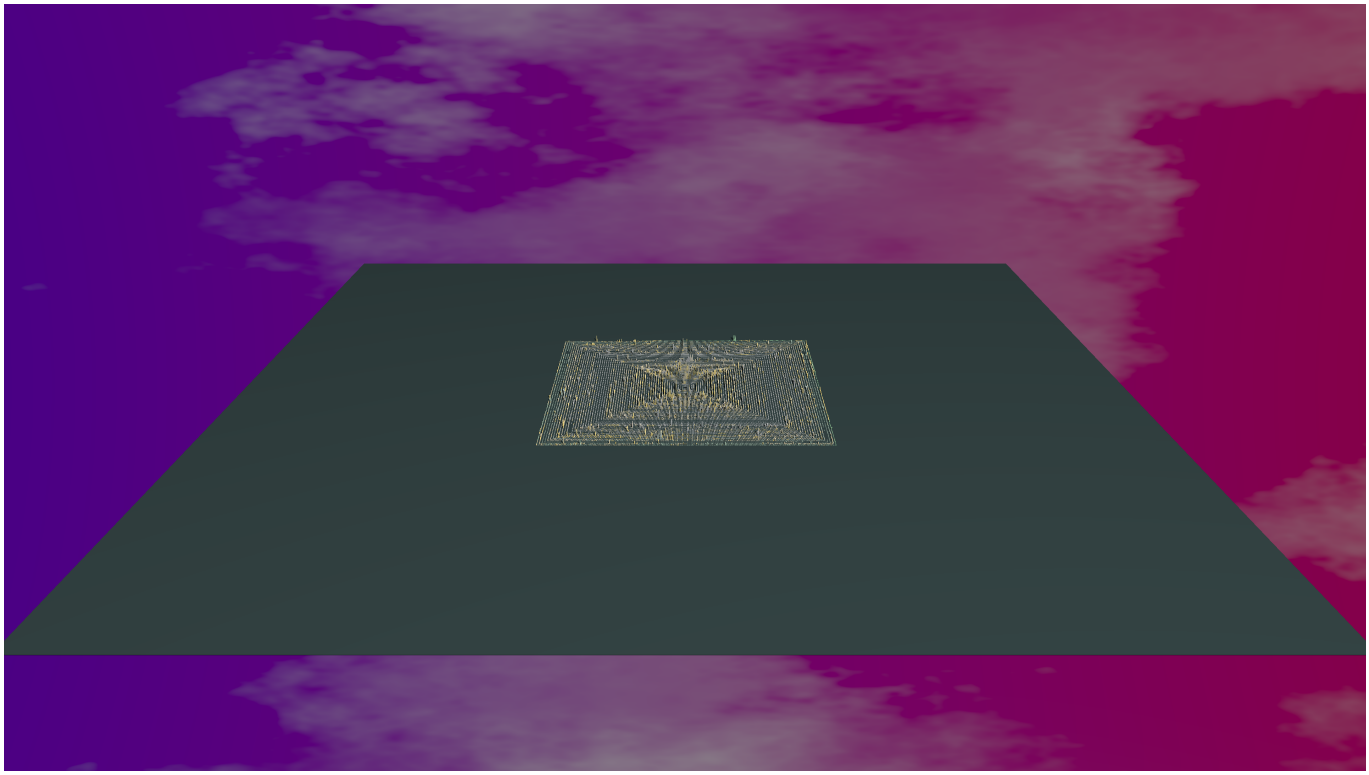


FIGURE D.3: LibreOffice in March 2005

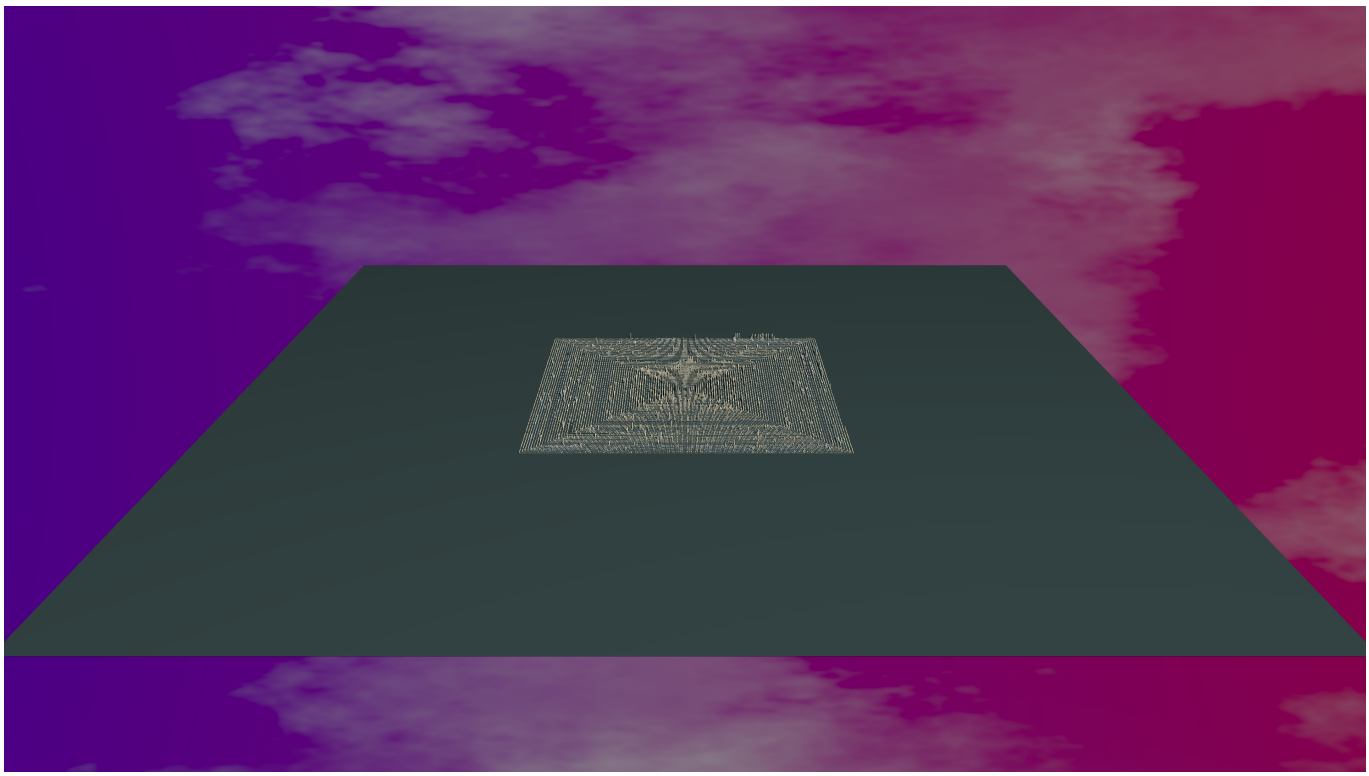


FIGURE D.4: LibreOffice in March 2006

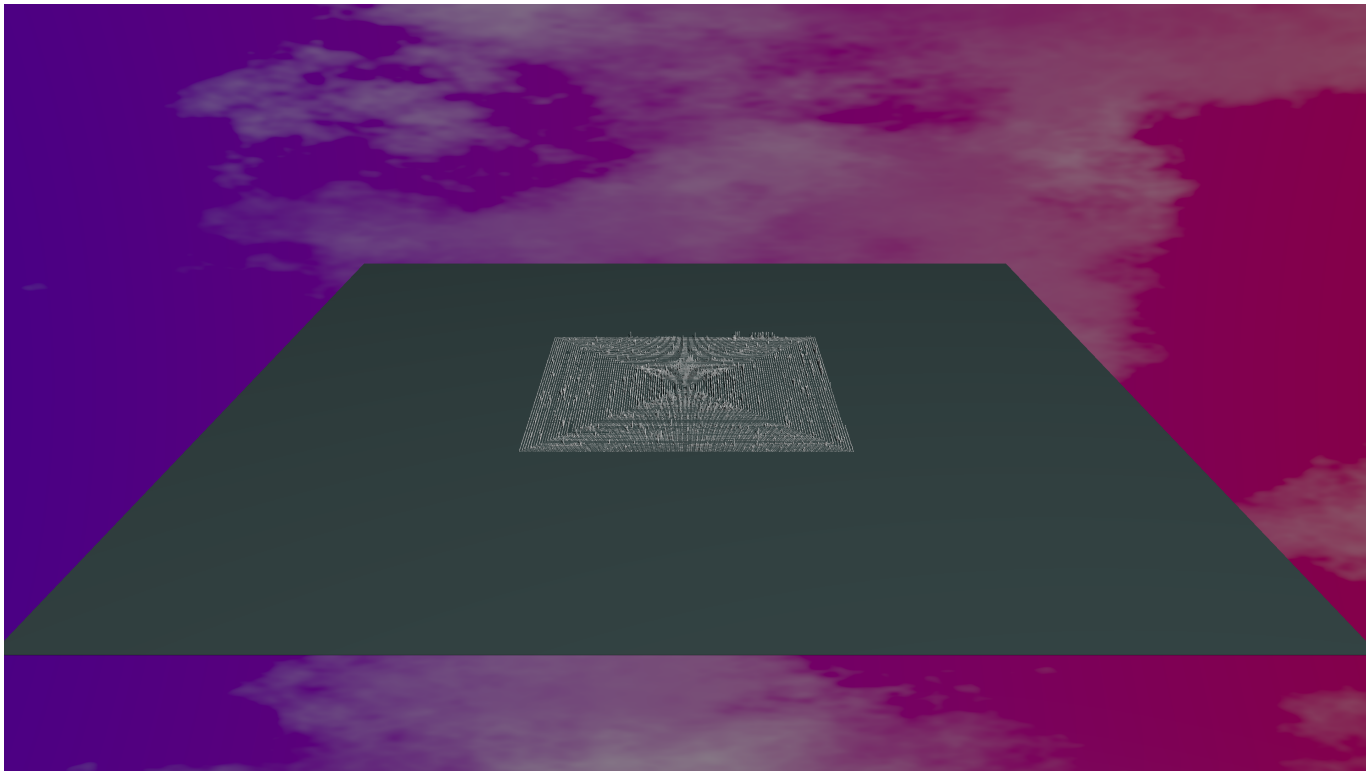


FIGURE D.5: LibreOffice in March 2007

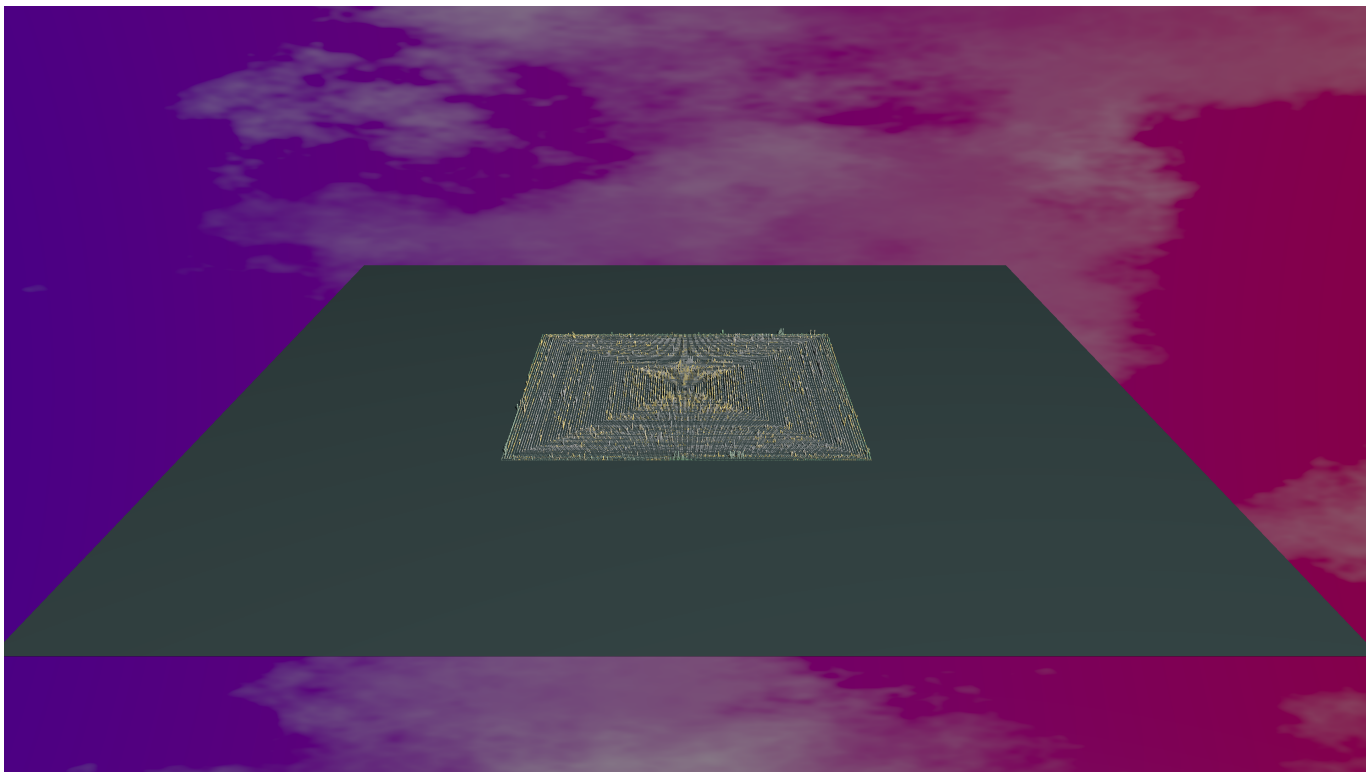


FIGURE D.6: LibreOffice in March 2008

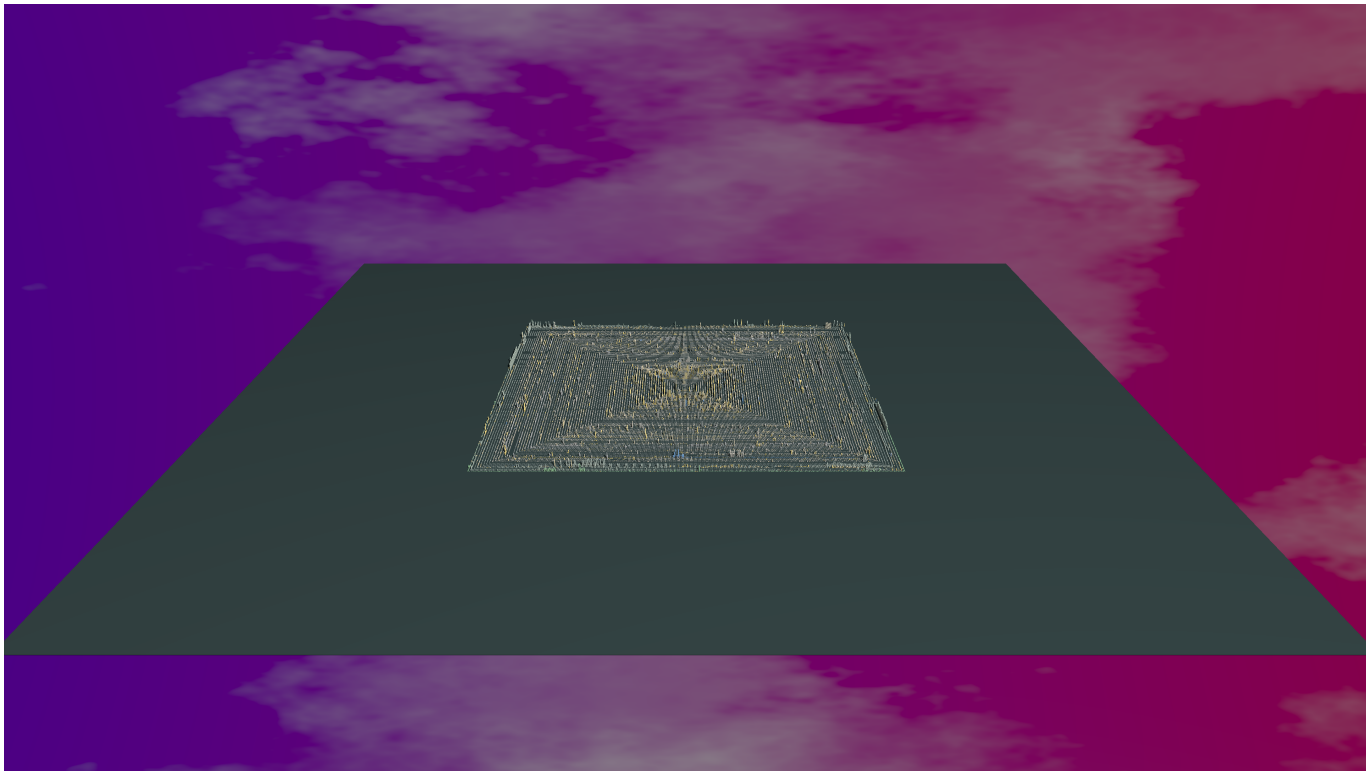


FIGURE D.7: LibreOffice in March 2009

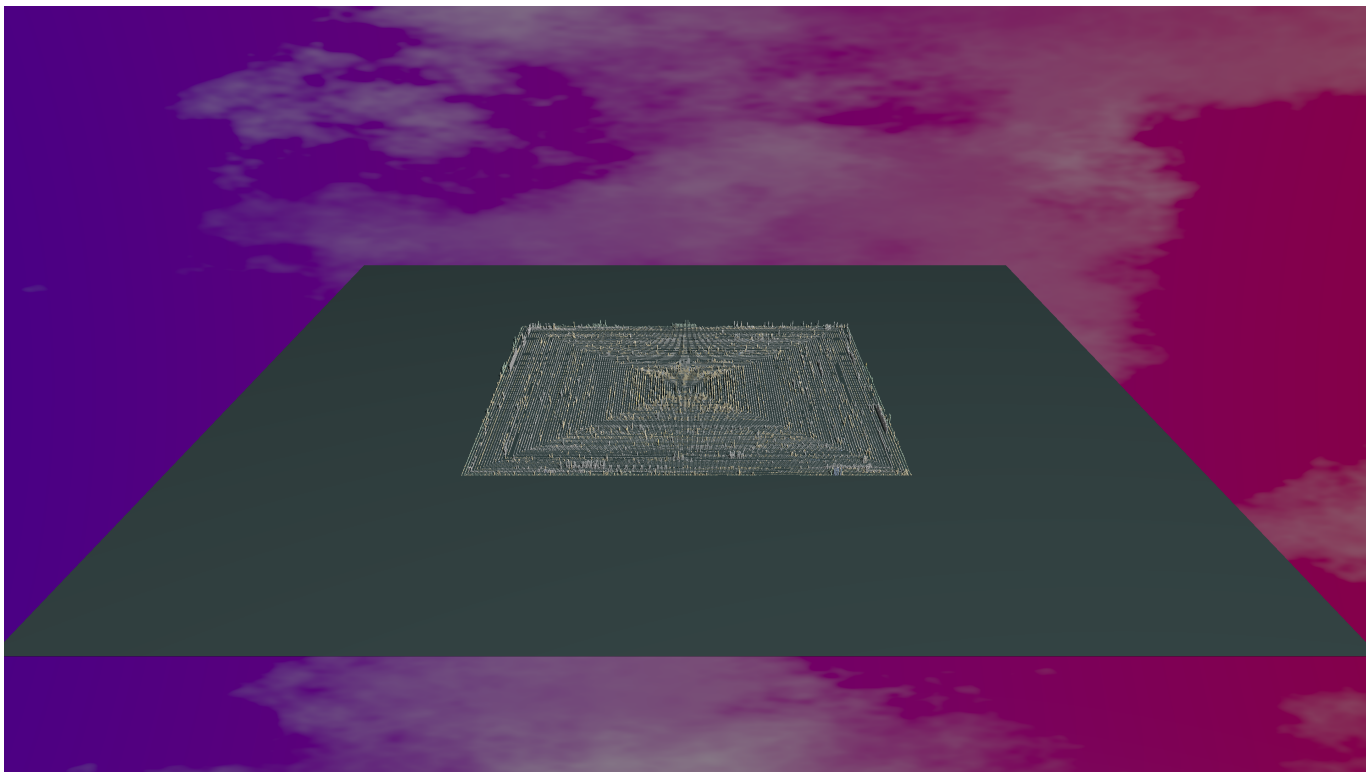


FIGURE D.8: LibreOffice in March 2010

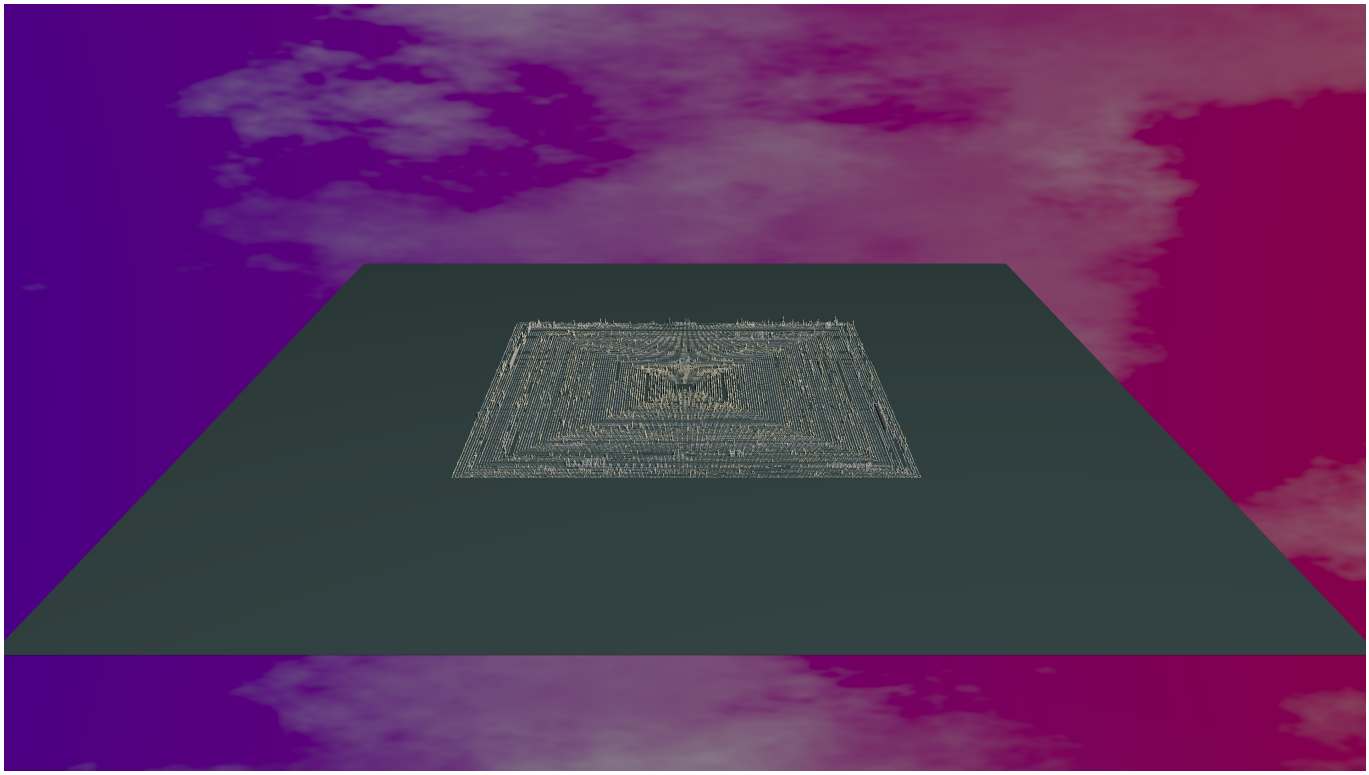


FIGURE D.9: LibreOffice in March 2011

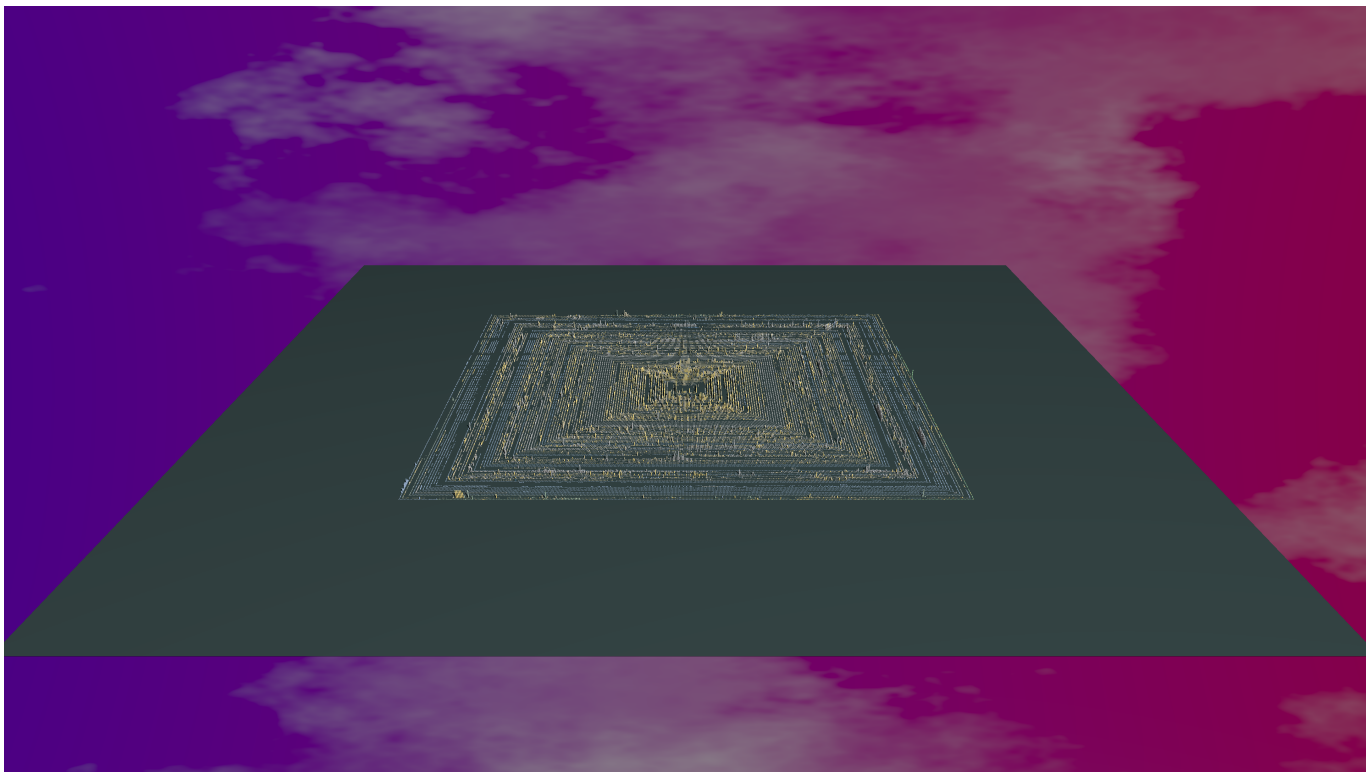


FIGURE D.10: LibreOffice in March 2012

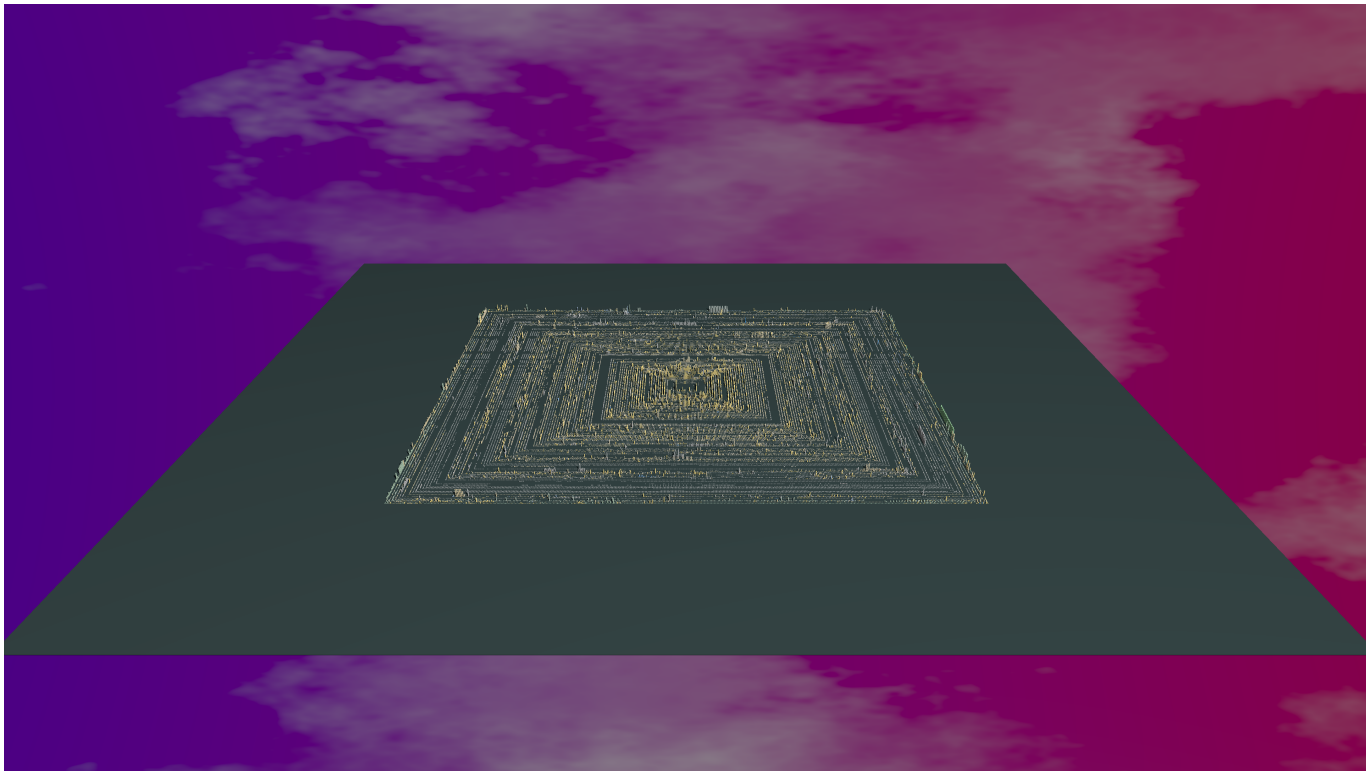


FIGURE D.11: LibreOffice in March 2013

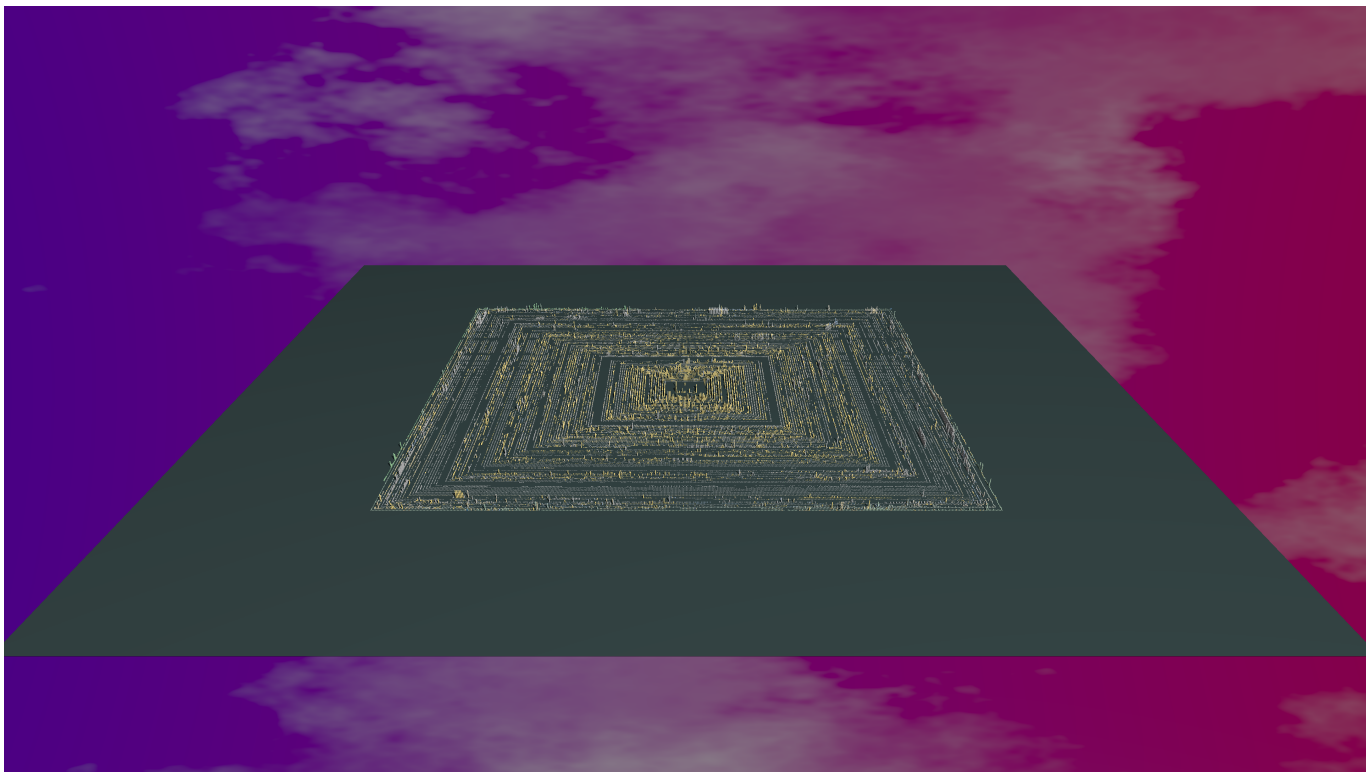


FIGURE D.12: LibreOffice in March 2014

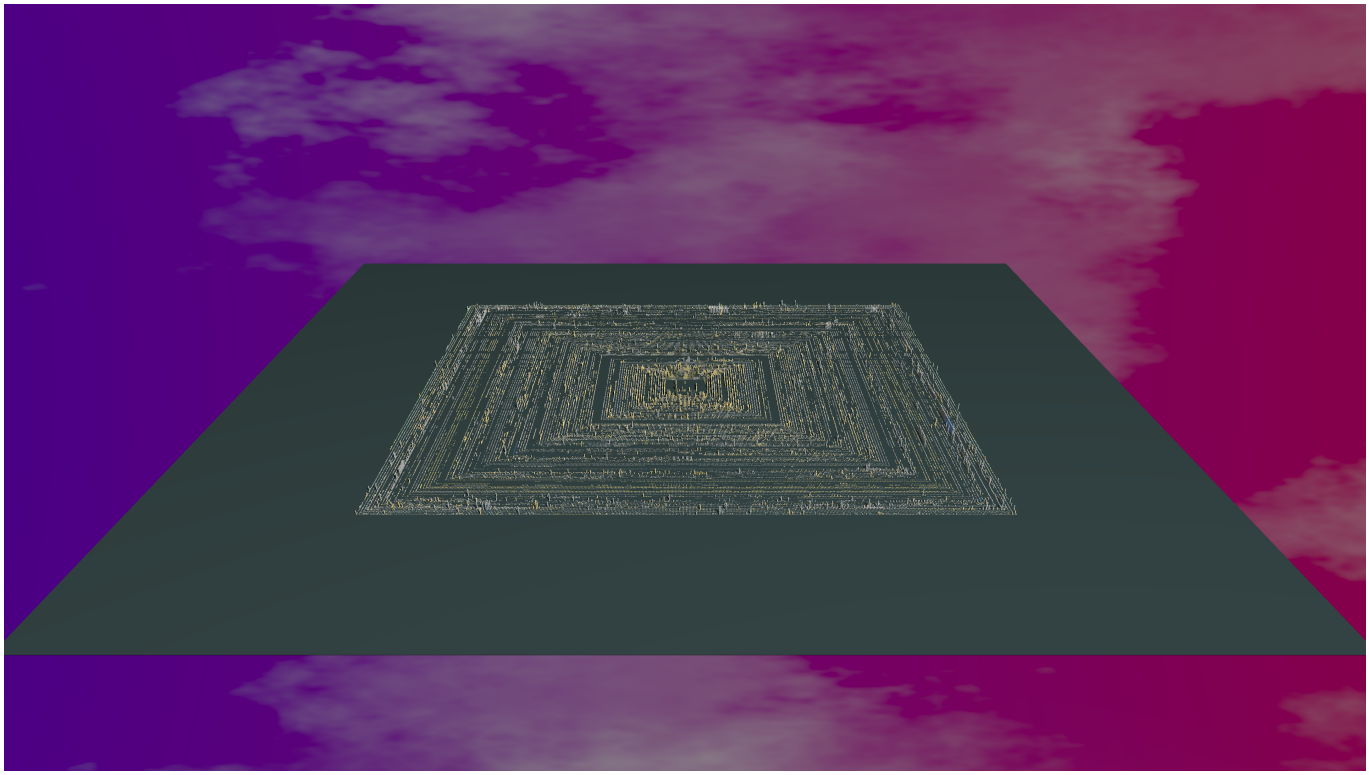


FIGURE D.13: LibreOffice in March 2015

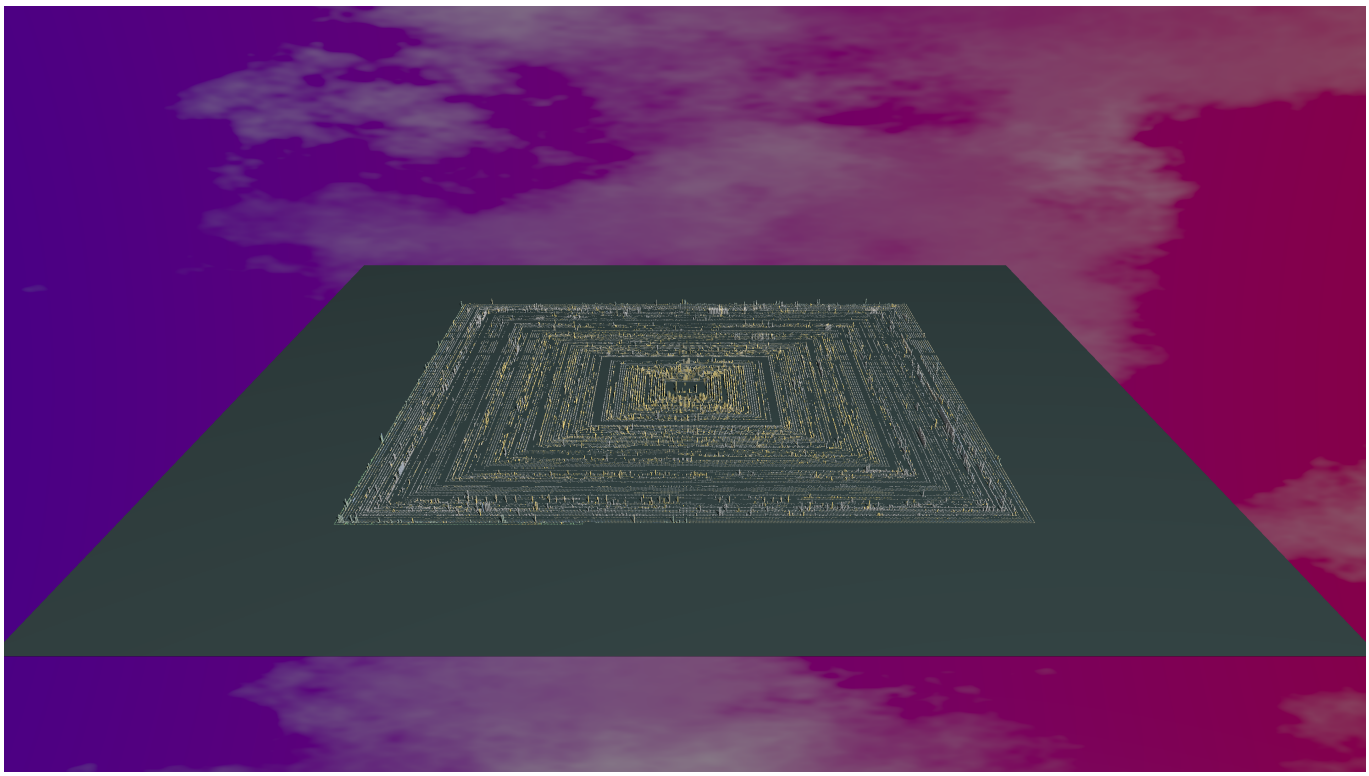


FIGURE D.14: LibreOffice in March 2016

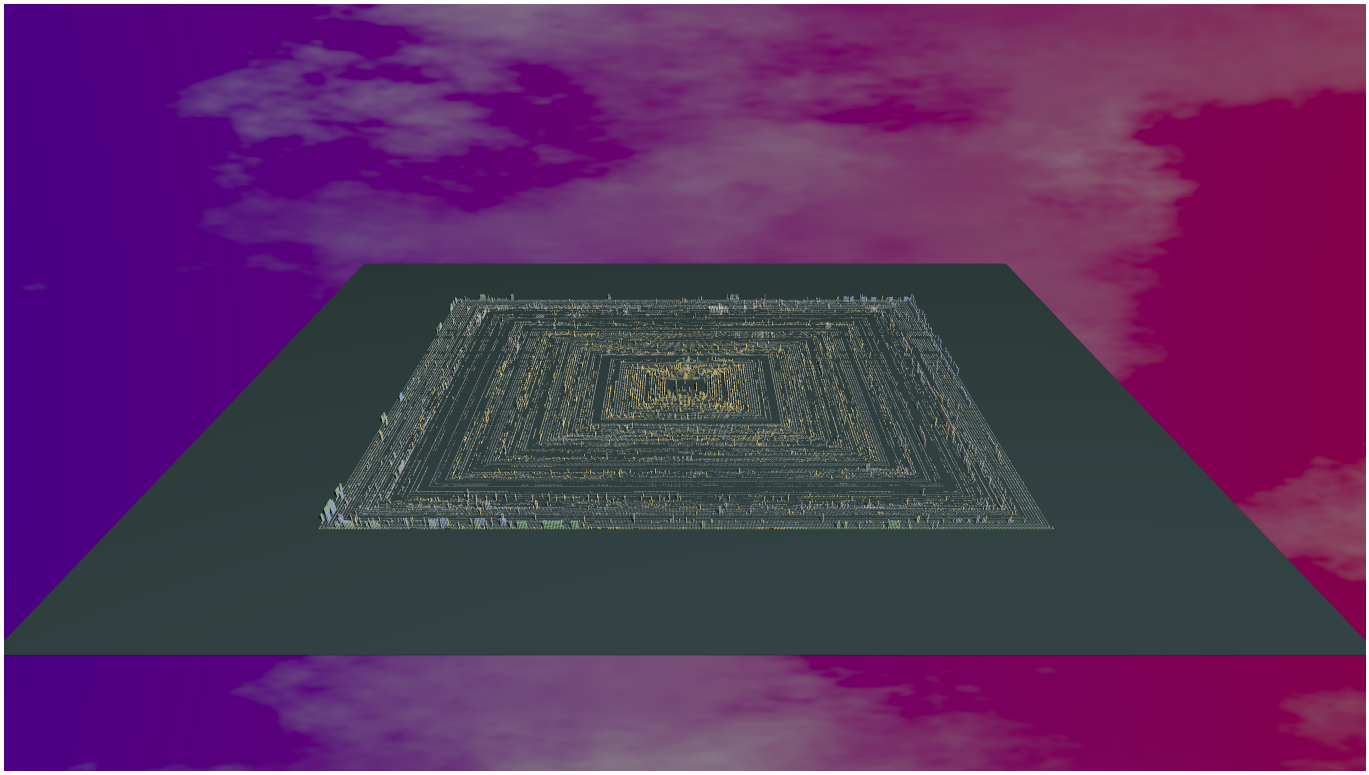


FIGURE D.15: LibreOffice in March 2017

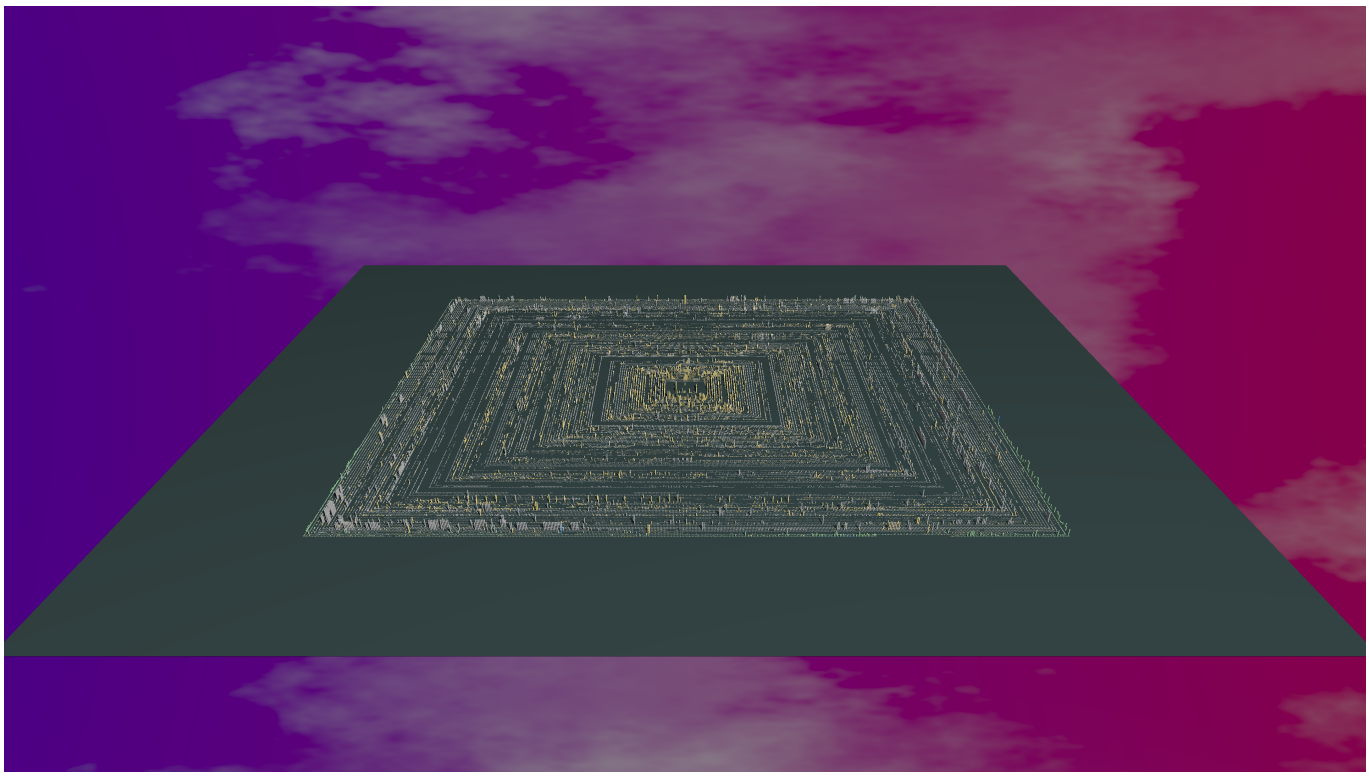


FIGURE D.16: LibreOffice in March 2018

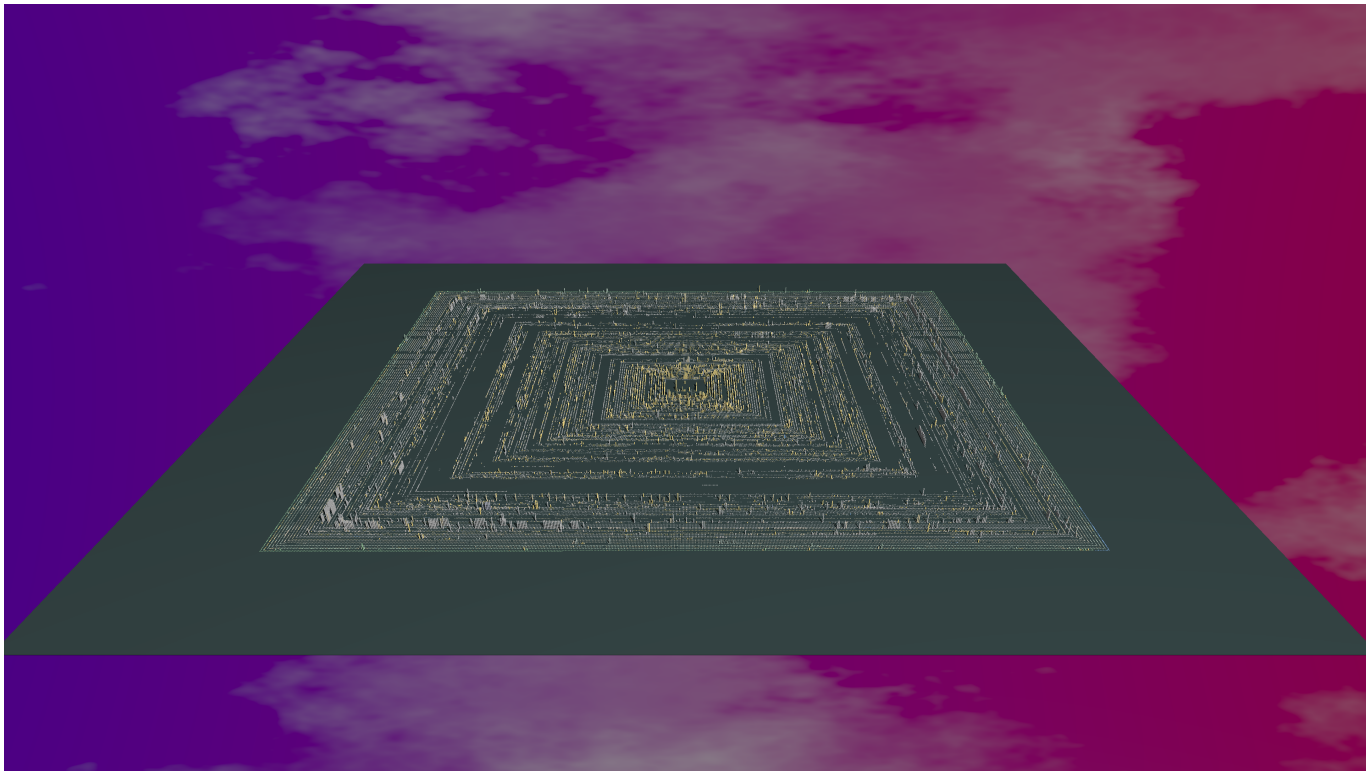


FIGURE D.17: LibreOffice in March 2019

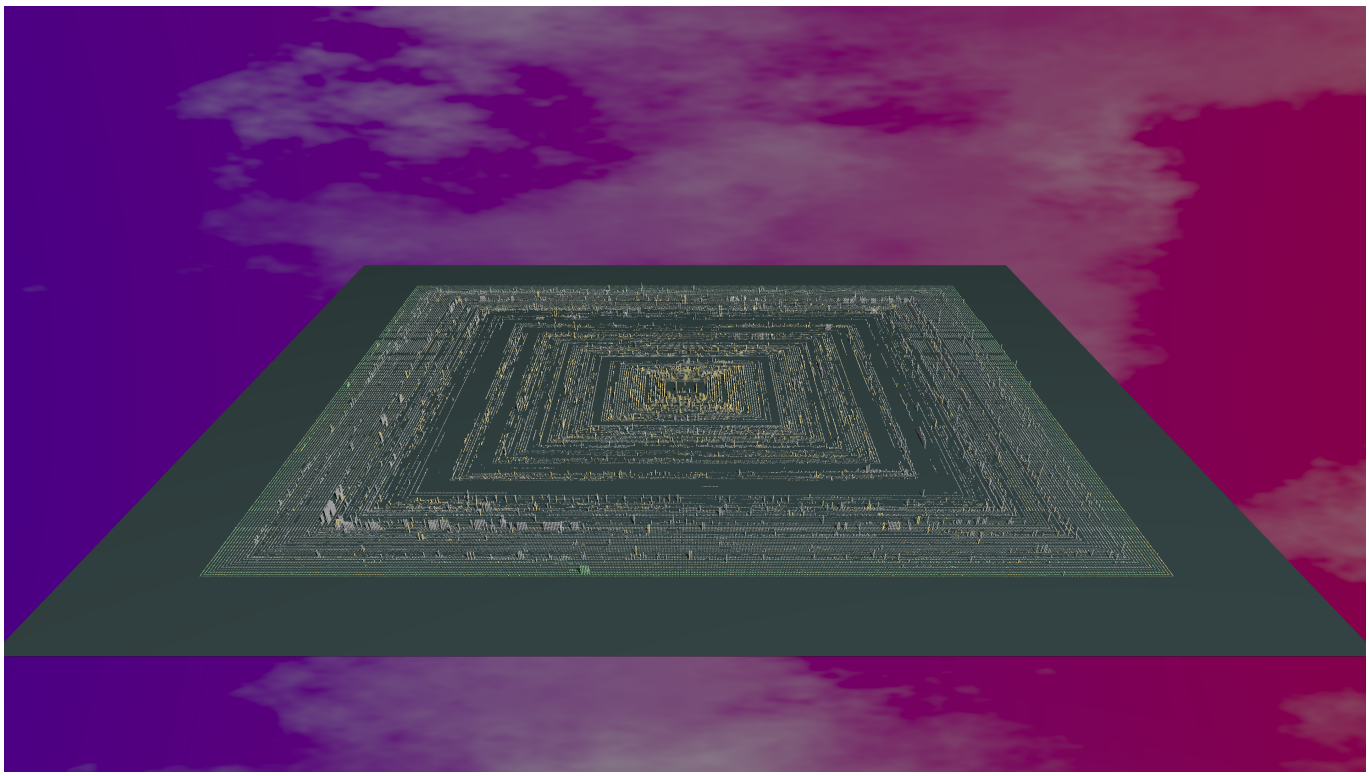


FIGURE D.18: LibreOffice in March 2020

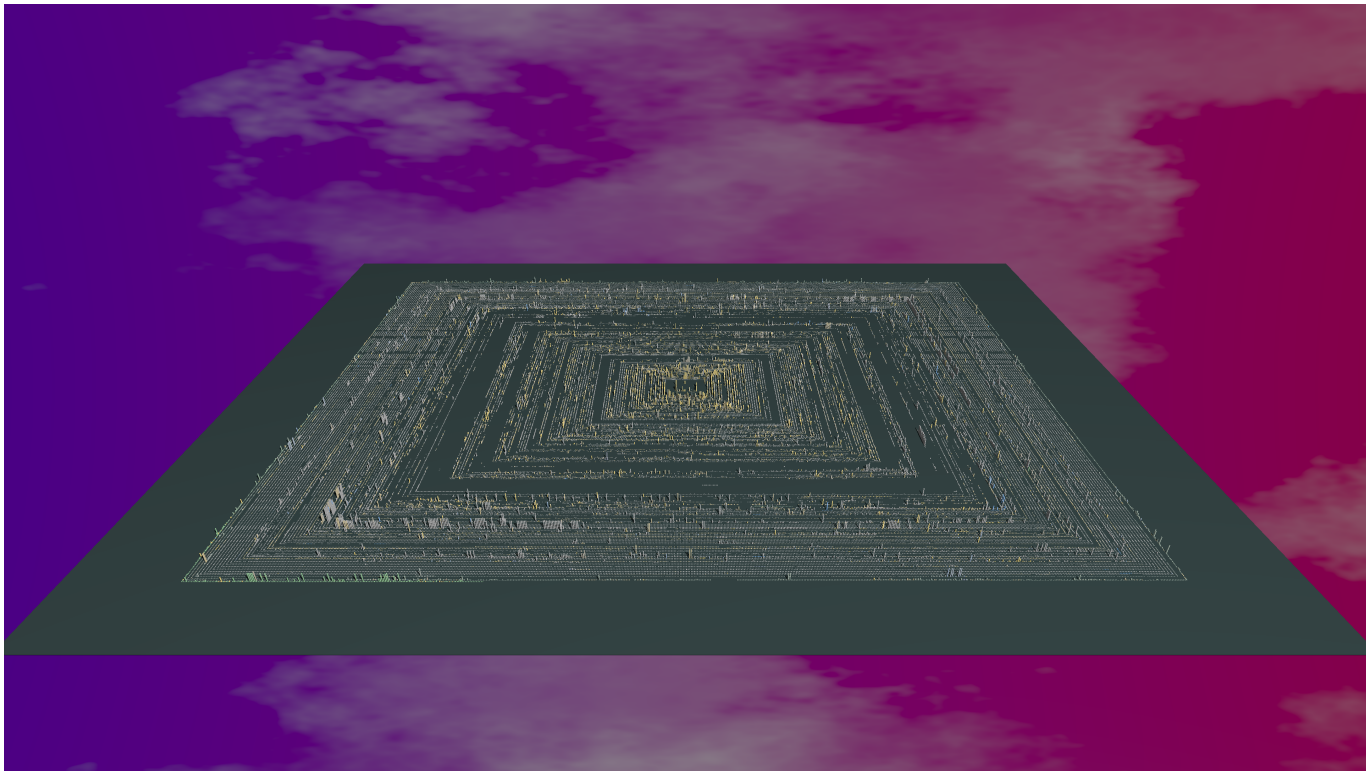


FIGURE D.19: LibreOffice in March 2021

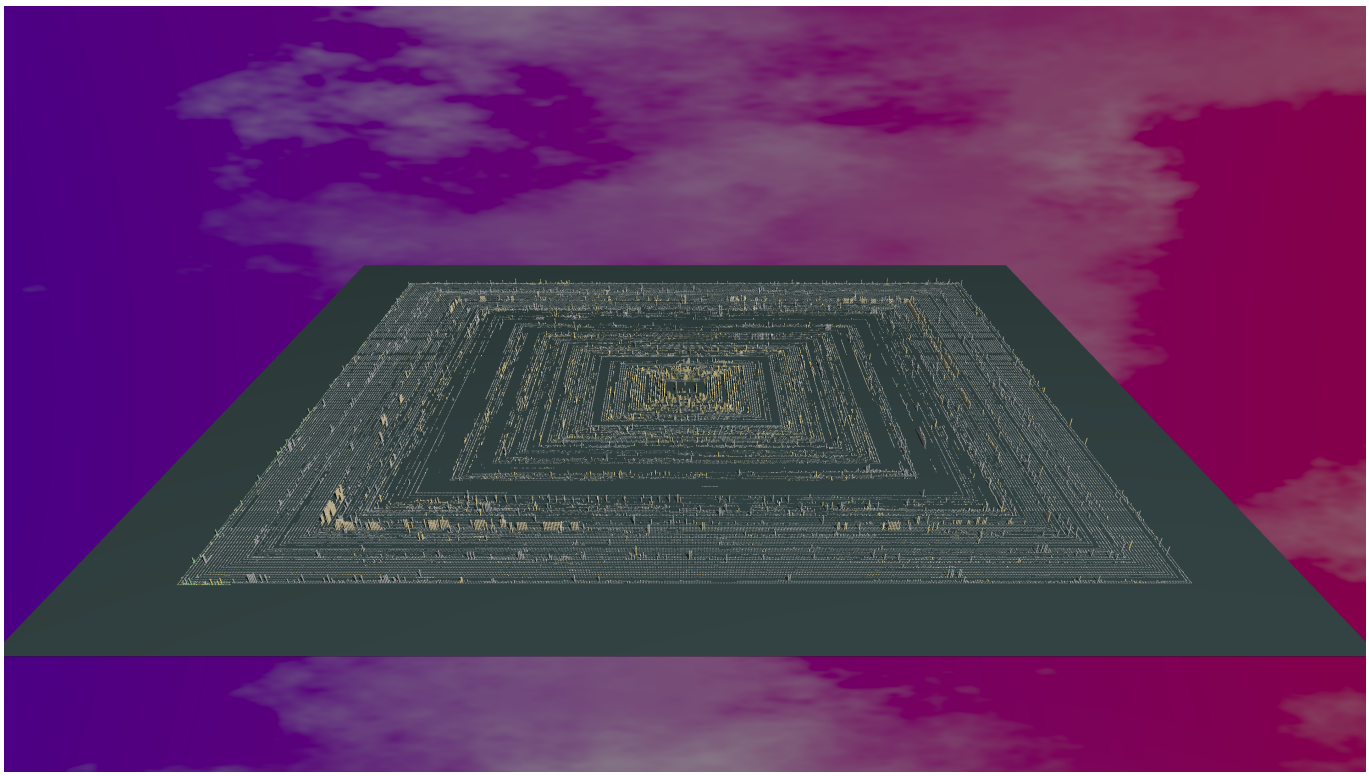


FIGURE D.20: LibreOffice in March 2022

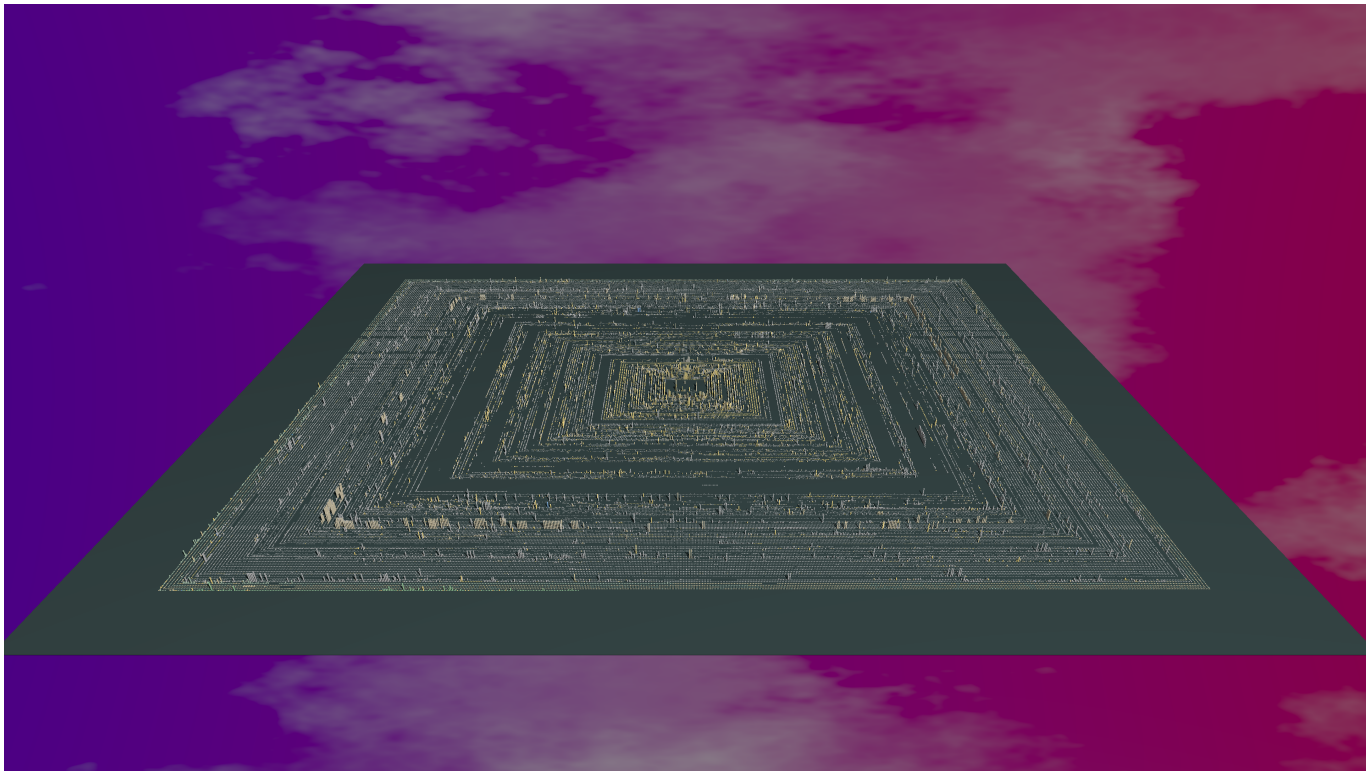


FIGURE D.21: LibreOffice in June 2022

Appendix E

Evolution of Linux

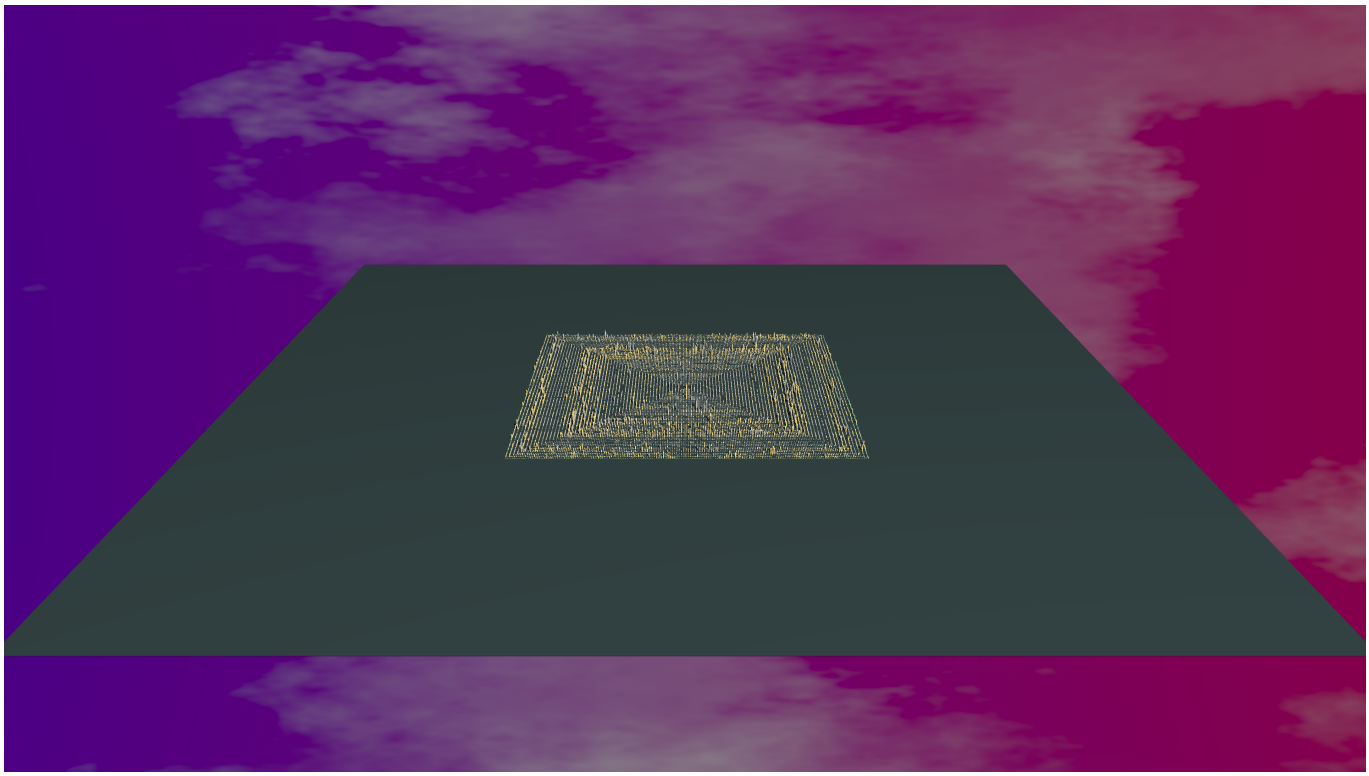


FIGURE E.1: Linux in April 2006

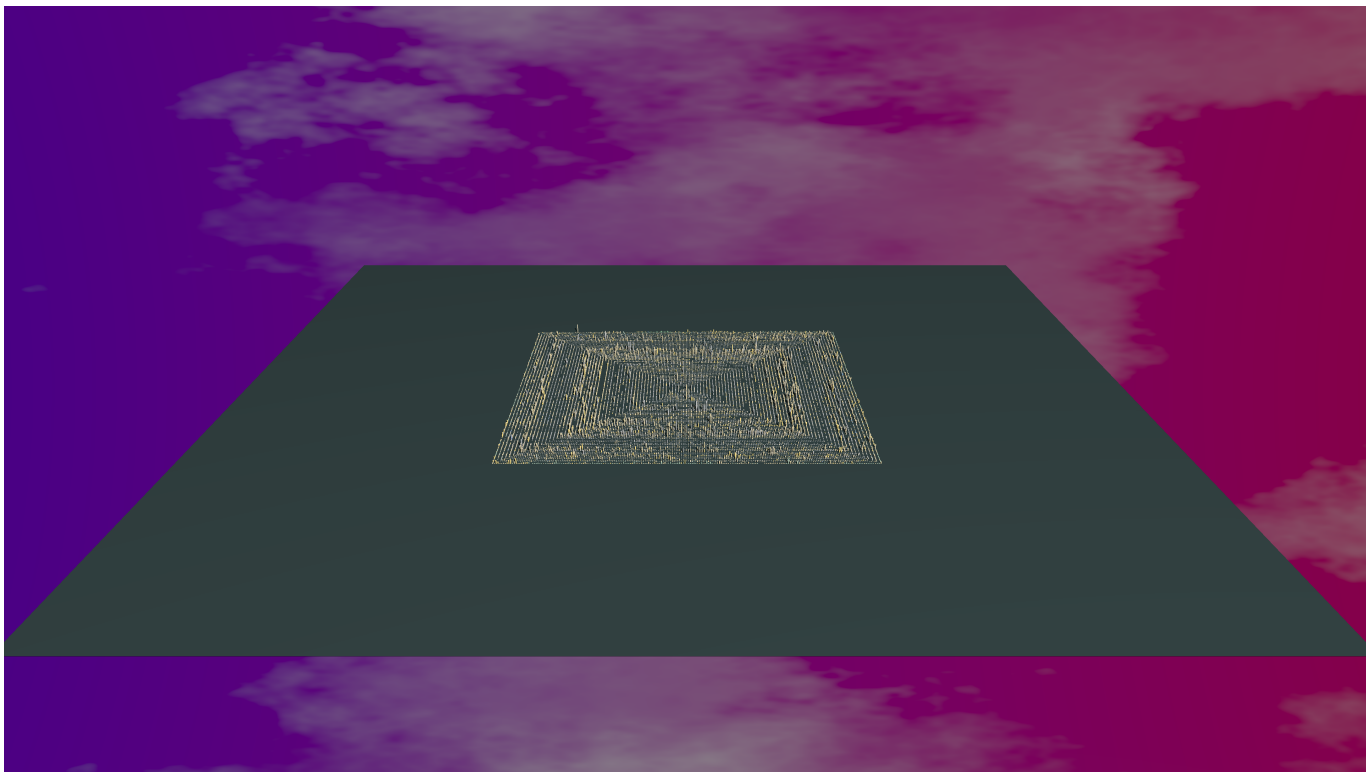


FIGURE E.2: Linux in April 2007

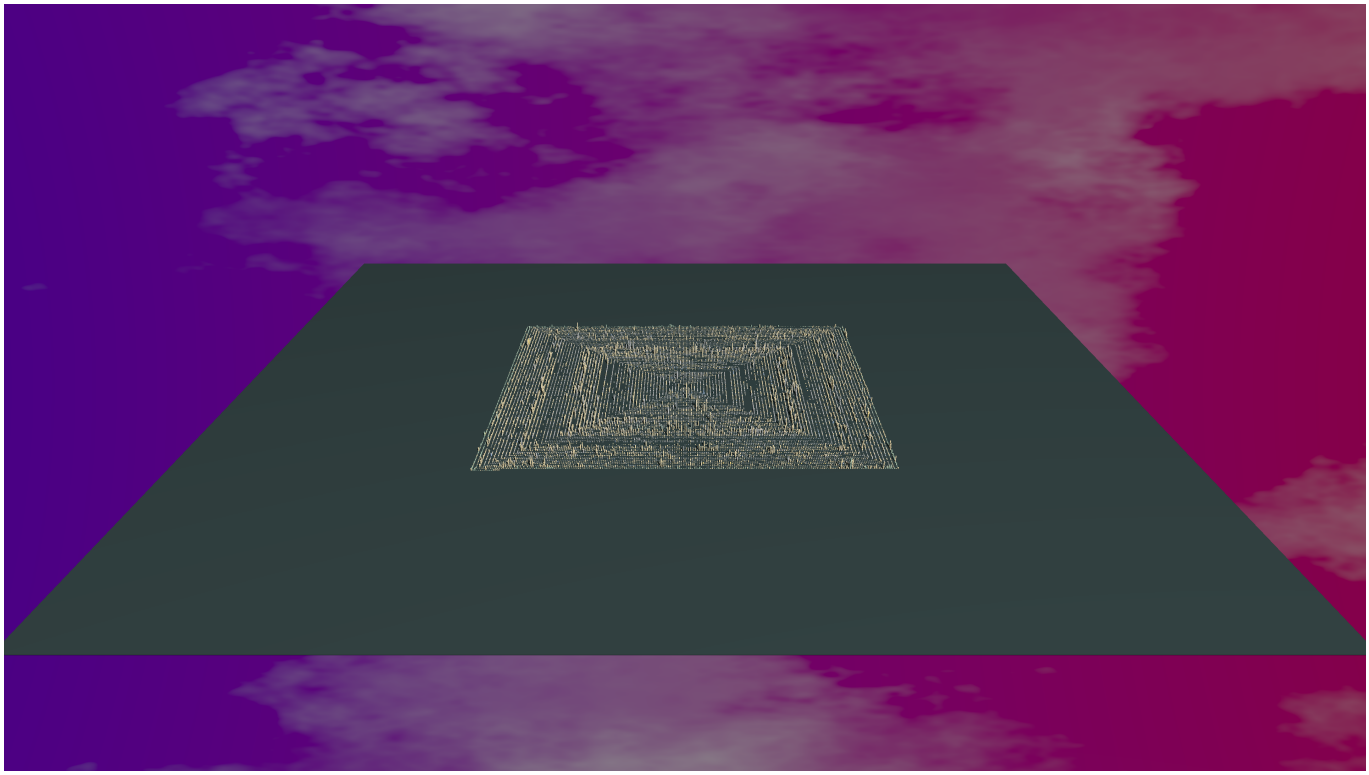


FIGURE E.3: Linux in April 2008

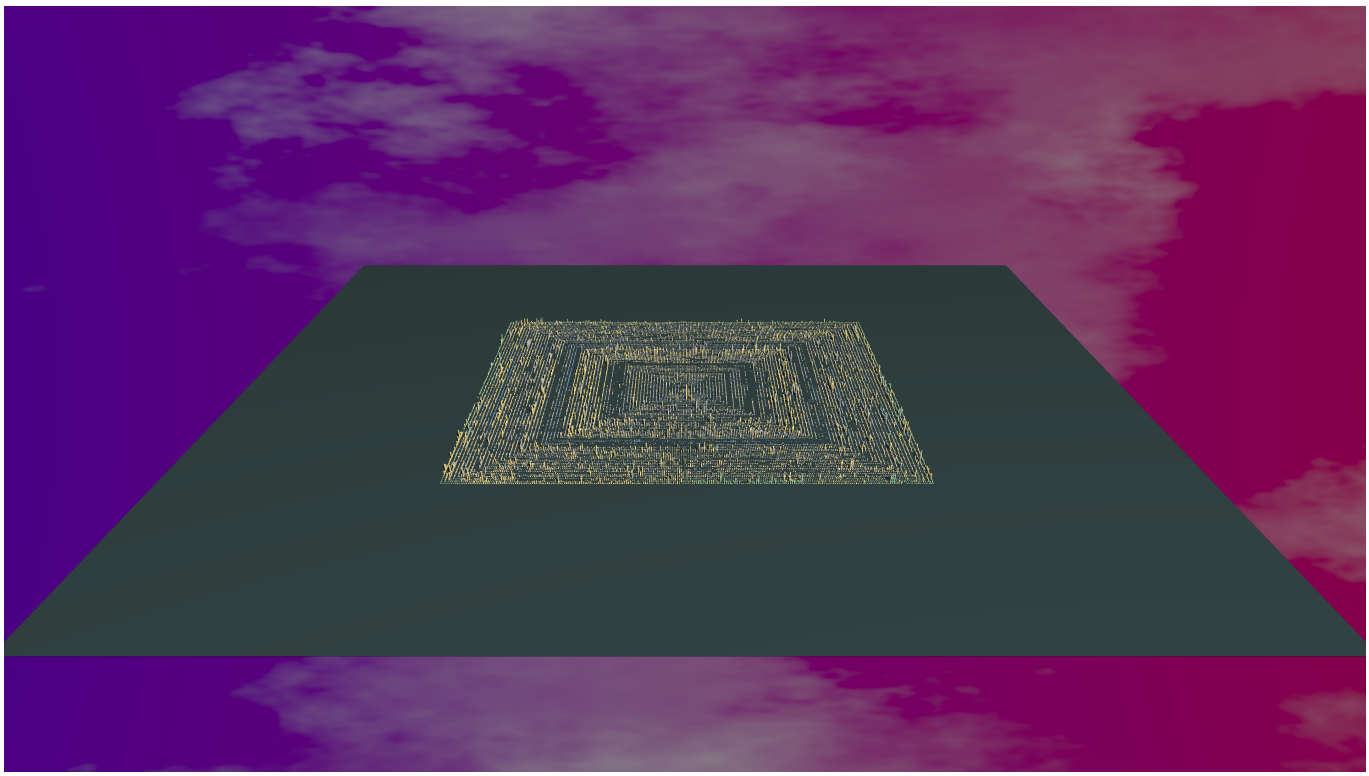


FIGURE E.4: Linux in April 2009

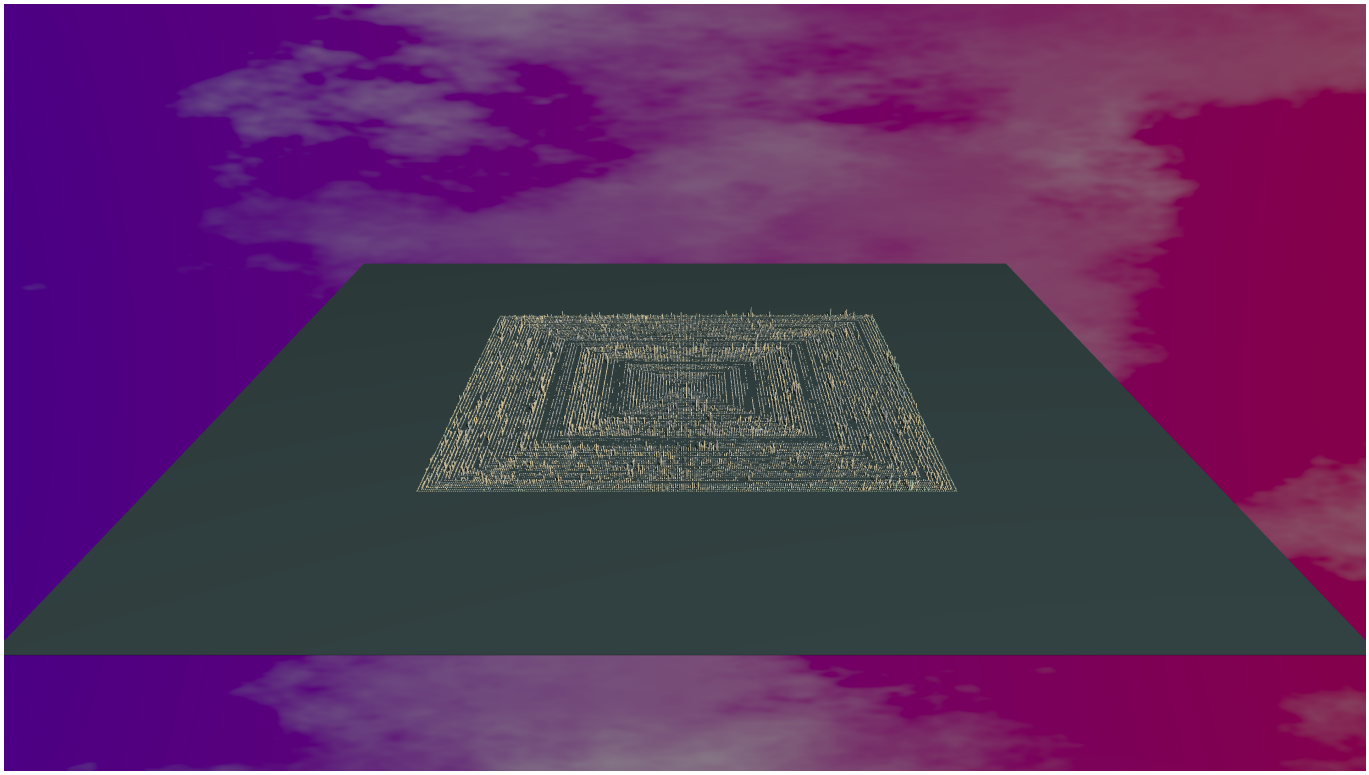


FIGURE E.5: Linux in April 2010

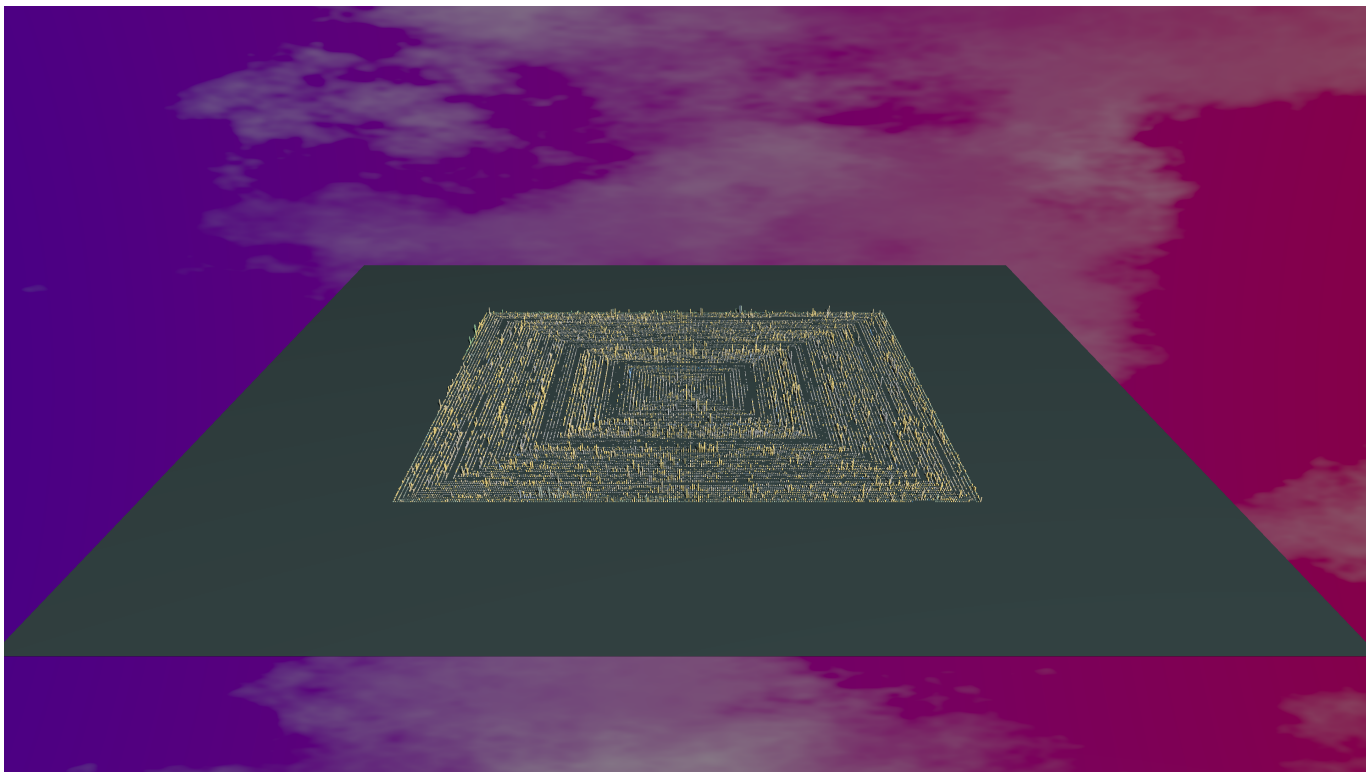


FIGURE E.6: Linux in April 2011

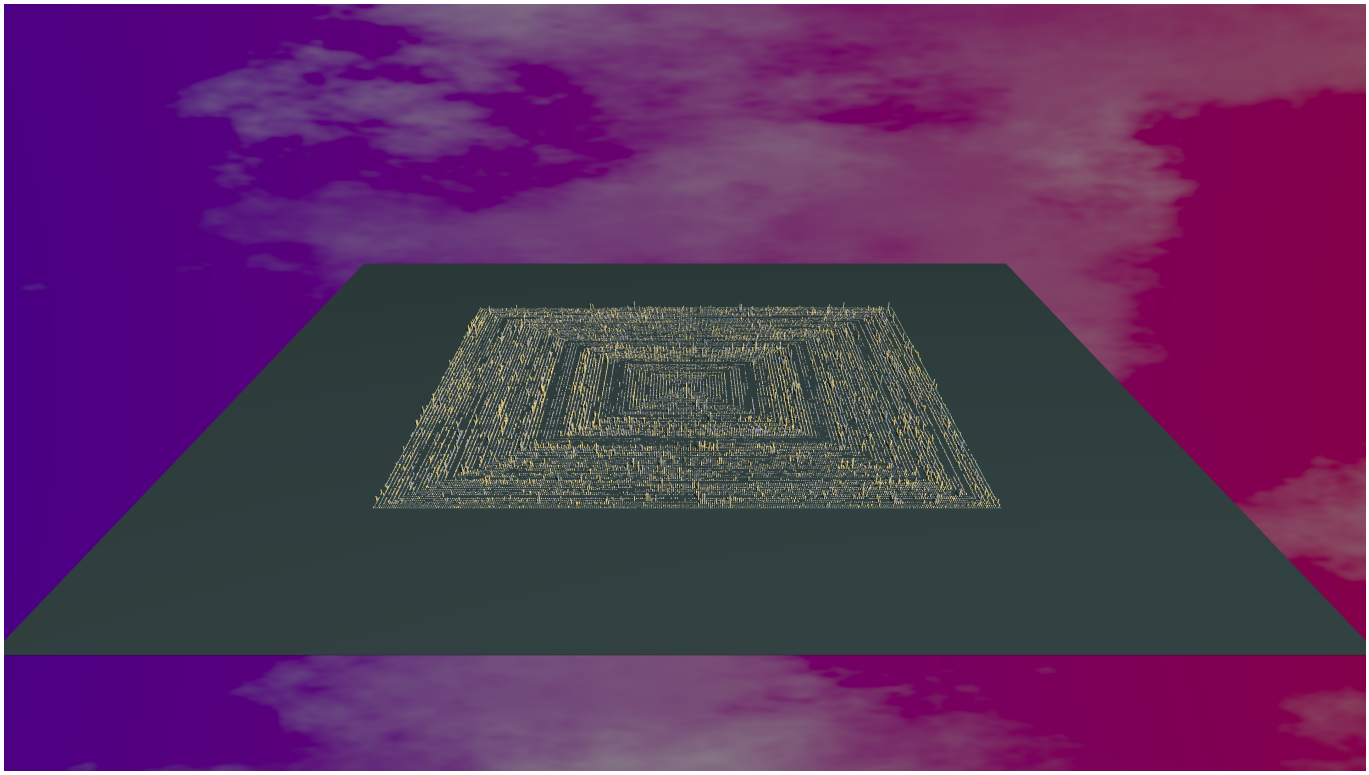


FIGURE E.7: Linux in April 2012

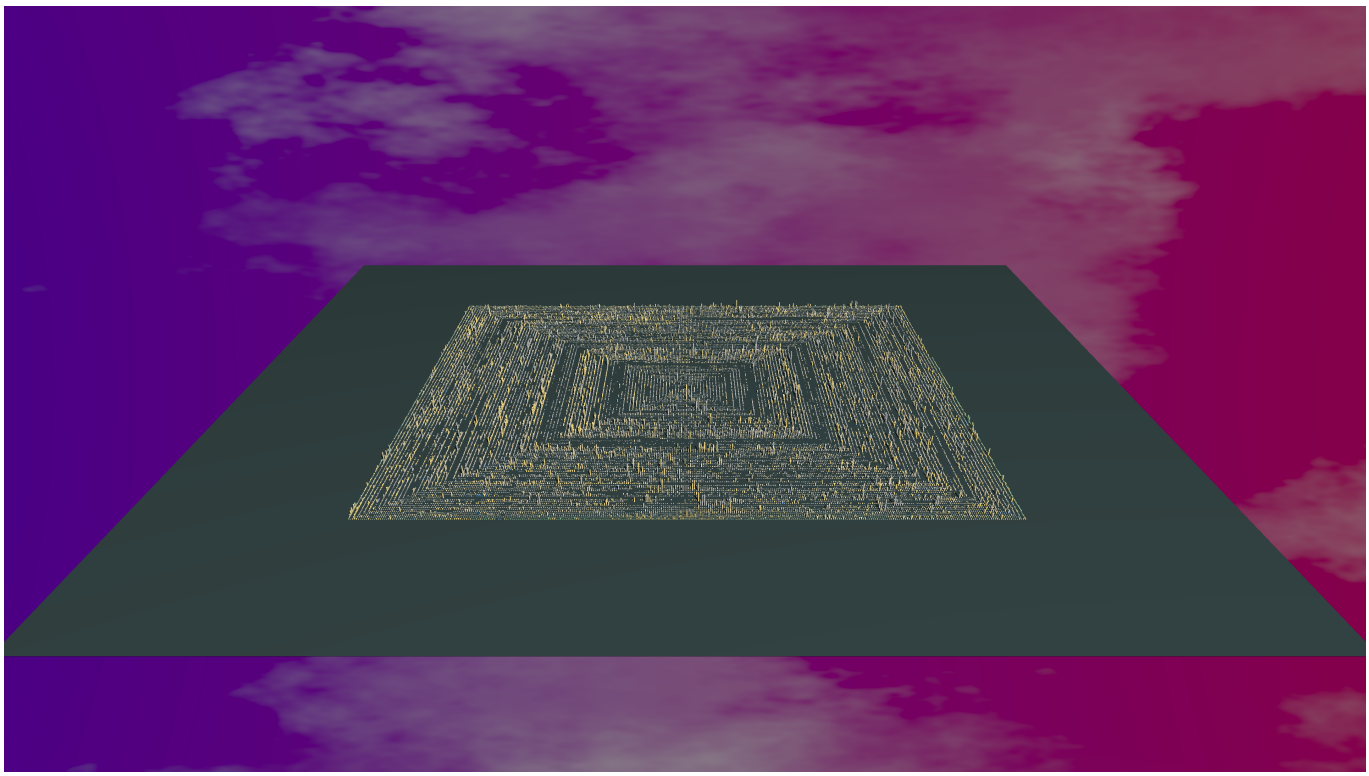


FIGURE E.8: Linux in April 2013

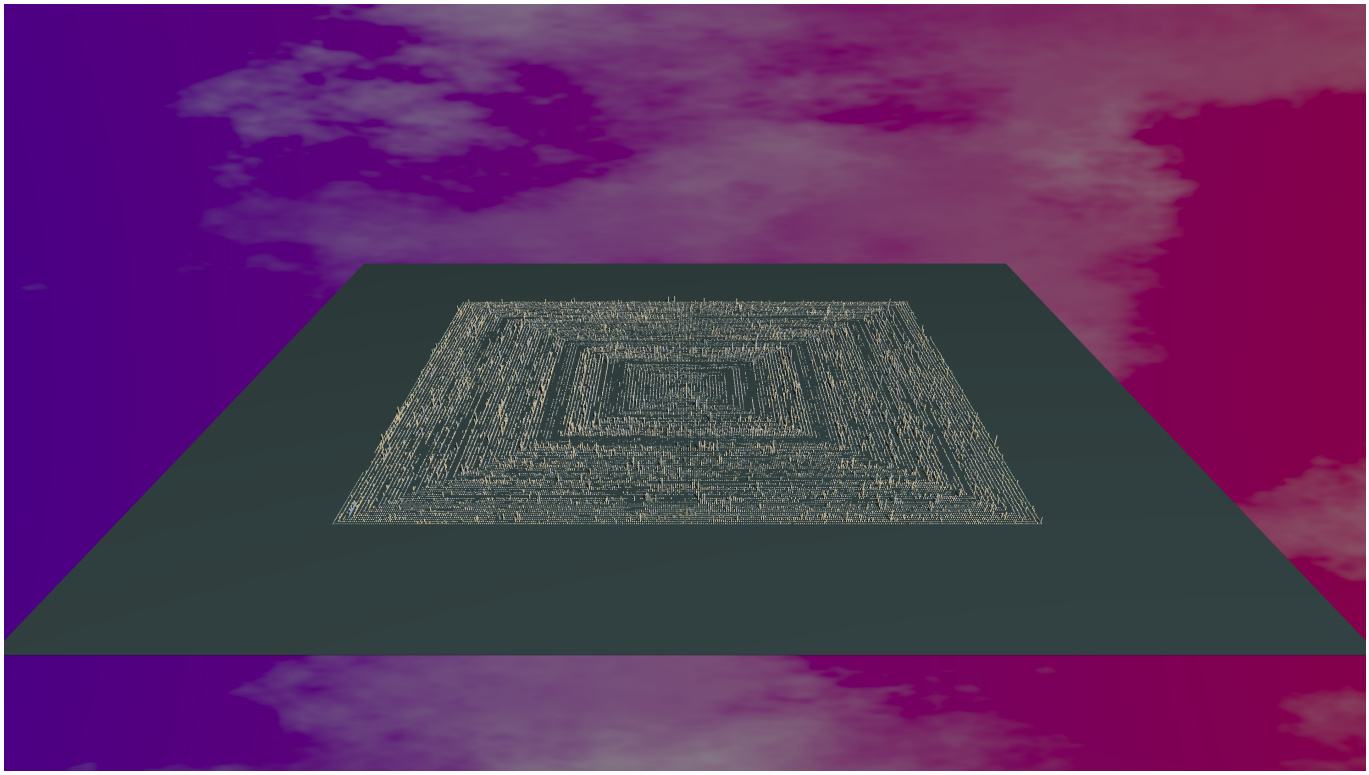


FIGURE E.9: Linux in April 2014

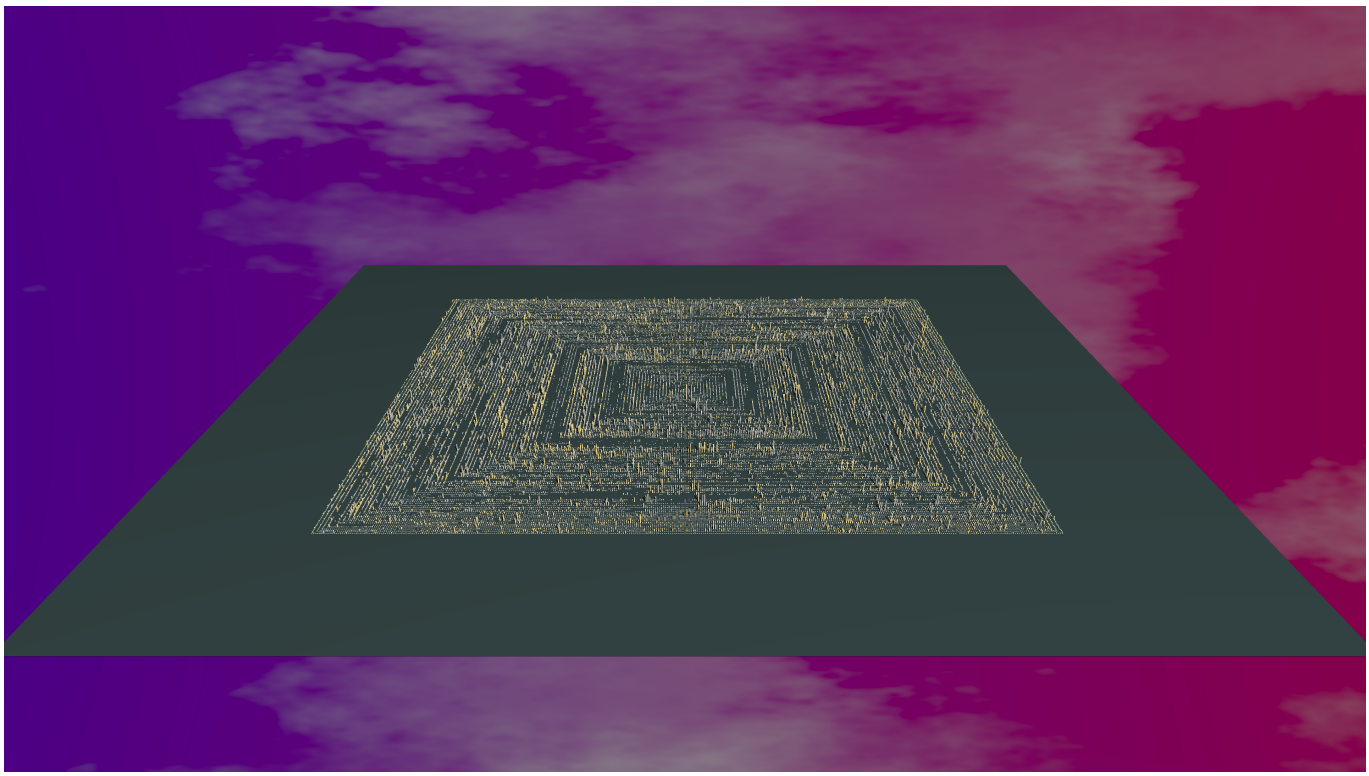


FIGURE E.10: Linux in April 2015

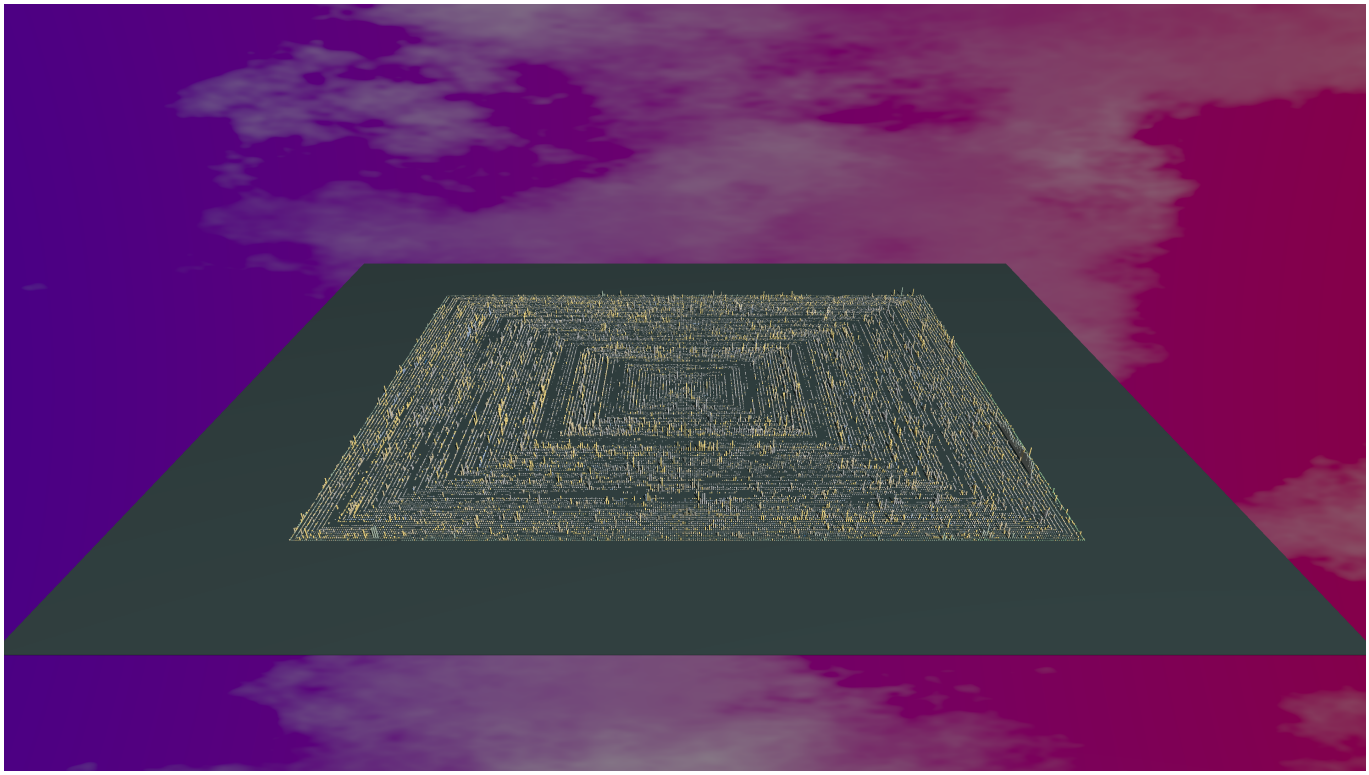


FIGURE E.11: Linux in April 2016

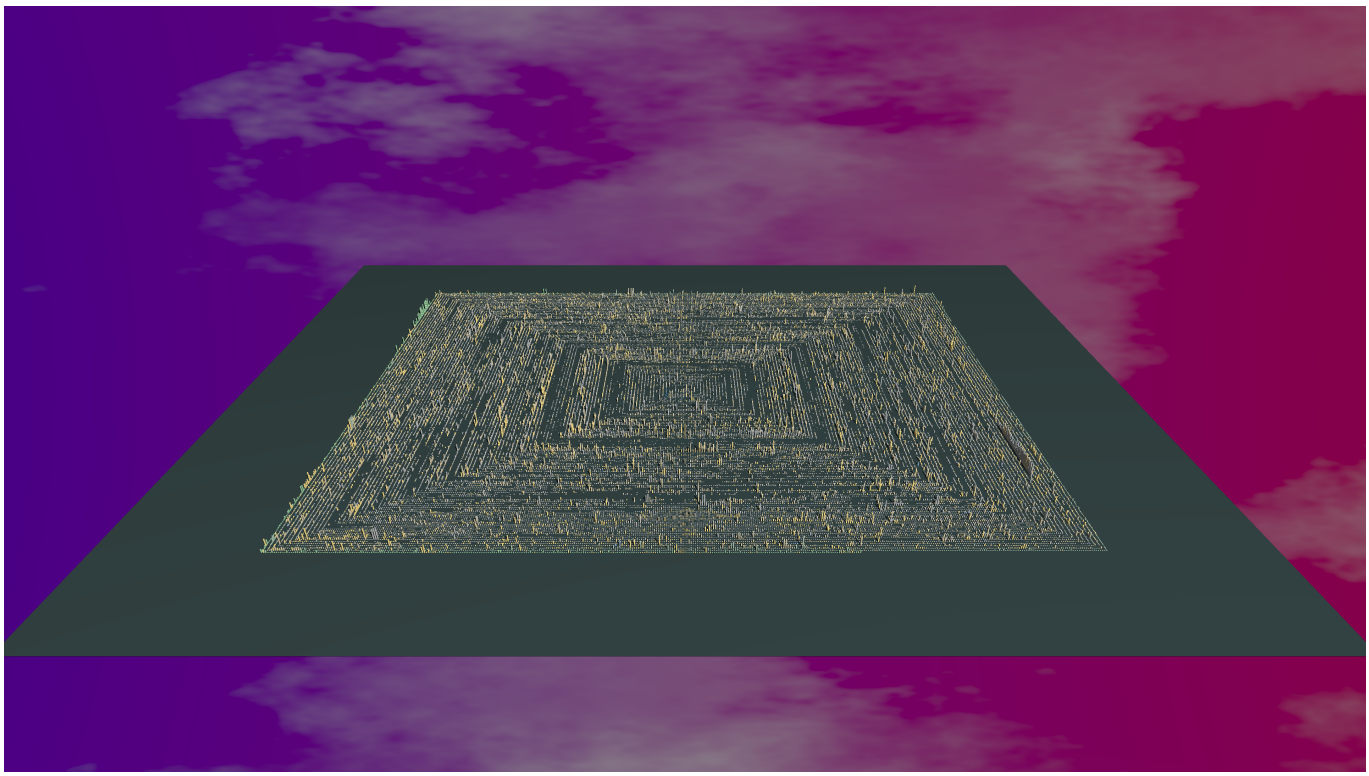


FIGURE E.12: Linux in April 2017

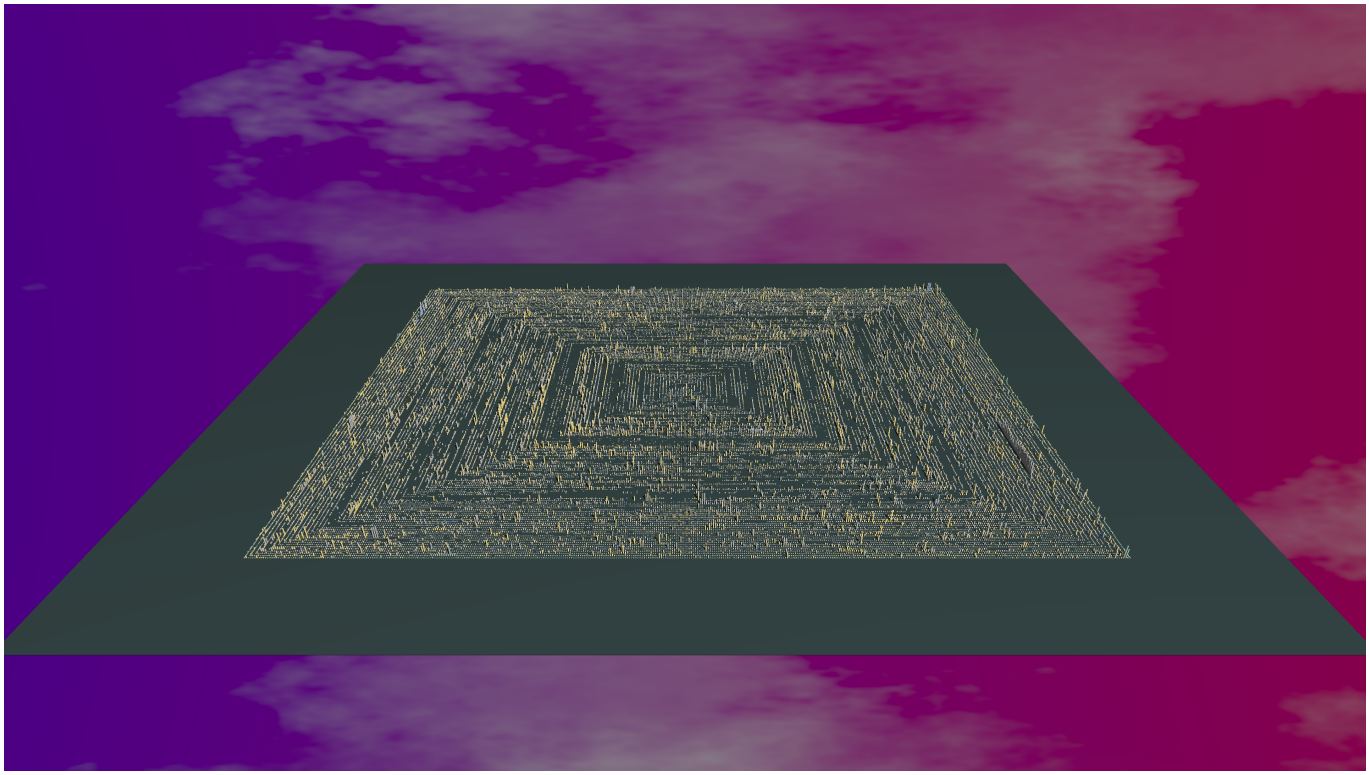


FIGURE E.13: Linux in April 2018

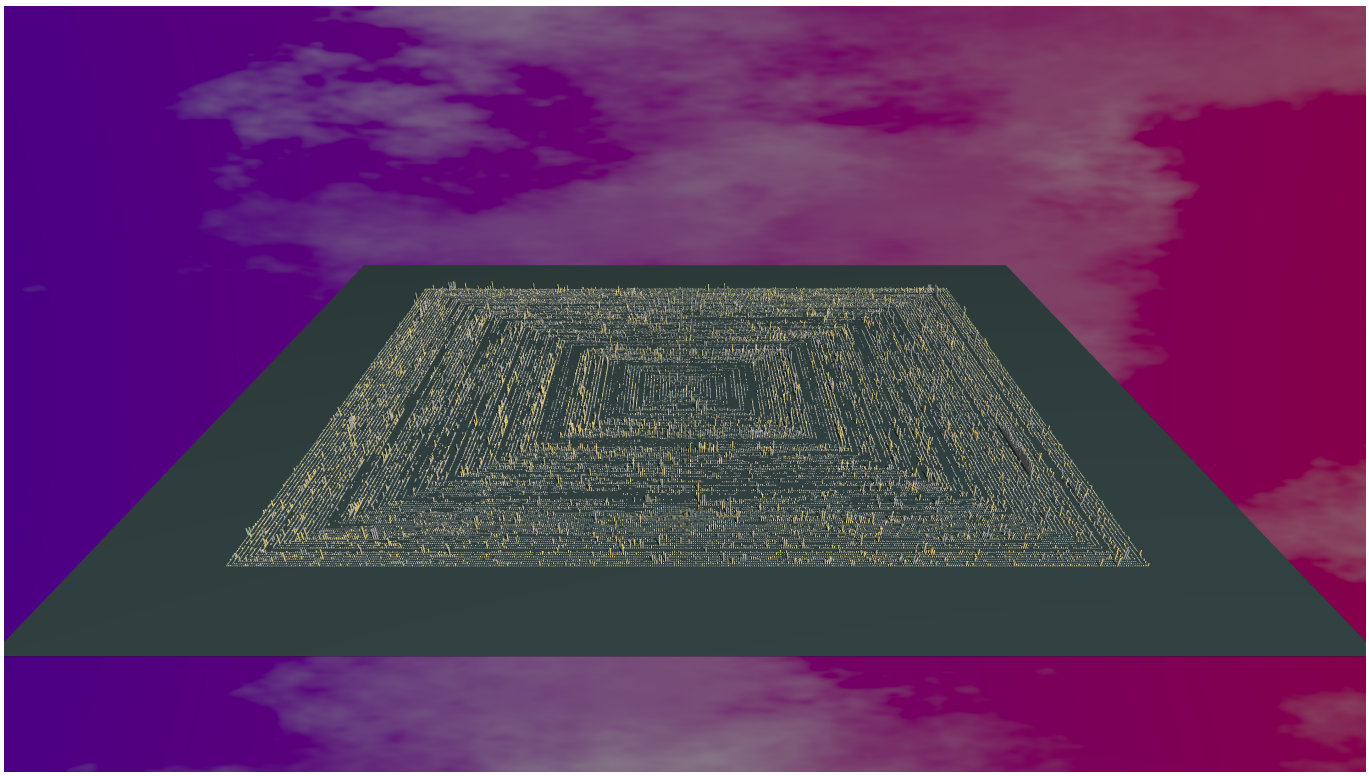


FIGURE E.14: Linux in April 2019

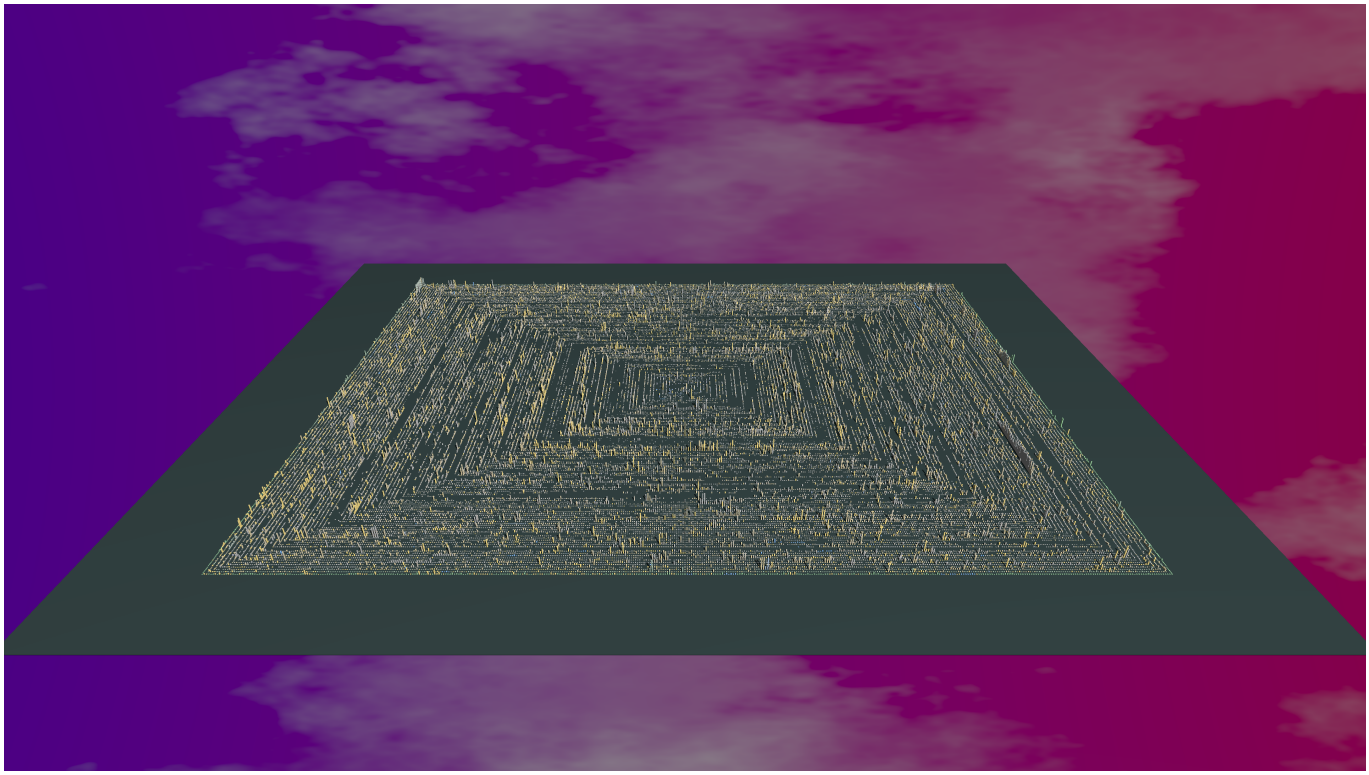


FIGURE E.15: Linux in April 2020

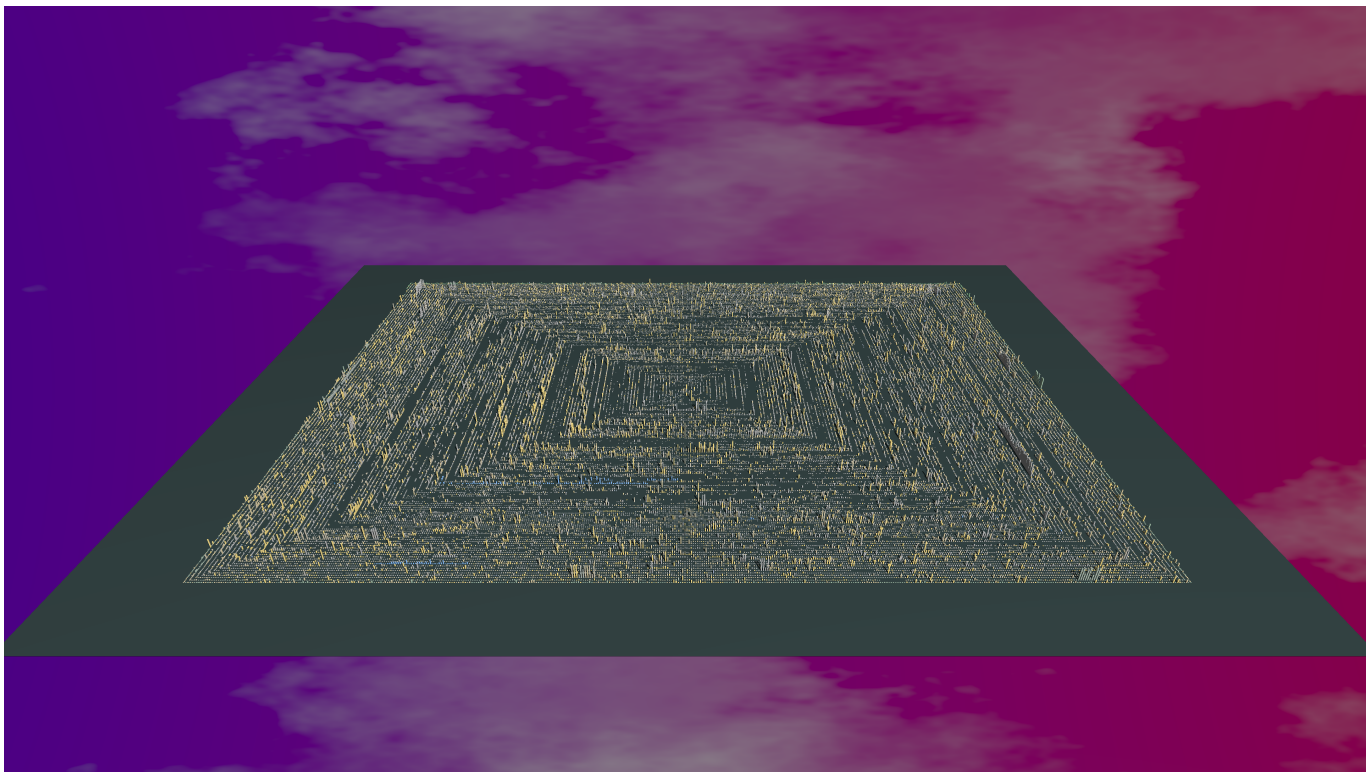


FIGURE E.16: Linux in April 2021

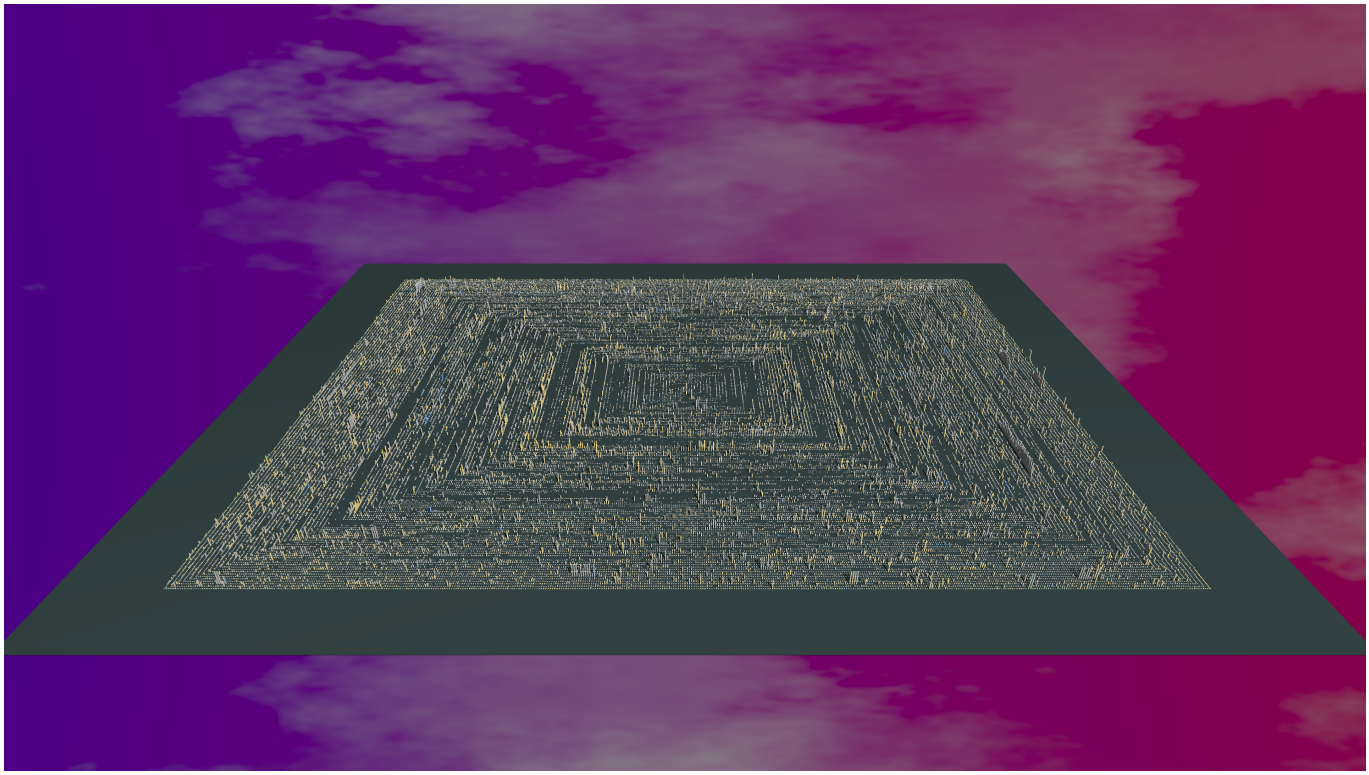


FIGURE E.17: Linux in April 2022

Bibliography

[1] C. V. Alexandru, S. Proksch, P. Behnamghader, and H. C. Gall. Evo-clocks: Software evolution at a glance. In *2019 Working Conference on Software Visualization (VISSOFT)*, pages 12–22, 2019.

[2] J. L. Alty. Computer-human communication? In *People and Computers X: Proceedings of the HCI'95 Conference*, volume 10, page 409. Cambridge University Press, 1995.

[3] S. Boccuzzo and H. C. Gall. Cocoviz with ambient audio software exploration. In *2009 IEEE 31st International Conference on Software Engineering*, pages 571–574, 2009.

[4] M. H. Brown. Exploring algorithms using balsa-ii. *Computer*, 21(5):14–36, may 1988.

[5] D. M. Butler, J. C. Almond, R. D. Bergeron, K. W. Brodlie, and R. B. Haber. Visualization reference models. In *Proceedings of the 4th Conference on Visualization '93*, VIS '93, pages 337–342, USA, 1993. IEEE Computer Society.

[6] M. Chuah and S. Eick. Information rich glyphs for software management data. *IEEE Computer Graphics and Applications*, 18(4):24–29, 1998.

[7] T. Clem and P. Thomson. Static analysis at github: An experience report. *Queue*, 19(4):42–67, aug 2021.

[8] T. A. Corbi. Program understanding: Challenge for the 1990s. *IBM Systems Journal*, 28(2):294–306, 1989.

[9] M. D'Ambros and M. Lanza. Software bugs and evolution: a visual approach to uncover their relationship. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 10 pp.–238, 2006.

[10] M. D'Ambros, M. Lanza, and M. Lungu. The evolution radar: Visualizing integrated logical coupling information. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 26–32, New York, NY, USA, 2006. Association for Computing Machinery.

[11] A. M. Davis. *201 Principles of Software Development*. McGraw-Hill, Inc., USA, 1995.

[12] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, pages 326–337, New York, NY, USA, 1993. Association for Computing Machinery.

[13] C. DiGiano and R. M. Baecker. Program auralization: sound enhancements to the programming environment. In *Program auralization: sound enhancements to the programming environment*, 1992.

[14] C. J. DiGiano, R. M. Baecker, and R. N. Owen. Logomedia: A sound-enhanced programming environment for monitoring program behavior. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 301–302, New York, NY, USA, 1993. Association for Computing Machinery.

[15] E. W. Dijkstra. *Ewd316: A short introduction to the art of programming*. Technische Hogeschool, 1971.

[16] S. G. Eick, J. L. Steffen, and E. E. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, nov 1992.

[17] B. Ens, D. Rea, R. Shpaner, H. Hemmati, J. E. Young, and P. Irani. Chronotwigger: A visual analytics tool for understanding source and test co-evolution. In *2014 Second IEEE Working Conference on Software Visualization*, pages 117–126, 2014.

[18] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[19] J.-D. Fekete, J. van Wijk, J. Stasko, and C. North. *The Value of Information Visualization*, volume 4950, pages 1–18. Springer Berlin, Heidelberg, 07 2008.

[20] T. Gîrba and S. Ducasse. Modeling history to analyze software evolution. *J. Softw. Maintenance Res. Pract.*, 18(3):207–236, 2006.

[21] G. J. Greene, M. Esterhuizen, and B. Fischer. Visualizing and exploring software version control repositories using interactive tag clouds over formal concept lattices. *Information and Software Technology*, 87:223–241, 2017.

[22] L. M. Haibt. A program to draw multilevel flow charts. In *Papers Presented at the the March 3-5, 1959, Western Joint Computer Conference*, IRE-AIEE-ACM '59 (Western), pages 131–137, New York, NY, USA, 1959. Association for Computing Machinery.

[23] J. Hueras and H. Ledgard. An automatic formatting program for pascal. *SIGPLAN Not.*, 12(7):82–84, jul 1977.

[24] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., USA, 1999.

[25] D. H. Jameson. Sonnet: Audio-enhanced monitoring and debugging. In *SANTA FE INSTITUTE STUDIES IN THE SCIENCES OF COMPLEXITY-PROCEEDINGS VOLUME-*, volume 18, pages 253–253. ADDISON-WESLEY PUBLISHING CO, 1994.

[26] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining github. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101, New York, NY, USA, 2014. Association for Computing Machinery.

[27] P. Kapec, G. Brndiarová, M. Gloger, and J. Marák. Visual analysis of software systems in virtual and augmented reality. In *2015 IEEE 19th International Conference on Intelligent Engineering Systems (INES)*, pages 307–312, 2015.

[28] P. Khaloo, M. Maghoumi, E. Taranta, D. Bettner, and J. Laviola. Code park: A new 3d code visualization tool. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 43–53, 2017.

[29] M. Kleiner, b.-i. dalenbäck, and p. svensson. auralization-an overview. *journal of the audio engineering society*, 41(11):861–875, november 1993.

[30] C. Knight and M. Munro. Virtual but visible software. In *2000 IEEE Conference on Information Visualization. An International Conference on Computer Visualization and Graphics*, pages 198–205, 2000.

[31] D. E. Knuth. Computer-drawn flowcharts. *Commun. ACM*, 6(9):555–563, sep 1963.

[32] G. Langelier, H. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 214–223, New York, NY, USA, 2005. Association for Computing Machinery.

[33] M. Lanza. The evolution matrix: Recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, New York, NY, USA, 2001. Association for Computing Machinery.

[34] M. Lanza. Object-oriented reverse engineering coarse-grained, fine-grained, and evolutionary software visualization, 2003.

[35] M. M. Lehman and L. A. Belady. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., USA, 1985.

[36] R. E. Lopez-Herrejon, S. Illescas, and A. Egyed. A systematic mapping study of information visualization for software product line engineering. *Journal of software: evolution and process*, 30(2):e1912, 2018.

[37] A. Mancino and G. Scanniello. Software musification. In *2017 21st International Conference Information Visualisation (IV)*, pages 127–132, 2017.

[38] S. McIntosh, K. Legere, and A. E. Hassan. Orchestrating change: An artistic representation of software evolution. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 348–352, 2014.

[39] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. Cityvr: Gameful software visualization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 633–637, 2017.

[40] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz. A systematic literature review of software visualization evaluation. *Journal of Systems and Software*, 144:165–180, 2018.

[41] H. A. Müller and K. Klashinsky. Rigi-a system for programming-in-the-large. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, pages 80–86, Washington, DC, USA, 1988. IEEE Computer Society Press.

[42] I. Nassi and B. Shneiderman. Flowchart techniques for structured programming. *SIGPLAN Not.*, 8(8):12–26, aug 1973.

[43] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228. IEEE, 2007.

[44] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. In *Proceedings of the 2005 ACM Symposium on Software Visualization*, SoftVis '05, pages 67–75, New York, NY, USA, 2005. Association for Computing Machinery.

[45] J. Ratzinger, M. Fischer, and H. Gall. Evolens: lens-view visualizations of evolution data. In *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, pages 103–112, 2005.

[46] V. Salis and D. Spinellis. Repofs: File system view of git repositories. *SoftwareX*, 9:288–292, 2019.

[47] T. Schneider, Y. Tymchuk, R. Salgado, and A. Bergel. Cuboidmatrix: Exploring dynamic structural connections in software components using space-time cube. In *2016 IEEE Working Conference on Software Visualization (VISSOFT)*, pages 116–125, 2016.

[48] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[49] I. Sommerville. *Software Engineering (5th Ed.)*. Addison Wesley Longman Publishing Co., Inc., USA, 1995.

[50] D. Sonnenwald, B. Gopinath, G. Haberman, W. Keese, and J. Myers. Infosound: an audio aid to program comprehension. In *Twenty-Third Annual Hawaii International Conference on System Sciences*, volume 2, pages 541–546 vol.2, 1990.

[51] D. Spadini, M. Aniche, and A. Bacchelli. Pydriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2018, pages 908–911, New York, NY, USA, 2018. Association for Computing Machinery.

[52] F. Steinbrückner and C. Lewerentz. Representing development history in software cities. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 193–202, New York, NY, USA, 2010. Association for Computing Machinery.

[53] C. M. B. Taylor and M. Munro. Revision towers. In *Proceedings of the 1st International Workshop on Visualizing Software for Understanding and Analysis*, VISSOFT '02, page 43, USA, 2002. IEEE Computer Society.

[54] S. D. Tom Mens. *Software evolution*. Springer, Berlin, 2008.

[55] P. Vickers. External auditory representations of programs: Past, present, and future an aesthetic perspective. In *ICAD 2004*, 01 2004.

[56] P. Vickers and J. L. Alty. Siren songs and swan songs debugging with music. *Commun. ACM*, 46(7):86–93, jul 2003.

[57] A. Von Mayrhauser and A. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[58] R. C. Waters. User format control in a lisp prettyprinter. *ACM Trans. Program. Lang. Syst.*, 5(4):513–531, oct 1983.

[59] R. Wettel and M. Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, 2007.

[60] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: a controlled experiment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 551–560, 2011.

[61] P. Young and M. Munro. Visualising software in virtual reality. In *Proceedings. 6th International Workshop on Program Comprehension. IWPC'98 (Cat. No.98TB100242)*, pages 19–26, 1998.