Visualization of Source Code Similarity using 2.5D Semantic Software Maps

Daniel Atzberger, Tim Cech, Willy Scheibel,
Daniel Limberger, and Jürgen Döllner

Hasso Plattner Institute, Digital Engineering Faculty, University of Potsdam

Abstract. For various program comprehension tasks, software visualization techniques can be beneficial by displaying aspects related to the behavior, structure, or evolution of software. In many cases, the question is related to the semantics of the source code files, e.g., the localization of files that implement specific features or the detection of files with similar semantics. This work presents a general software visualization technique for source code documents, which uses 3D glyphs placed on a two-dimensional reference plane. The relative positions of the glyphs captures their semantic relatedness. Our layout originates from applying Latent Dirichlet Allocation and Multidimensional Scaling on the comments and identifier names found in the source code files. Though different variants for 3D glyphs can be applied, we focus on cylinders, trees, and avatars. We discuss various mappings of data associated with source code documents to the visual variables of 3D glyphs for selected use cases and provide details on our visualization system.

Keywords: Source Code Mining · Software Visualization · Glyph Visualization.

1 Introduction

About 90 % of the entire costs of a software project are related to the maintenance phase [14], i.e., to prevent problems before they occur (preventive maintenance), correct faults (corrective maintenance), improve the functionality or performance (perfective maintenance), or adapt to a changing environment (adaptive maintenance) [23]. There are various visualization techniques to represent aspects related to the structure, the behavior, or the evolution of the underlying software, to assist users in program comprehension tasks during the maintenance phase. Nevertheless, since software has no intrinsic gestalt, software visualization uses suitable *abstractions and metaphors* to depict aspects of and relations within software data to support and, at best, align users in their mental representation of selected software aspects. Interactive visualizations allow users to analyze a software project in an exploratory way and thus support finding information and gaining knowledge. Examples for well-established, interactive software visualization techniques are:

– *Icicle Plots* for representations of trace executions [36,10],

- *Treemaps* depicting the hierarchical structure of software projects [41,32],
- *Circular Bundle Views* illustrating relations, e.g., include dependencies [11],
- *Software Cities* that reflect the development history of software [45,46], and
- similar approaches based on cartographic metaphors [21,28].

Many specific questions in maintenance are related to the semantic structure of software projects. For example, in the case of perfective maintenance, source code files implementing a specific functionality or concept need to be identified. It is helpful to be aware of other files that share semantics in this context. Such tasks can become intensively time-consuming with long-lasting software systems and with an increasing number of different developers. In order to support such tasks, various layouts exist that can reflect semantic similarities between files, i.e., by placing files with a similar semantic closer to one another [27,28,2,4]. Using 2D or 3D glyphs to represent files with a semantic positioning and additional data mapping, e.g., software metrics mapped to the glyphs' visual variables, facilitates the comprehension of the semantic structure of a software project. For the remainder of this work, we refer to the term *glyphs* as defined by Ward et al.; "In the context of data and information visualization, a glyph is a visual representation of a piece of data or information where a graphical entity and its attributes are controlled by one or more data attributes" [51].

In this work, we present a general approach for placing custom 3D glyphs in a 2D reference space for software visualization tasks, in order to (1) capture the semantic structure of source code files and (2) allow for an additional, inherent visual display of related data, e.g., software metrics. For our layout technique, we assume developer comments and deliberately chosen identifiers to not only provide instructions for compilers but to simultaneously document and communicate intent, function, and context to developers. This assumption motivates the use of techniques from the Natural Language Processing (NLP) domain for mining the semantic structure of source code documents. We apply Latent Dirichlet Allocation (LDA), a probabilistic topic model, to capture the semantic structure of a software structure, which leads to a mathematical description of source code files. By applying Multidimensional Scaling (MDS) as a dimension reduction technique, we generate a two-dimensional layout that reflects the semantic relatedness between the source code files. We represent every source code unit or file as a single 3D glyph. Though plenty of glyphs and metaphors have been applied to software visualization tasks, we focus our discussion on three examples we considered valuable:

Cylinders with their extent, height, and color as visual variables.

Trees with a variety of visual variables, e.g., size, type, leaf color, health, age, and season.

Avatars which can be easily distinguished from each other and clearly identified, e.g., for depicting software developers or teams.

We describe fitting use cases for every glyph and provide examples using popular Open Source projects data. Figure 1 shows one exemplary result of our visualization approach.

Fig. 1. Example of a *Software Forest* using handcrafted tree models from *SketchFab* (sketchfab.com) as 3D glyphs. Each tree represents a source code file. Quantitative and qualitative data associated to the files can be mapped to age, type, and health of a tree.

The remainder of this work is structured as follows: In section 2 we review existing work related to our approach. We provide an overview of possible layouts for visualizing source code and glyphs and natural metaphors in the software visualization domain. In Section 3 we detail the layout approach, which is based on LDA and MDS and applied to comments and identifiers in source code. Section 4 describes use case scenarios and shows how data related to the semantics of source code files can be represented. We further present a detailed explanation of our system and its implementation in Section 5 and, finally, conclude this paper in section 6 and present directions for future work.

2 Related Work

Our visualizations are created in two distinct steps. First, we generate a semantic layout that is then used for placing 3D glyphs (representing source code files). Second, we map quantitative and qualitative data of source code files to the available visual variables of the 3D glyphs. With respect to the prior art, we, therefore, focus on these two aspects. We describe existing approaches for placing documents in a reference space in order to reflect their semantic similarity and also describe existing 2.5D approaches based on treemaps for software visualization tasks. We then present relevant work on three widely used visualization metaphors, namely the island metaphor, the tree metaphor, and the city metaphor. Selected glyphs are presented at the end of this section.

Semantic Layouts for Software Visualization. When designing visualizations, one has to consider the placement of data items in the reference space. In the case of document visualization, we call a layout whose goal is to reflect the semantic relatedness between the data items a semantic layout. In a semantic layout,

documents that share a common similarity are placed nearby each other. As documents are mostly viewed as Bag-of-Words (BOW), i.e., the order of words within a document is neglected, and only their frequency is taken into account, dimension reduction techniques are used to project the high-dimensional points to a two-dimensional plane or a three-dimensional space.

Skupin et al. proposed an approach for generating two-dimensional visualizations for text documents using cartographic metaphors [44]. The authors applied Self-Organizing Map (SOM) on the BOW [26], as dimension reduction technique, to place abstracts of publications about geography on the plane. Furthermore, dominant terms were displayed, thus showing the semantic content of the region in the visualization.

Kuhn et al. were the first to propose a semantic layout for software visualization tasks [27,28]. First, each source code file is considered as a single document and several preprocessing tasks are undertaken to remove noise from the vocabulary. Then, the high-dimensional BOW is reduced in their dimension in two steps. The topic model Latent Semantic Indexing (LSI) [13] is applied, which describes each document through its expression in the latent topics within a software project, which can already be seen as a dimension reduction of the BOW. After this, MDS [12] is applied on the dissimilarity matrix that captures the pairwise dissimilarities of the documents using the cosine-similarity. The resulting two-dimensional scatterplot is then equipped with height lines, resulting in a cartographic visualization. In addition, two-dimensional glyphs are placed for displaying coding activities, e.g., test tubes.

Linstead et al. [34,35] were the first to propose a semantic software layout based on LDA and its variant, the Author-Topic Model (ATM), which additionally takes information about authorship into account [39]. By applying the topic models on the source code of the Eclipse project, both source code files and authors are described as distributions over latent topics. The final layout is computed by applying MDS on the dissimilarity matrix, which contains the pairwise symmetrized Kullback-Leibler divergence of the authors or files.

Another approach that models the semantic structure of source code files using LDA for visualization tasks was presented by Atzberger et al. [2]. In their approach, the authors first apply MDS on the topic-word distributions to compute two-dimensional vertices, representing the topics, as presented in [43]. The position of a document is then computed as a convex linear combination according to its document-topic distribution. Using this layout, the authors introduced the tree metaphor for software visualization, resulting in the so-called *Software Forest*. In a later work, Atzberger et al. discussed the use of pawns and chess figures as 3D glyphs for visualizing the knowledge distribution across software development teams [3]. In this case, the layout reflects the semantic similarity between developers, additional information about the expertise of each developer can then be mapped on the visual variables of the representing glyph.

In another work, Atzberger et al. applied their layout approach to a 3D reference space, creating a stylized scatter plot for the depiction of software projects [4]. Inspired by a metaphor introduced by Lanza et al. [29], the authors

displayed each source code file as a star, thereby creating a *Software Galaxy*. The authors also introduced transparent volumetric nebulae to make use of the metaphor of galactic star clusters or nebulae. Attributes such as cluster density or distribution can subsequently be mapped to the nebulaes' intensities and colors.

The Island Metaphor. The 2.5D approach used by Atzberger allows for the integration of a terrain (based on a dynamically generated heightfield), resulting in visualizations resembling islands. Indeed the island metaphor is a widely used visualization metaphor in the Software Visualization domain. Štěpánek developed *Helveg*, a framework for visualizing C# code as islands, based on a graph-drawing algorithm layout [47]. Their approach also uses 3D glyphs, e.g., bridges representing dependencies and trees depicting classes, for visualizing the structure of a project. *CodeSurveyor* is another approach that makes use on the cartographic metaphor [21]. Based on a hierarchical graph layout algorithm, files are positioned in a 2D reference plane and are aggregated to states, countries, or continents according to the architectural structure of the software project. *CodeSurveyor* shares characterisitcs of treemaps that use non-rectangular shapes [40]. Schreiber et al. proposed *ISLANDVIZ*, another approach using the island metaphor. It enables users to interactively explore a software system in virtual reality and augmented reality alike [42].

Treemap Layouts. Another widely used class of layout algorithms in the software visualization domain are *Treemaps*. Treemaps are inherently capable of reflecting the typically hierarchical structure of software projects [41,40]. Given their 2D layout, they can be extended into the third dimension, thus resulting in a 2.5D visualization. Besides height, color, and texture 2.5D treemaps offer additional visual variables for additional information display [32,31,33]. An approach that refer to natural phenomena, e.g., fire or rain, for visualizing software evolution in a 2.5D treemap was proposed by Würfel et al. [54]. It is worth mentioning that the class of treemap algorithms includes a large number of shapes other than just rectangles or Voronoi cells [41].

The Tree Metaphor. In our considerations, we use the tree metaphor since trees offer a variety of visual variables. Kleiner and Hartigan were the first to propose a mapping of multivariate data to a 2D tree [25]. Based on hierarchical clustering of variables, for each data point, the geometry of each tree, i.e., the thickness of a branch, the angle between branches, and their orientation, is derived from the data attributes. Erra presented an approach to visualize object-oriented systems using the tree metaphor, thus resulting in a forest [16,17]. For every revision each file is depicted as a tree, whose visual variables reflect properties of the source code, i.e., software metrics, in a predefined way. Later Atzberger et al. applied the tree metaphor for software visualization tasks [2]. The main difference between Atzberger et al. and Erra et al. is the placement of the trees in the reference plane. The approach of Atzberger et al. is not restricted to the case of object-oriented programming languages, as it only uses the natural language in source code. Furthermore, the system of Atzberger et al. allows users to specify

custom mappings of data and visual attributes. The authors do not focus on rendering realistic trees but rather apply handcrafted models for their approach. Kleiberg et al. use the tree metaphor for visualizing an entire set of hierarchically structured data. This approach differs from most other approaches because each tree does not represent a single document [24].

The City Metaphor. This metaphor is probably the most popular use of 3D glyphs in the software visualization domain. In their approach *CodeCity*, Wettel and Lanza applied a city metaphor for exploring object-oriented software projects using a 2.5D visualization, referring to real-world cities [52,53]. Each class is represented by a building and packages are grouped into districts that are placed according to a modified treemap algorithm. By mapping software metrics onto the visual variables of the cuboids, e.g., its height and the size of the base, a user can get an overview of the structure of a project. Steinbrückner adopted the idea of the city metaphor and introduced a novel layout approach, based on a hierarchical street system, that captures a project development over time [45,46].

Other Approaches. Beck proposed a mapping between software metrics of object-oriented software projects and geometric properties of figurative feathers, e.g., its size, shape, and texture [5]. Their approach *Software Feathers* is intended to support developers in getting a first overview of a software project and to deticting interesting code entities. Fernandez et al. extended an approach by Lewis et al. [30], that generates 2D glyphs in order to identify classes with the same dependencies and similar set of methods [19]. Chuah and Eick proposed the three glyph visualizations *InfoBUG*, *Time-wheel*, and *3D-wheel* for the task of visualizing project-oriented software data [9].

3 Glyph Placement in a 2D Reference Space

According to Ward et al. there are three general strategies for placing glyphs [51]:

1. **uniform** All glyphs are placed in equidistant positions.
2. **structure-driven** The positions of the glyphs arise from the structure of the data set, e.g., a hierarchy or graph structure within the data.
3. **data-driven** The positions of the glyphs are determined by a set of data attributes.

In this section, we present the layout approach presented by Atzberger et al. [2]. In a semantic layout, the relative position between two points on the 2D reference plane should reflect the semantic relatedness between the corresponding data points, i.e., source code files of a software project. For this, the assumption is made that the semantic similarity between source code files is reflected in a shared vocabulary and can therefore be captured using techniques from the NLP domain. The approach for placing 3D glyphs on a plane has three stages. First, the source code files of a software project are preprocessed to get rid of words that carry no semantic information. In the second step LDA is applied to the corpus

of preprocessed documents to model each source code file as a high-dimensional vector. Lastly, in the third step, MDS is applied to reduce the vectors in their dimensionality.

3.1 Data Preprocessing

In our considerations each source code file of a project is viewed as a single document, the set of all documents is called the corpus, and the set of all words in the corpus forms the vocabulary. We neglect the ordering of the words within a document and only store their frequencies in the so-called term-document-matrix. In order to remove words from the vocabulary that carry no semantic information, e.g., stopwords of the natural language, it is necessary to perform several preprocessing steps before applying topic models. Moreover, source code often follows naming conventions, e.g., the Camel Case convention, thus requiring additional preprocessing steps [7]. In our experiments, the following sequence of preprocessing steps has turned out to produce a usable vocabulary [2].

1. **Removal of Non-text Symbols:** All special characters such as dots and semicolons are replaced with white spaces to avoid accidental connection of words not meant to be combined. This includes the splitting of identifier names, e.g., the word *foo.bar* gets split into *foo* and *bar*.
2. **Split of Words:** Identifiers are split according to delimiters and the Camel Case convention, e.g., *FooBar* is split into *foo* and *bar*, and stripped from redundant white space subsequently.
3. **Removal of Stop Words:** Stop words based on natural language and programming language keywords are removed as they carry no semantic content. Additionally, we filter the input based on a hand-crafted list comprising domain-specific stop words, e.g., data types and type abstractions.
4. **Lemmatization:** To avoid grammatical diversions, all words are reduced to their basic form, e.g., *said* and *saying* are reduced to *say*.

After applying the four preprocessing steps, we store each document as a BOW. For the remainder of this paper, we refer to a documents' BOW after preprocessing as a document.

3.2 Latent Dirichlet Allocation on Source Code

Topic models are a widely used class of techniques for investigating collections of documents, e.g., for knowledge comprehension or classification tasks [1]. For software engineering tasks, LDA proposed by Blei et al. [6], is the most common technique [7]. Assuming a set of documents $\mathcal{D} = \{d_1, \ldots, d_m\}$, the so-called *corpus*, LDA extracts latent topics $\varphi_1, \ldots, \varphi_K$, underlying the corpus, where the number of topics K is a hyperparameter of the model. As topics are given as multinomial distributions over the vocabulary $\mathcal{V}$, which contains the terms of the corpus $\mathcal{D}$, the "concept" underlying a topic, in most cases can be derived from its most probable words. Table 1 shows an example for three topics with

their ten most probable words extracted from the Bitcoin project [2]. From the most probable words, we suggest that topic #1 deals with the internal logic of cryptocurrency. Words like “thread”, “time”, “queue”, and “callback” are related to the general concept of parallel processing in C++, and topic #3 is concerned about the UI.

Besides the topics, LDA learns representations $\theta_1, \ldots, \theta_m$ of the documents as distributions over the topics. The distributions $\theta_1, \ldots, \theta_m$ therefore capture the semantic structure of the documents and allow a comparison between them on a semantic level. LDA makes the assumption of an underlying generative process, which is given by

1. For each document d in the corpus $\mathcal{D}$ choose a distribution over topics $\theta \sim \text{Dirichlet}(\alpha)$
2. For each word w in d
   (a) Choose a topic $z \sim \text{Multinomial}(\theta)$
   (b) Choose the word w according to the probability $p(w|z, \beta)$

The parameter $\alpha = (\alpha_1, \ldots, \alpha_K)$, where $0 < \alpha_i$ for all $1 \leq i \leq K$, is the Dirichlet prior for the document-topic distribution. Its meaning is best understood, when written as the product $\alpha = a_c \cdot m$ of its concentration parameter $a_c \in \mathbb{R}$ and its base measure $m = (m_1, \ldots, m_k)$, whose components sum up to 1. The case of a base measure $m = (1/K, \ldots, 1/K)$ is denoted as symmetrical Dirichlet prior. For small values of a_c, the Dirichlet distribution would favor points in the simplex that are close to one edge, i.e., LDA would try to describe a document with a minimum of topics. The larger the value of a_c the more likely that LDA is to fit all topics a non-zero probability for a document. Analogous, those considerations hold true for the Dirichlet prior $\beta = (\beta_1, \ldots, \beta_N)$, $0 < \beta_i$ for $1 \leq i \leq N$ for the topic-term distribution, where N denotes the size of the vocabulary $\mathcal{V}$.

Since inference for LDA is intractable, approximation techniques need to be taken into account [6]. Among the most widely used are Collapsed Gibbs Sampling (CGS) [20], Variational Bayes (VB) [6], and its online version (OVB) [22].

3.3 Multidimensional Scaling

LDA applied on the source code files leads to a description of each document as a high-dimensional vector, whose components represent the expression in the respective topic. Therefore using a similarity measure, e.g., the Jensen-Shannon divergence, two documents can be compared on a semantic level, thus forming structures, e.g., clusters and outliers, in the set of all documents. Linstead et al. used this notion of similarity and applied the dimension reduction MDS on the documents to generate a two-dimensional layout. However, this approach implicitly assumes that all extracted topics are “equally different” to each other and neglects the fact that the topics, viewed as distributions over the vocabulary, can be compared among each other themselves. The layout approach by Atzberger et al. addresses this issue and applies the dimension reduction technique MDS on the topics $\varphi_1, \ldots, \varphi_K$, which can be compared to each other using the Jensen-Shannon distance [43,2]. This results in points $\bar{\varphi}_1, \ldots, \bar{\varphi}_K \in \mathbb{R}^2$, whose Euclidean

Table 1. Three exemplary topics extracted from *Bitcoin Core*[1] source code with $K = 50$ and the Dirichlet priors set to their default values.

Topic #1		Topic #2		Topic #3	
Term	**Prob.**	**Term**	**Prob.**	**Term**	**Prob.**
std	0.070	thread	0.132	address	0.115
transaction	0.031	time	0.070	model	0.108
fee	0.027	queue	0.064	table	0.065
tx	0.026	std	0.054	label	0.051
ban	0.024	callback	0.040	qt	0.033
str	0.023	run	0.037	index	0.030
handler	0.016	call	0.025	dialog	0.024
output	0.016	mutex	0.021	column	0.024
bitcoin	0.015	scheduler	0.020	ui	0.021
reason	0.015	wait	0.018	role	0.019

distance reflects the Jensen-Shannon-distance of the high-dimensional topics. A document d, given by its document-topic distribution $\theta = (\theta^{(1)}, \ldots, \theta^{(K)})$, is then represented as the convex linear combination $\bar{d}$, precisely

$$\bar{d} = \sum_{j=1}^{K} \theta^{(j)} \bar{\phi}_j. \tag{1}$$

A document with a strong expression in a topic is subsequently placed next to that topic, taking the similarity of topics into account.

4 Visual Attributes of 3D Glyphs and Use Cases

In section 2, we summarized popular visualization metaphors based on 3D glyphs in the Software Visualization domain. In this section, we review (1) the city metaphor and (2) the forest metaphor together with (3) the island metaphor. We categorize our visualization as $\mathcal{A}^3 \oplus \mathcal{R}^2$, i.e., three-dimensional primitives placed on a two-dimensional reference space pl [2,15]. We further present a novel idea of placing avatars into the visualization, thus indicating developer activities.

4.1 City Metaphor

The city metaphor, as proposed by Wettel et al., owes its name its visual similarity to modern cities, caused by displaying software files as cuboids [52]. The motivation for choosing this metaphor was to support a user navigating through a software system by adopting a well-known metaphor from everyday life. All existing approaches rely on a layout that captures the hierarchical structure

[1] Source code taken from github.com/bitcoin/bitcoin

of a software project, thus focusing on the architectural aspect of a project. Therefore, the approaches do not support program comprehension tasks related to the underlying semantic concepts of a software project. One advantage of the city metaphor is that they offer various visual variables. Examples proposed in the literature are:

- Wettel et al. mapped the number of methods to height and the number of attributes to the cuboids' horizontal extent [52].
- Steinbrückner et al. used stacked cylinders, where each cylinder displays the coding activity of a single developer made on a file [46].
- Limberger et al. present various advanced visual variables, e.g., sketchiness or transparency, for cuboids that can be applied for the city metaphor [32]. Recently, Limberger et al. investigated the use of animations for displaying the evolution of source code artifacts measured between two revisions [31].

We adapt the idea of the city metaphor in our considerations by representing each file of a software project as a cylinder. Figure 3 shows a simple example for the project *globjects*, where the height and the color of the cylinders are used as visual variables. The height of the cylinder displays the Lines-of-Code (LOC) of the respective file, thus revealing the impact of underlying concepts for the software project. The color displays the percentage of commented lines in the respective file using a sequential color scheme, which allows drawing conclusions about the code quality in relation to a concept. In most cases, large files are grouped nearby each other, thus indicating their underlying concepts seem to have a large impact on the size of the project in terms of LOC. Furthermore, large files often harbor the risk of non-sufficient documentation in the form of comments. This observation could motivate the project maintainer to focus on that concept, distributed over the individual files, in a future refactoring process.

4.2 Forest Metaphor

As shown in section 2, the idea of forest islands is two widely used metaphors in the Software Visualization domain, especially as they offer a grouping of files according to some "relatedness". Furthermore, islands and forests are real-world structures, thus making them suitable for creating a mental map for the user to support program comprehension tasks. Our presentation here follows the preceding work of Atzberger et al. closely [2]. We focus our discussion to the following set of visual variables:

- The tree height, e.g., for depicting the size of a file in terms of LOC.
- The color of the tree crown, e.g., displaying software metrics related to the quality or complexity of the respective file.
- The tree type, e.g., to distinguish the source code files by their file endings.
- The health status of a tree, e.g., for displaying failed tests.
- Chopped trees, e.g., for visualizing deleted files.

Fig. 2. Examples of two sets of tree glyphs: the top row inherits the visual variables size, color, and health status. The bottom row shows trees of different types. Both models were purchased on *SketchFab*: "HandPainted Pine Pack" by ZugZug and "Low Poly Nature Pack" by NONE.

Figure 2 shows two sets of tree glyphs and demonstrates the visual attributes they inherit [2].

When visualizing a software project using the tree metaphor, we first compute the position of each file as presented in section 3, i.e., each file is represented as a single tree, thus forming an entire forest. Then for each point, the value of a height field is computed as presented in [28]. This has the effect that dense regions are placed on higher ground than regions with fewer trees, but still assuring that single trees stand on a terrain of $30\% - 50\%$ of the maximal height. Furthermore, we integrated the possibility to configure the water height, which can be seen as a height filtering technique.

Figure 4 shows the result of applying our approach on the *notepad-plus-plus* project[3] based on the set of pine trees shown in Figure 2. The underlying visual mapping aimed to represent the document-topic distribution of each file. The tree type is chosen according to the main topic of each file, i.e., the topic that has the highest probability in the file, e.g., documents with the main topic "User Interface Code" are displayed as green pine trees. All trees of the same color are highlighted when hovering over a tree. Here we want to mention that we manually labeled the topics for the project by examining their most probable words. In general, this is a time-consuming task that needs to be done manually [37].

Our next application demonstrates the use of Software Forest for the bitcoin project[1]. In Section 3 we showed three interesting topics extracted from the source code by applying LDA with $K = 50$ topics and default values for the Dirichlet priors. We map the topic with the highest impact for a document onto the tree type. As the number of tree types is usually limited, this visualization approach does not scale for a large number of topics. The bitcoin project is mainly written in C++, C, and Python. We ignore the other source code files

[2] Source code taken from https://github.com/cginternals/globjects
[3] Source code taken from https://github.com/notepad-plus-plus/notepad-plus-plus

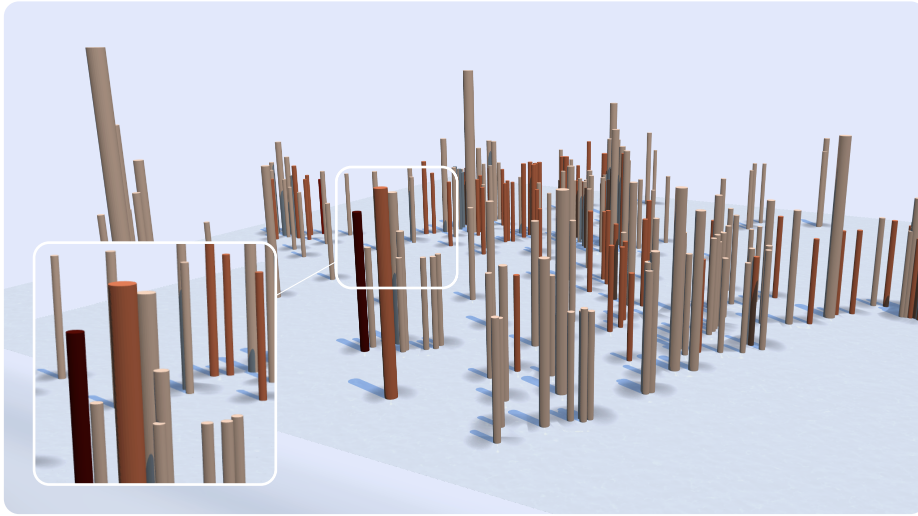

Fig. 3. Example for cylinders placed on a 2D reference plane. Each cylinder represents a single source code file of the project *globjects*[2]. The height displays the LOC of each file, and the color the percentage of commented lines.

and map the programming language onto the color of the tree. The height of the tree captures the LOC for the respective file (Figure 5).

4.3 Developer Avatars

Our last visualization metaphor uses 3D glyphs displaying people for showing coding activities within a software project. Each developer or team is assigned an avatar whose position shows the source code contributed to within a given timespan. One question that could be addressed with this visualization would be the assignment of suitable developers when a bug related to a concept or a file would occur. Figure 6 shows an example for placing avatars and cylinders on an island for the example of the *globjects* project. The color of the cylinders displays a complexity metric, whereas each figure represents a single developer. The avatars are placed nearby the file on which they contributed the most. The large red cylinder indicates a considerable risk in a file, as its complexity is very high. Moreover, we can deduce from the visualization that an avatar next to a cylinder might have the required knowledge to maintain or review the risk.

In our example, each team member can choose among a set of given figures, which he would favor as a representation, however the idea of mapping data to glyphs displaying developers has been presented by Atzberger et al. for the task of displaying data related to the skills and expertise of developers [3]. However when using human-looking glyphs for displaying developer related data, various visual attributes become critical and should be considered very carefully. One idea to overcome this issue, would be the use of abstract forms, which only reminds on human faces, as initially presented as the popular Chernoff faces [8].

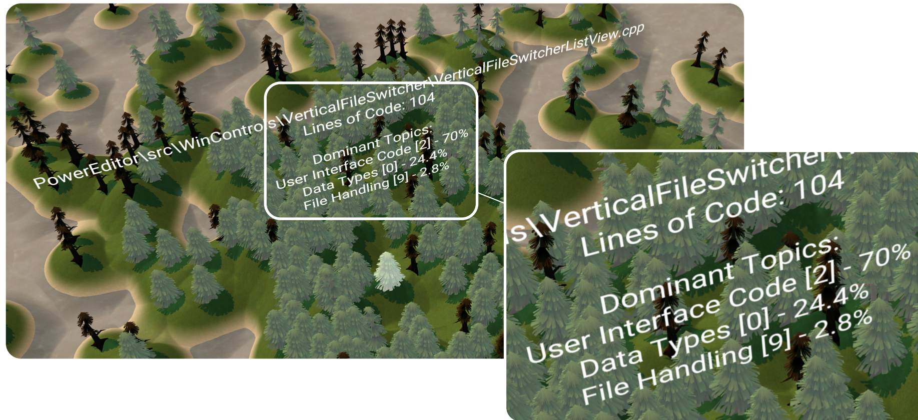

Fig. 4. Software Forest of the *notepad-plus-plus* project. The tooltip shows the underlying topic distribution of the file represented by the selected tree (highlighted). All trees with the same dominant topic are highlighted.

5 System Design and Implementation Details

In this section, we present implementation details with respect to the layout computation and the rendering of our visualization prototype, i.e., the tools and libraries we choose for generating a 2D layout based on the vocabulary in the source code files and the visualization mapping and rendering, respectively. We further describe the supported interaction techniques for enabling a user to explore a software project. We use a separation of the layout computation and the interactive rendering component, where the layout computation component computes a visualization dataset from a source code repository that is used as an input of the rendering component (see Figure 7).

5.1 Layout Computation

Our approach follows the implementation presented by Atzberger et al. in their earlier work about the Software Forest [2]. For preprocessing, the natural language in the source code documents, we apply the *nltk*[4] library for obtaining a list of stopwords for the English language. We further use *spacy*[5] for lemmatization. We used the LDA implementation provided by the library *Gensim*[6]. *Gensim* offers an LDA implementation based on the original implementation by Blei et al. [6] as well as its online version introduced by Hofman et al. [22]. The implementation of MDS is taken from the Machine learning library *scikit-learn*[7]. The result of the layout computation is a 2D layout that is merged with other static source code metrics into a CSV file that is later used for input to the rendering component.

[4] https://www.nltk.org/
[5] https://spacy.io/
[6] https://radimrehurek.com/gensim/
[7] https://scikit-learn.org/stable/

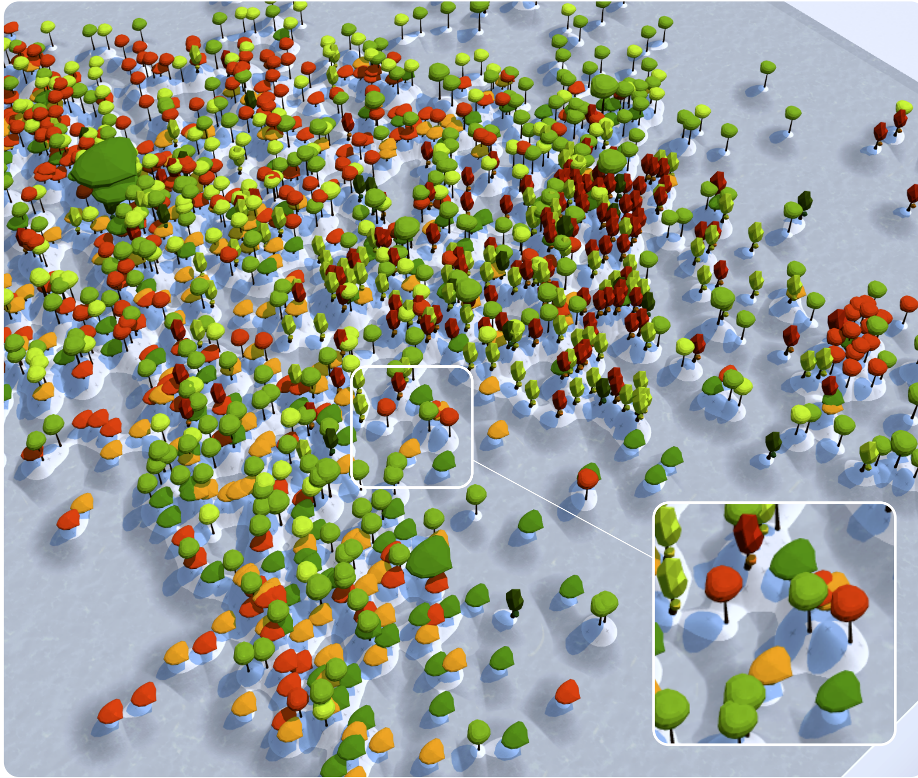

Fig. 5. Part of the Software Forest for the bitcoin project.

5.2 Rendering

The rendering component is an extension to a scatter plot renderer, written written in *TypeScript* and *WebGL* [49]. The main dependency is the open-source framework *webgl-operate*[8], which handles canvas and WebGL context management, as well as labeling primitives and glTF scene loading and rendering. The 3D glyph models are integrated into the rendering component by means of a glyph atlas and a configuration file. The basic designs for our more advanced visualization metaphors, i.e., trees and people, are taken from *SketchFab*[9]. The 3D glyph atlases are constructed manually using *Blender*[10], but every other 3D editor with glTF is a feasible alternative. Together with a glyph atlas, we have to specify its objects within a *JSON* configuration file (an example is given in Listing 1). This rendering component is embedded into a Web page with further GUI elements for the visual mapping and direct interaction techniques on the canvas to support navigation in the semantic software map.

[8] https://webgl-operate.org/
[9] https://sketchfab.com/feed
[10] https://www.blender.org/

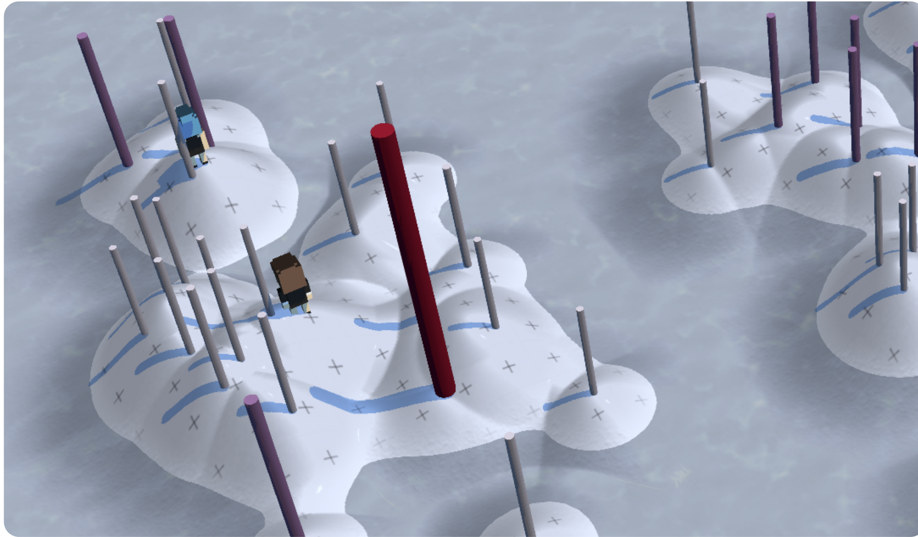

Fig. 6. Examples using avatars positioned in relation to their coding activities.

5.3 User Interaction

As basic interaction techniques, a user can choose mappings from data to visual variables of the selected glyphs, navigate through the 2.5D visualization, and retrieve details on demand displayed by tooltips by rotating and zooming. Figure 8 shows the user interface of our web-based implementation prototype. Our system supports basic interaction techniques, e.g., rotating and zoom. Furthermore, a tooltip displaying the entire entries of the respective data point contained in the CSV file shows up. As our approach highly depends on LDA, we highlight all trees with the same dominant topic as the selected one when hovering over it. Furthermore, our system allows the user to define a custom mapping between data columns and the visual variables provided by the selected model. For each model, we ensure that at least the type, the height, and the color are available as visual metaphors. By adjusting the effect of the variable tree size, which depends on a data attribute, the user can further interactively explore the effect of data variables for the source code files. In order to enhance the rendering with visual cues and more fidelity, the user can modify rendering details, e.g., by toggling Anti-Aliasing or soft shadows.

6 Conclusion

6.1 Discussion

Software Visualization techniques support users in program comprehension tasks by displaying images based on data related to software artifacts. In many cases, the questions are about the semantics of a software project, e.g., for locating

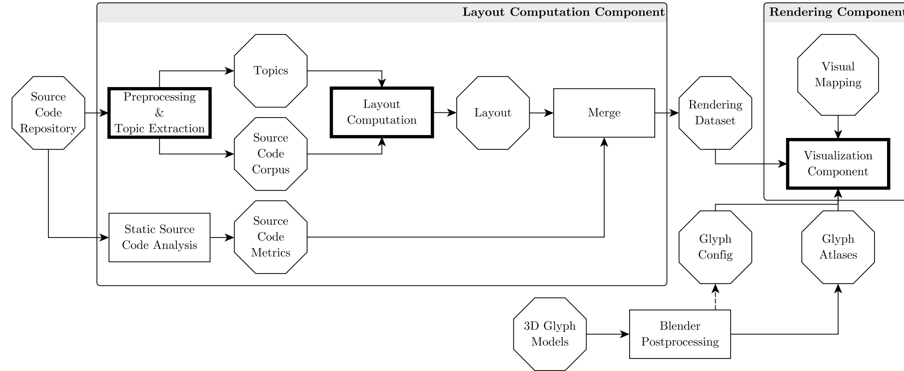

Fig. 7. The data processing pipeline to compute the semantic layouts. The rendering component composes the layouts and glyph atlases based on the visual mapping.

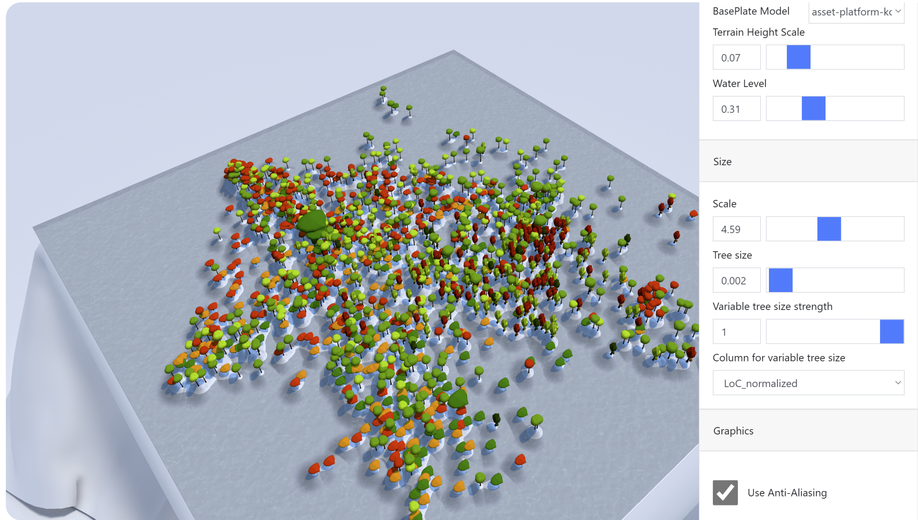

Fig. 8. User interface of our web-based implementation. Besides the full mapping configuration, rendering parameters can be adjusted by the user.

concepts or functionalities in the source code. Our previous work used a tree metaphor and a semantic layout to create map-like visualizations to support users in program comprehension tasks related to the semantics. In this extended work, we detailed how the topic model LDA and the dimension reduction technique MDS are applied for generating a layout for capturing the semantic relatedness between source code files. We presented mappings between quantitative and qualitative aspects of source code files, e.g., source code metrics or file types, and visual variables of selected 3D glyphs for concrete program comprehension tasks. We applied the city, the forest, and the island metaphor for our use-cases. Our

```json
{
  "modelFile": "PeopleCylinders2.glb",
  "attributes": [ "color" ],
  "modelScale": 1.0,
  "types": [
    { "name": "Cylinder",
      "baseModel": "Cylinder_Ax.001",
      "variants": [
        { "name": "Cylinder_Ax.001", "color": 1.0 },
        { "name": "Cylinder_Ax.002", "color": 0.75 },
        { "name": "Cylinder_Ax.003", "color": 0.5 },
        { "name": "Cylinder_Ax.004", "color": 0.25 },
        { "name": "Cylinder_Ax.005", "color": 0.0 }
      ]
    },
    { "name": "People",
      "baseModel": "Person0",
      "variants": [
        { "name": "Person0", "color": 1.0 },
        { "name": "Person1", "color": 0.5 },
        { "name": "Person2", "color": 0.0 }
      ]
    }
  ]
}
```

Listing 1: An example JSON configuration of a mapping from attribute values to 3D model. Two types of glyph categories are defined, namely “Cylinder” and “People”. This configuration can be used to display source code files as cylinders and developers as people-looking glyphs in the same semantic software map.

web-based visualization can visualize large data sets and provides a significant degree of freedom to the user by supporting various interaction techniques.

Though LDA has shown great success in modeling the concepts inherent in a software project [38], the possibility of visual indication of misleading or irrelevant relations must not be neglected. For example, the positioning of a document in 2D is not unique as it arises from a convex linear combination of the reduced topic-word distributions. Therefore, two documents with totally different document-topic distributions may be placed next to one another. In practice, however, the choice of the Dirichlet prior α forces LDA to favor document-topic distributions with only a few topics.

For our experiments, we created visualizations for the two Open Source projects *globjects* and *bitcoin*. Both can be seen as representatives for mid-sized software projects. However, a software visualization should also provide insights into large projects as the need for program comprehension increases with project size. Figure 9 shows a 2.5D visualization for the Machine Learning framework *TensorFlow*[11] that comprises a total of 13 154 files, where each file is represented by a cylinder with the same visual mapping as presented in Section 4. The data volume makes it difficult to maintain interactive framerates on average machines. Though the island (without glyphs) is a map-like visualization in itself, its capabilities are limited, as its number of visual variables is limited.

[11] Source code taken from https://github.com/tensorflow/tensorflow

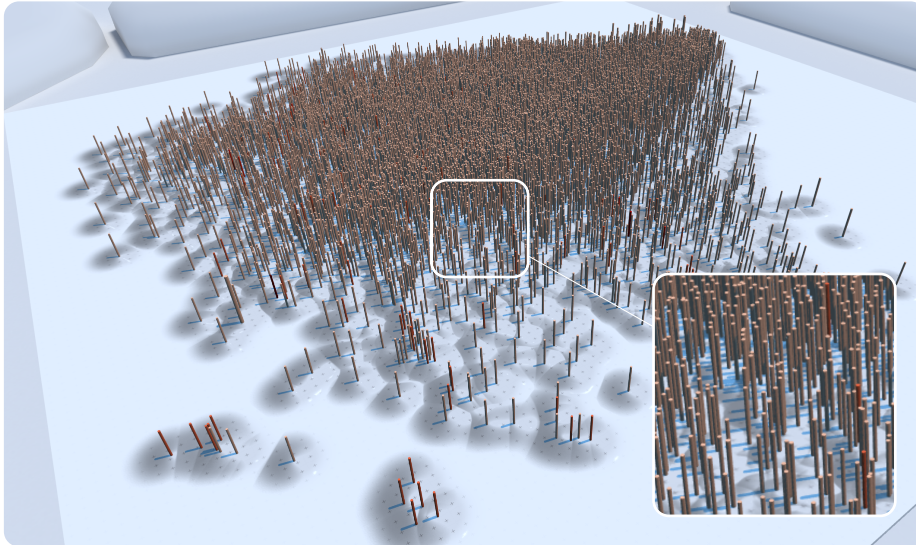

Fig. 9. Visualization of the tensorflow dataset using cylinders. The dataset contains 13 154 files, which shows the limitation in discernible data items of our technique.

Our visual mappings were motivated by common questions in a software development process. However, we do not provide empirical measurements, e.g., provided by a user study, that would demonstrate the actual benefit of our visualizations for users. It is yet unclear whether the choice of visual variables and glyphs is appropriate for users in real-world settings. The ideas presented in Section 4 so far only provide a starting point for future investigations. Nevertheless, the given examples of map configurations indicate that our 2.5D software visualization is suitable for depicting aspects of and relations within software data, supports finding information and gaining knowledge, and possibly synchronizes the mental representation of selected software aspects with the actual data.

6.2 Future Work

Concerning our visualization approach, various possibilities for future work exist. Most importantly, the effectiveness of our approach and the visual mappings should be evaluated in a systematic user study, e.g., to identify visual mappings best-suited for program comprehension tasks in an industrial setting with developers and project managers. Furthermore, we can imagine including more advanced visual mappings, especially in the case of the city metaphor [32].

Our glyph placement strategy is an example of a data-driven approach [50] for that distortion techniques should be considered. Our examples indicate that an increased glyph height tends to increase visual clutter. Therefore, distortion strategies as presented in [50] seem well-suited for mitigation. A modern approach for removing distortion in 2.5D visualizations was presented in [48] and

should be applicable for our case. Furthermore, quality metrics associated with the results of dimension reduction techniques can help measure whether the dimension reduction was able to capture local and global structures within a dataset. As our visualization approach mainly builds upon the semantic layout, we plan to implement a feature in our framework that generates the layout for a given software project automatically by evaluating various dimension reduction techniques with respect to selected quality metrics, as presented in [18].

Acknowledgements

This work is part of the “Software-DNA” project, which is funded by the European Regional Development Fund (ERDF or EFRE in German) and the State of Brandenburg (ILB). This work is part of the KMU project “KnowhowAnalyzer” (Förderkennzeichen 01IS20088B), which is funded by the German Ministry for Education and Research (Bundesministerium für Bildung und Forschung). We further thank the students Maximilian Söchting and Merlin de la Haye for their work during their master’s project at the Hasso Plattner Institute during the summer term 2020.

References

1. Aggarwal, C.C., Zhai, C.: Mining text data. Springer Science & Business Media (2012). https://doi.org/10.1007/978-1-4614-3223-4
2. Atzberger, D., Cech, T., de la Haye, M., Söchting, M., Scheibel, W., Limberger, D., Döllner, J.: Software Forest: A visualization of semantic similarities in source code using a tree metaphor. In: Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications – Volume 3 IVAPP. pp. 112–122. IVAPP ’21, INSTICC, SciTePress (2021). https://doi.org/10.5220/0010267601120122
3. Atzberger, D., Cech, T., Jobst, A., Scheibel, W., Limberger, D., Trapp, M., Döllner, J.: Visualization of knowledge distribution across development teams using 2.5d semantic software maps. In: Proc. 13th International Conference on Information Visualization Theory and Applications. IVAPP ’22, INSTICC, SciTePress (2022)
4. Atzberger, D., Scheibel, W., Limberger, D., Döllner, J.: Software Galaxies: Displaying coding activities using a galaxy metaphor. In: Proc. 14th International Symposium on Visual Information Communication and Interaction. pp. 18:1–2. VINCI ’21, ACM (2021). https://doi.org/10.1145/3481549.3481573
5. Beck, F.: Software feathers – figurative visualization of software metrics. In: Proc. 5th International Conference on Information Visualization Theory and Applications - Volume 1: IVAPP. pp. 5–16. IVAPP ’14, INSTICC, SciTePress (2014). https://doi.org/10.5220/0004650100050016
6. Blei, D.M., Ng, A.Y., Jordan, M.I.: Latent dirichlet allocation. Journal of Machine Learning Research **3**, 993–1022 (2003). https://doi.org/10.5555/944919.944937
7. Chen, T.H., Thomas, S.W., Hassan, A.E.: A survey on the use of topic models when mining software repositories. Empirical Software Engineering **21**(5), 1843–1919 (2016). https://doi.org/10.1007/s10664-015-9402-8

8. Chernoff, H.: The use of faces to represent points in k-dimensional space graphically. Journal of the American Statistical Association **68**(342), 361–368 (1973). https://doi.org/10.1080/01621459.1973.10482434
9. Chuah, M., Eick, S.: Glyphs for software visualization. In: Proc. 5th International Workshop on Program Comprehension. pp. 183–191. IWPC '97, IEEE (1997). https://doi.org/10.1109/WPC.1997.601291
10. Cornelissen, B., Zaidman, A., van Deursen, A.: A controlled experiment for program comprehension through trace visualization. IEEE Transactions on Software Engineering **37**(3), 341–355 (2011). https://doi.org/10.1109/TSE.2010.47
11. Cornelissen, B., Zaidman, A., Holten, D., Moonen, L., van Deursen, A., van Wijk, J.J.: Execution trace analysis through massive sequence and circular bundle views. Journal of Systems and Software **81**(12), 2252–2268 (2008). https://doi.org/10.1016/j.jss.2008.02.068
12. Cox, M.A.A., Cox, T.F.: Multidimensional scaling. In: Handbook of Data Visualization, pp. 315–347. Springer (2008). https://doi.org/10.1007/978-3-540-33037-0_14
13. Deerwester, S., Dumais, S.T., Furnas, G.W., Landauer, T.K., Harshman, R.: Indexing by latent semantic analysis. Journal of the American Society for Information Science **41**(6), 391–407 (1990). https://doi.org/10.1002/(SICI)1097-4571(199009)41:6%3C391::AID-ASI1%3E3.0.CO;2-9
14. Dehaghani, S.M.H., Hajrahimi, N.: Which factors affect software projects maintenance cost more? Acta Informatica Medica **21**(1), 63–66 (2013). https://doi.org/10.5455/aim.2012.21.63-66
15. Dübel, S., Röhlig, M., Schumann, H., Trapp, M.: 2d and 3d presentation of spatial data: A systematic review. In: Proc. VIS International Workshop on 3DVis. pp. 11–18. 3DVis '14, IEEE (2014). https://doi.org/10.1109/3DVis.2014.7160094
16. Erra, U., Scanniello, G.: Towards the visualization of software systems as 3d forests: The CodeTrees environment. In: Proc. 27th Annual ACM Symposium on Applied Computing. pp. 981–988. SAC '12, ACM (2012). https://doi.org/10.1145/2245276.2245467
17. Erra, U., Scanniello, G., Capece, N.: Visualizing the evolution of software systems using the forest metaphor. In: Proc. 16th International Conference on Information Visualisation. pp. 87–92. iV '12 (2012). https://doi.org/10.1109/IV.2012.25
18. Espadoto, M., Martins, R.M., Kerren, A., Hirata, N.S.T., Telea, A.C.: Toward a quantitative survey of dimension reduction techniques. Transactions on Visualization and Computer Graphics **27**(3), 2153–2173 (2021). https://doi.org/10.1109/TVCG.2019.2944182
19. Fernandez, I., Bergel, A., Alcocer, J.P.S., Infante, A., Gîrba, T.: Glyph-based software component identification. In: Proc. 24th International Conference on Program Comprehension. pp. 1–10. ICPC '16, IEEE (2016). https://doi.org/10.1109/ICPC.2016.7503713
20. Griffiths, T.L., Steyvers, M.: Finding scientific topics. Proc. the National Academy of Sciences **101**, 5228–5235 (2004). https://doi.org/10.1073/pnas.0307752101
21. Hawes, N., Marshall, S., Anslow, C.: CodeSurveyor: Mapping large-scale software to aid in code comprehension. In: Proc. 3rd Working Conference on Software Visualization. pp. 96–105. VISSOFT '15, IEEE (2015). https://doi.org/10.1109/VISSOFT.2015.7332419
22. Hoffman, M., Bach, F., Blei, D.: Online learning for latent dirichlet allocation. In: Advances in Neural Information Processing Systems. NIPS '10, vol. 23, pp. 856–864 (2010)
23. Systems and software engineering–Vocabulary. Standard, International Organization for Standardization (2017). https://doi.org/10.1109/IEEESTD.2017.8016712

24. Kleiberg, E., van de Wetering, H., van Wijk, J.J.: Botanical visualization of huge hierarchies. In: Proc. Symposium on Information Visualization. pp. 87–87. INFOVIS '01, IEEE (2001). https://doi.org/10.1109/INFVIS.2001.963285
25. Kleiner, B., Hartigan, J.A.: Representing points in many dimensions by trees and castles. Journal of the American Statistical Association **76**(374), 260–269 (1981). https://doi.org/10.1080/01621459.1981.10477638
26. Kohonen, T.: Exploration of very large databases by self-organizing maps. In: Proc. International Conference on Neural Networks. pp. 1–6. ICNN '97, IEEE (1997). https://doi.org/10.1109/ICNN.1997.611622
27. Kuhn, A., Loretan, P., Nierstrasz, O.: Consistent layout for thematic software maps. In: Proc. 15th Working Conference on Reverse Engineering. pp. 209–218. WCRE '08, IEEE (2008). https://doi.org/10.1109/WCRE.2008.45
28. Kuhn, A., Erni, D., Loretan, P., Nierstrasz, O.: Software cartography: Thematic software visualization with consistent layout. Journal of Software Maintenance and Evolution: Research and Practice **22**(3), 191–210 (2010)
29. Lanza, M.: The Evolution Matrix: Recovering software evolution using software visualization techniques. In: Proc. 4th International Workshop on Principles of Software Evolution. pp. 37–42. IWPSE '01, ACM (2001). https://doi.org/10.1145/602461.602467
30. Lewis, J.P., Rosenholtz, R., Fong, N., Neumann, U.: VisualIDs: Automatic distinctive icons for desktop interfaces. Transactions on Graphics **23**(3), 416–423 (2004). https://doi.org/10.1145/1015706.1015739
31. Limberger, D., Scheibel, W., Dieken, J., Döllner, J.: Visualization of data changes in 2.5d treemaps using procedural textures and animated transitions. In: Proc. 14th International Symposium on Visual Information Communication and Interaction. pp. 6:1–5. VINCI '21, ACM (2021). https://doi.org/10.1145/3481549.3481570
32. Limberger, D., Scheibel, W., Döllner, J., Trapp, M.: Advanced visual metaphors and techniques for software maps. In: Proc. 12th International Symposium on Visual Information Communication and Interaction. pp. 11:1–8. VINCI '19, ACM (2019). https://doi.org/10.1145/3356422.3356444
33. Limberger, D., Trapp, M., Döllner, J.: Depicting uncertainty in 2.5d treemaps. In: Proc. 13th International Symposium on Visual Information Communication and Interaction. pp. 28:1–2. VINCI '20, ACM (2020). https://doi.org/10.1145/3430036.3432753
34. Linstead, E., Rigor, P., Bajracharya, S., Lopes, C., Baldi, P.: Mining eclipse developer contributions via author-topic models. In: Proc. 4th International Workshop on Mining Software Repositories. pp. 30:1–4. MSR '07, IEEE (2007). https://doi.org/10.1109/MSR.2007.20
35. Linstead, E., Bajracharya, S., Ngo, T., Rigor, P., Lopes, C., Baldi, P.: Sourcerer: mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery **18**(2), 300–336 (2009). https://doi.org/10.1007/s10618-008-0118-x
36. Malony, A., Hammerslag, D., Jablonowski, D.: Traceview: a trace visualization tool. IEEE Software **8**(5), 19–28 (1991). https://doi.org/10.1109/52.84213
37. Markovtsev, V., Kant, E.: Topic modeling of public repositories at scale using names in source code. arXiv CoRR cs.PL (2017), https://arxiv.org/abs/1704.00135
38. Maskeri, G., Sarkar, S., Heafield, K.: Mining business topics in source code using latent dirichlet allocation. In: Proc. 1st India Software Engineering Conference. pp. 113–120. ISEC '08, ACM (2008). https://doi.org/10.1145/1342211.1342234
39. Rosen-Zvi, M., Griffiths, T., Steyvers, M., Smyth, P.: The author-topic model for authors and documents. In: Proc. 20th Conference on Uncer-

tainty in Artificial Intelligence. pp. 487–494. UAI '04, AUAI Press (2004). https://doi.org/10.5555/1036843.1036902
40. Scheibel, W., Limberger, D., Döllner, J.: Survey of treemap layout algorithms. In: Proc. 13th International Symposium on Visual Information Communication and Interaction. pp. 1:1–9. VINCI '20, ACM (2020). https://doi.org/10.1145/3430036.3430041
41. Scheibel, W., Trapp, M., Limberger, D., Döllner, J.: A taxonomy of treemap visualization techniques. In: Proc. 15th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications – Volume 3: IVAPP. pp. 273–280. IVAPP '20, INSTICC, SciTePress (2020). https://doi.org/10.5220/0009153902730280
42. Schreiber, A., Misiak, M.: Visualizing software architectures in virtual reality with an island metaphor. In: Proc. International Conference on Virtual, Augmented and Mixed Reality: Virtual, Augmented and Mixed Reality: Interaction, Navigation, Visualization, Embodiment, and Simulation. pp. 168–182. VAMR '18, Springer (2018). https://doi.org/10.1007/978-3-319-91581-4_13
43. Sievert, C., Shirley, K.: LDAvis: A method for visualizing and interpreting topics. In: Proc. Workshop on Interactive Language Learning, Visualization, and Interfaces. pp. 63–70. ACL (2014). https://doi.org/10.3115/v1/W14-3110
44. Skupin, A.: The world of geography: Visualizing a knowledge domain with cartographic means. Proc. National Academy of Sciences **101**(suppl 1), 5274–5278 (2004). https://doi.org/10.1073/pnas.0307654100
45. Steinbrückner, F., Lewerentz, C.: Representing development history in software cities. In: Proc. 5th International Symposium on Software Visualization. pp. 193–202. SOFTVIS '10, ACM (2010). https://doi.org/10.1145/1879211.1879239
46. Steinbrückner, F., Lewerentz, C.: Understanding software evolution with software cities. Information Visualization **12**(2), 200–216 (2013). https://doi.org/10.1177/1473871612438785
47. Štěpánek, A.: Procedurally generated landscape as a visualization of C# code. Tech. rep., Masaryk University, Faculty of Informatics (2020), bachelor's Thesis
48. Vollmer, J.O., Döllner, J.: 2.5d dust & magnet visualization for large multivariate data. In: Proc. 13th International Symposium on Visual Information Communication and Interaction. pp. 21:1–8. VINCI '20, ACM (2020). https://doi.org/10.1145/3430036.3430045
49. Wagner, L., Scheibel, W., Limberger, D., Trapp, M., Döllner, J.: A framework for interactive exploration of clusters in massive data using 3d scatter plots and webgl. In: Proc. 25th International Conference on 3D Web Technology. pp. 31:1–2. Web3D '20, ACM (2020). https://doi.org/10.1145/3424616.3424730
50. Ward, M.O.: A taxonomy of glyph placement strategies for multidimensional data visualization. Information Visualization **1**(3–4), 194–210 (2002)
51. Ward, M.O., Grinstein, G., Keim, D.: Interactive Data Visualization: Foundations, Techniques, and Applications. CRC Press (2010)
52. Wettel, R., Lanza, M.: Visualizing software systems as cities. In: Proc. International Workshop on Visualizing Software for Understanding and Analysis. pp. 92–99. VISSOFT '07, IEEE (2007). https://doi.org/10.1109/VISSOF.2007.4290706
53. Wettel, R., Lanza, M.: CodeCity: 3d visualization of large-scale software. In: Companion of the 30th International Conference on Software Engineering. pp. 921–922. ICSE Companion '08, Association for Computing Machinery (2008). https://doi.org/10.1145/1370175.1370188

54. Würfel, H., Trapp, M., Limberger, D., Döllner, J.: Natural phenomena as metaphors for visualization of trend data in interactive software maps. In: Proc. Conference on Computer Graphics and Visual Computing. pp. 69–76. CGVC '15, EG (2015). https://doi.org/10.2312/cgvc.20151246