Understanding the NPM Dependencies Ecosystem of a Project Using Virtual Reality

David Moreno-Lumbreras[†], Jesús M. González-Barahona[†], Michele Lanza[‡]

[†]*EIF @ Universidad Rey Juan Carlos – Fuenlabrada, Spain* [‡]*REVEAL @ Software Institute – USI, Lugano, Switzerland*

***Abstract*—Modern JavaScript development relies heavily on using *Node Package Manager* (*NPM*) modules. These modules are related by dependency relationships, possibly requiring dozens or hundreds of modules to build a complete JavaScript web application. Studying dependencies, in terms of their sustainability, vulnerability, size, defects, etc., is fundamental for the deployment and maintenance of JavaScript web applications.**

We use a 3D metaphor based on presenting dependencies as an "elevated city", mapping both dependency relationships and characteristics of interest of each module. We developed a VR (virtual reality) scene representing the dependencies of several web applications using the elevated city metaphor, and exposed industrial experts to it to check its suitability. They explored a medium-sized project, with more than 200 dependencies, sharing their insights.

The results highlight different aspects of our approach and how the combination of metrics helps experts to obtain insights from the ecosystem. The feedback shows the usefulness of the visualization to check and explore several aspects of the dependencies of an application, helping to identify problems related to maintainability, license usage, or vulnerabilities, and to design strategies to address them.

***Index Terms*—virtual reality, elevated city, software visualization, ecosystem, npm dependencies**

I. Introduction

Software systems are usually composed of collections of interdependent software modules, with some studies estimating that up to 80% of the code in current software applications comes from libraries and frameworks [1]. Understanding the dependencies among the modules is important for the development and maintenance of those systems. The increasing popularity of FOSS (free, open-source software) components in these systems led to the appearance of software registries. In the case of web development, the *Node Package Manager* (*NPM*) registry has become the main mechanism for sharing modules [2].

The structure of dependencies among modules (packages) in *NPM* is a network that has been shown to grow with time [3], with many of them (for example, frameworks) having several dozens of direct dependencies. This situation is attributed to the lack of standard libraries in JavaScript implementations (such as *NodeJS*), which has fostered a culture of reusing large quantities of packages for web development. Because of the sheer quantity of dependencies, and the length of the chains of dependencies, understanding the relationships between all modules needed by an application is difficult and complex.

Visualizing software dependencies can provide a powerful way to understand and manage the complexity of modern software systems [4]. While text or table formats may be useful for presenting information in a concise and structured manner, they lack the ability to convey the nuanced relationships between different components of the ecosystem. In contrast, visual representations of software dependencies allow developers to see the entire ecosystem at once in a way that is intuitive and easy to grasp. By seeing the ecosystem in this way, practitioners can quickly identify areas of the ecosystem that are particularly sensitive to change, and make informed decisions about how to modify the ecosystem without introducing unintended consequences. Visualizations of software dependencies also provide a shared language that allows developers to communicate complex ideas and collaborate effectively, fostering a culture of innovation and continuous improvement. In short, visualizing software dependencies is an essential tool for modern software development, enabling developers to build better, more reliable ecosystems that meet the needs of today's rapidly evolving technological landscape.

The tree structure [5] is a commonly used way to visualize software dependencies, where modules are arranged in a hierarchical tree according to their dependencies. This helps developers identify design issues and make informed decisions about modifying their systems. However, as systems become more complex, alternative visualizations such as Treemaps or Dependency Structure Matrices may be better suited. On the other hand, graph or network visualizations are powerful techniques for representing software dependencies [6], where nodes represent modules and edges represent the dependencies between them. These visualizations enable developers to identify high coupling and the flow of information. Graphs are especially useful for large systems to highlight potential issues. However, understanding and navigating these visualizations can be challenging, and it is important to design them carefully to fit the needs and expertise of the users.

We address the problem of understanding dependencies by using an "elevated city" metaphor in virtual reality. The elevated city is derived from the *CodeCity* metaphor [7]. It represents software modules as buildings of a city, with their features mapped to characteristics of the buildings, or to their relative location. Each building corresponds to a module, and a neighborhood of buildings corresponds to all modules that are dependencies of the same module.

Each neighborhood is elevated with respect to the building on which it depends. This elevation, therefore, represents the level of dependency, with more elevated neighborhoods representing deeper dependencies. The viewer can map different metrics of activity, size, type of license, and vulnerabilities to features of the buildings, thus obtaining different views of the modules. We have built this visualization, reusing the freely accessible *BabiaXR* framework [8], for several applications built from *NPM* modules, focusing on the identification of groups of modules with vulnerabilities, maintenance problems, license compatibility, and issues related to module size (which has an impact on load-time), and possible solutions to them.

The paper is organized as follows. We begin by presenting the results of previous field studies that motivated our approach and the development of the elevated city metaphor. Then, we describe *BabiaXR* and the elevated city for visualizing *NPM* dependencies, including the metrics, interactions, and layout. We then illustrate four use cases, followed by the description of an experiment that we ran with industry experts. We conclude with a discussion of our findings and the envisioned future work.

II. Related Work

Dependency Analysis. Dependencies ecosystems and their vulnerabilities have been studied extensively [9]–[12]. In the case of the *NPM* network, Hejderup *et al.* [13] report that one-third of the *NPM* packages use dependencies with at least one vulnerability. Abdalkareem *et al.* [9] conducted an empirical analysis in the *PyPi* network of *Python*, finding that the number of vulnerabilities increases over time. Vulnerabilities with the number of dependencies are related: Lauinger *et al.* [14] studied that relation with *JavaScript* open source projects, and Williams *et al.* [1] reported that 26% of open source *Maven* packages have known vulnerabilities. Zapata *et al.* [15] studied the impact of a vulnerability in the *ws* package on 60 *JavaScript* projects, finding that up to 73.3% of the dependent applications are safe from the vulnerability. Zerouali *et al.* [16] reported that outdated *NPM* packages increase the risk of potential vulnerabilities.

Virtual Reality. VR has been shown to facilitate discovery in domains where space plays an important role, for example in the field of brain tumors [17], perception of shapes and forms [18], paleontology [19], caves [20], and magnetic resonance imaging [21]. Data visualization in virtual reality allows the use of multidimensionality for abstract analysis, and even more so for large data sets. Regarding VR and immersive data visualizations, there is extensive prior work on the use of VR [22], [23] for visualizing scientific data coming from various domains [24]–[27].

Bryson [28] highlighted the possibilities offered by VR for interaction with complex phenomena and their data visualization. The standard way to visualize is 2D space, data visualizations in 3D (and VR) have been introduced only slowly. Munzner [29] and Few [30] warned of unjustified 3D usage. At the same time, researchers are arguing for the benefits of 3D and VR for data visualization. Some examples are: Batch *et al.* [31] focused on the spatial use, Jacob *et al.* [32] focused on the embodiment of interfaces, Rosenbaum *et al.* [33] focused on abstract involvement, and García-Hernández *et al.* [34] in the aerospace engineering field. Regarding the city metaphor visualization, the idea comes from Munro *et al.* [35] and one of its greatest exponents was *CodeCity*, proposed by Wettel *et al.* [7]. The city metaphor visualization was fully explored in VR: CityVR [36] is an example. We also conducted an experiment for comparing this visualization between the on-screen and VR version [37].

In the use of the city metaphor for visualizing software dependencies, Kobayashi *et al.* [38] presented an approach based on the city metaphor for representing software architectures and showing the inner dependencies between packages as lines between buildings. Some years after, Yano *et al.* [39] continued the work and they described an approach based on the city metaphor and some case studies to calculate indexes to represent the characteristics of software by their dependencies. The approach called *IslandViz* [40] also explored the visualization of software architectures as islands, including the package dependencies shown relationships between islands, and arrows for representing the dependencies hierarchy.

Apart from the approaches described in this section, to our knowledge, there is still no investigation of the visualization of dependencies using the city metaphor in virtual reality mixing metrics about community, vulnerability, employment, and metrics about the size or number of lines of the code.

III. The Elevated City

In this section, we describe our implementation of the elevated city within the *BabiaXR* visualization framework.

A. BabiaXR in a Nutshell

BabiaXR[1] is a toolset for extended reality (including both AR and VR) data visualization in web browsers. It facilitates the definition of data visualization scenes by automating and generalizing the most common tasks. It is based on *A-Frame*[2], a JavaScript framework to build 3D, augmented reality (AR), and VR scenes in the browser. *A-Frame* extends `HTML` with new entities to build 3D scenes as if they were `HTML` documents, using techniques common to any front-end web developer. It also leverages WebGL to run on AR and VR devices. *A-Frame* is built on top of *Three.js*,[3] which uses the *WebGL* API available in modern web browsers.

BabiaXR extends *A-Frame* by providing components for creating a data visualization 3D scene that can be presented in immersive virtual reality or in augmented reality when run on a corresponding device (headset, AR glasses). It also includes components to assist in data retrieval and processing (data filtering, mapping of fields to visualization features, etc), and to simplify the production of scenes suitable for use in experiments with human subjects. Scenes built with *BabiaXR* can be displayed on-screen, or in VR/AR devices.

[1]*BabiaXR* source code: **https://gitlab.com/babiaxr/aframe-babia-components**

[2]*A-Frame*: **https://aframe.io**

[3]*Three.js*: **https://threejs.org**

B. Visualization of Dependencies

The elevated city component of *BabiaXR* visualizes the dependencies of an application, specifically the dependencies of JavaScript applications using *NPM* packages, including all transitive dependencies.

Layout

The city is composed of buildings and districts/quarters. A building represents a package, so it corresponds to a *NPM* package which is a transitive dependency of the application. Each quarter is represented by a platform including buildings for all direct dependencies of a package. The level of the quarter corresponds to a dependency level, higher as dependencies are deeper (see Figure 1). Buildings on the first level are the direct dependencies found in the *package.json* file of the project.

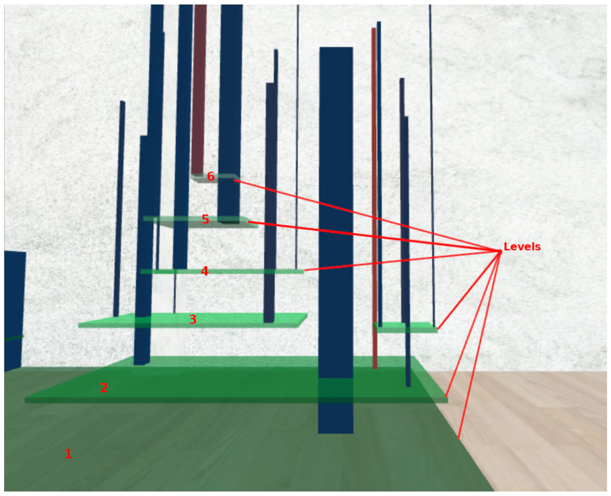
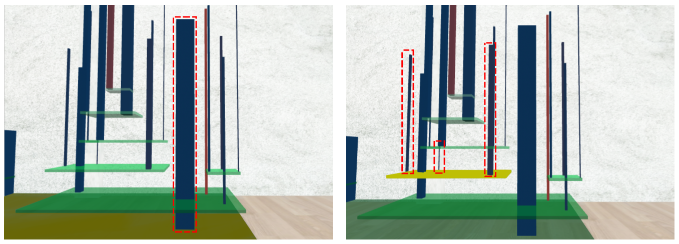


Fig. 1: Example of a *NodeJS* project with six levels of dependencies.

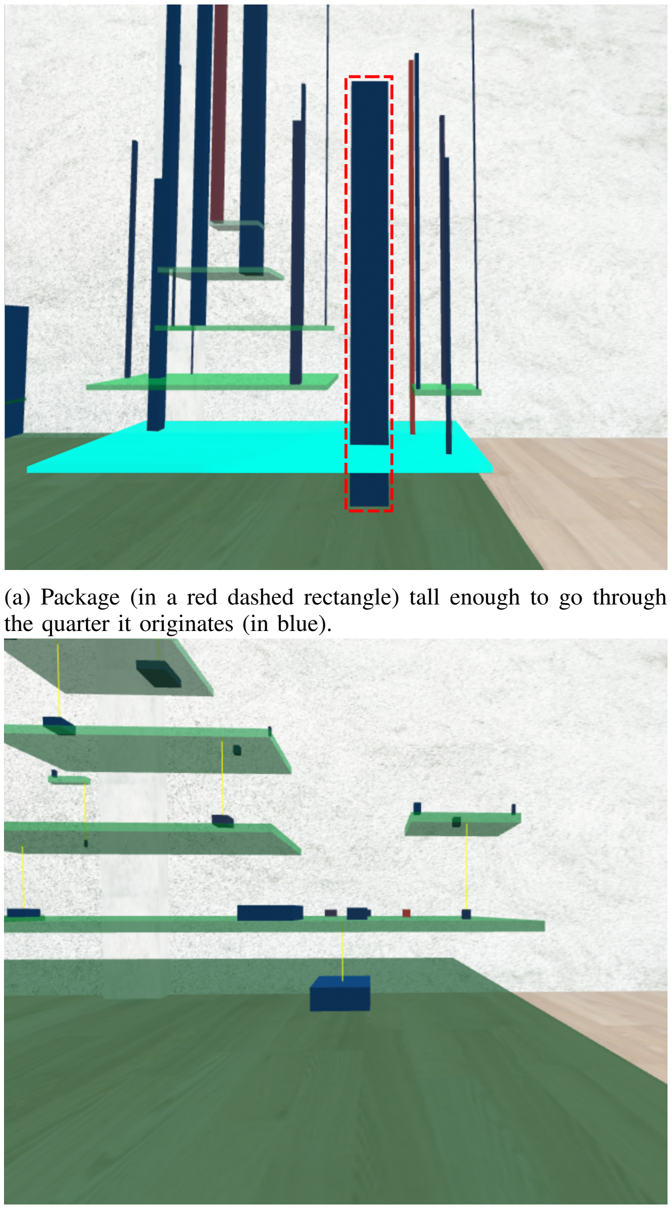
Each level has a color gradient from dark green to light green as the levels increase, to ease the identification of dependencies levels. The vertical width of the platforms representing quarters is always the same. If a building lies on a quarter, its base starts from the platform meaning that the package is a dependency of the building below the quarter (See Figure 2).



(a) First level quarter. (b) Third level quarter.

Fig. 2: Examples of buildings belonging to a quarter: buildings in a red dashed square belong to the quarter with a base in brown/yellow.

When a building, because of its outgoing dependencies "generates" a quarter, the quarter base is below it, and the building "goes through" its base. If the building is not tall enough to go through the base of its quarter, a line from the top of the building to its base is shown. Therefore, if a building does not go through a quarter or does not have a line above it, the package has no dependencies.



(a) Package (in a red dashed rectangle) tall enough to go through the quarter it originates (in blue).

(b) Short buildings with yellow lines above them, linking to the quarter they originate.

Fig. 3: Examples of buildings that are the origin of quarters.

Figure 3 shows two examples of a package going through a quarter base, and packages that are not tall enough to go through a quarter. Only a single building can be the origin of a quarter, so it is not possible to have two buildings that go through the same quarter.

Metrics

Three metrics can be mapped onto a building *height*, *area*, and *color*.

- The metric mapped to height is always *age_days*: the age of a package (in days).
- There are three metrics that can be mapped to the area:
  - **loc/age**: lines of code of the package divided by the package age (in days).
  - **size/age**: package size (in bytes) divided by package age (in days).
  - **ncommits/age**: number of commits of the package git repository divided by the age of the package (in days).

  All of these metrics are divided by age so that the volume of the building represents a meaningful metric: lines of code, size, or number of commits.
- There are several metrics that can be mapped to color. If the metric is categorical, each value will have a defined color. If the metric is numeric, the color of the building will follow a continuous palette from blue to red. The user can select any of these metrics for color in real-time:
  - **license** (categorical): license of the package.
  - **timesInstalled** (numeric): how many times a package is installed. For *NPM* dependencies, if the same package is pulled as a dependency in several versions, each of these versions will be installed. Therefore, this metric is also the number of different versions of the package in the transitive list of dependencies.
  - **timesAppear** (numeric): how many times a package appears as a dependency, regardless of the version. This is also the number of buildings that can be found in the whole city for that package.
  - **last_act_days** (numeric): the number of days since the last commit in the repository of a package.
  - **ncommits_ly** (numeric): the number of commits during the last year in a package repository.
  - **ncommiters_ly** (numeric): the number of different committers during the last year in a package repository.
  - **nvuln** (numeric): the number of vulnerabilities of a package.
  - **nissues_ratio** (numeric): the number of issues closed divided by the total number of issues in the repository of a package.

Interactions

While immersed in VR or AR, the user sees a panel with selectors for data sources, metrics for the base and color of the buildings, and for representing them solid, transparent, or as wireframe (see Figure 4). This panel can be hidden by pressing the middle button of the controller. For selecting in the panel, the user can "fire" with the raycaster of the right controller on its selectors.

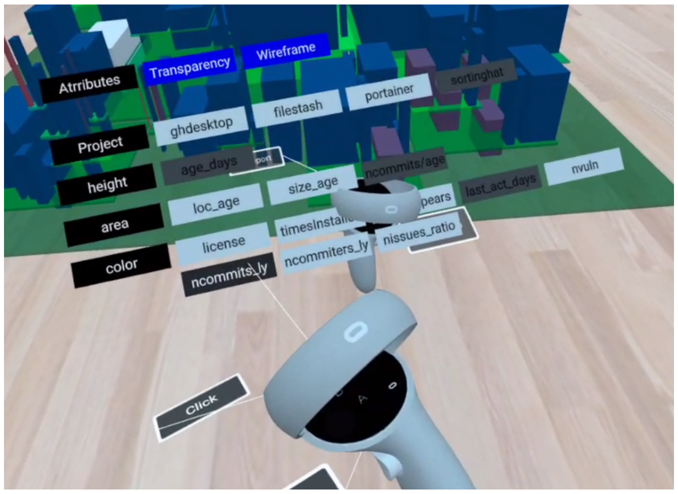
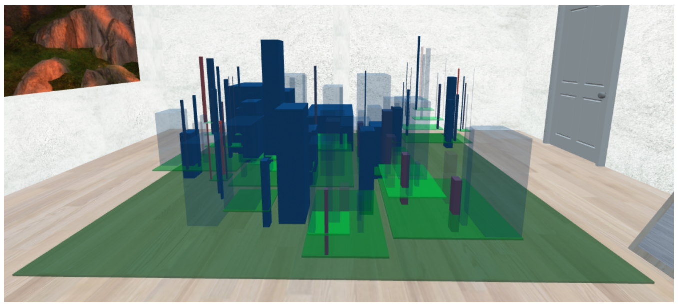
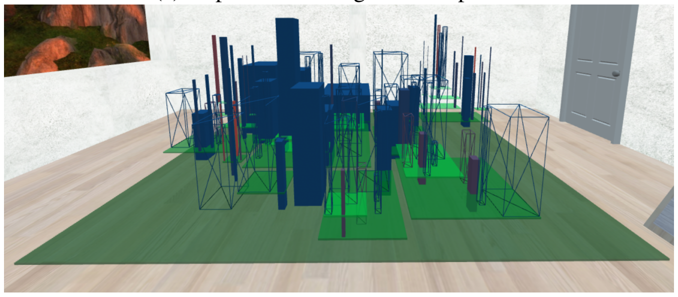


Fig. 4: User interface on the left controller.

There are cases when a package is a direct dependency of more than one other package. These "repeated" packages can be shown as transparent buildings and/or as wireframes, as shown in Figure 5. This option can be activated/deactivated using the first row called *Attributes* of the user interface.



(a) Repeated buildings as transparent.



(b) Repeated buildings represented as wireframes.

Fig. 5: Examples of the transparency and wireframe for highlighting repeated packages of one of them.

By default, these options make transparent or wireframe all repeated buildings. For finding the repeated packages of a specific building, the user can point, or "fire", on a building: its repeated buildings will be highlighted in white. Figure 6 shows this behavior. If a quarter is "fired at", a transparent gray box appears to highlight it, with a panel in black showing the path of its dependency level.

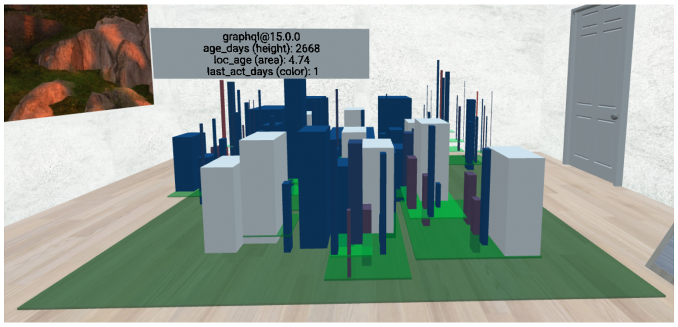
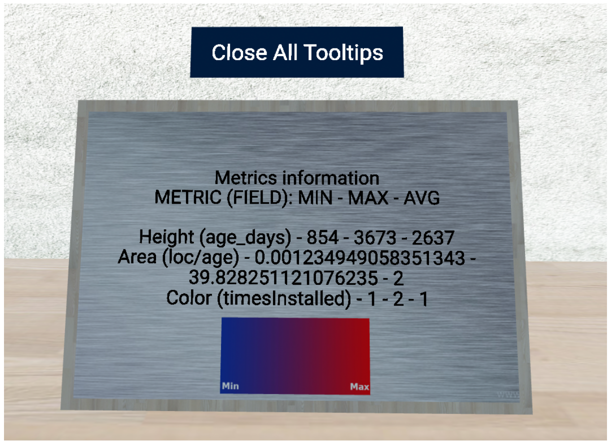

graphql@15.0.0
age_days (height): 2668
loc_age (area): 4.74
last_act_days (color): 1

Fig. 6: One package and its replicas in white color when pointed/fired.

Close to the city, there is a panel with a summary of the metrics, and if the color metric is categoric, close to it, there will be another panel with the defined color and its values. On top of it, as shown in Figure 7, there is a button for closing all the legends opened in the city, with the goal of cleaning it if many are opened.



Metrics information
METRIC (FIELD): MIN - MAX - AVG

Height (age_days) - 854 - 3673 - 2637
Area (loc/age) - 0.001234949058351343 - 39.828251121076235 - 2
Color (timesInstalled) - 1 - 2 - 1

Fig. 7: Metrics information panel.

C. Views

Depending on which metrics are mapped to the colors of buildings in the elevated city, we can identify four "views":

License. With *license* selected as a metric for colors, each building has a color determined by its license. On the right side of the scene, there is a color legend with the different licenses found in the packages. In this view, the user can easily check for, for example, the most used licenses in the project, or if a certain branch of dependencies has non-compatible licenses.

Replication. This view includes mapping either *timesInstalled* or *timesAppear* metrics to colors. Having the same package in several versions among the installed dependencies is usually a bad smell that may cause problems. Mapping *timesInstalled* to colors makes it easy to spot those packages, and in which parts of the dependency tree they are (when combined with the wireframe and transparency features). Mapping *timesAppear* to colors helps to quickly spot packages that are pulled in the installation because they are dependencies of many other dependencies.

Activity. This view includes mapping *last_act_days*, *ncommits_ly* or *ncommiters_ly* to colors. It aims to help in the analysis of how active is the development of the packages so that it is easy to locate which packages, or which parts of the dependency tree, may be no longer maintained, or have a very low maintenance activity. A healthy absolute activity (be it the number of days since the last activity, or commits or active committers during the last year) may be different for packages of different sizes. But viewing these metrics in combination with the size of the package will help to spot likely abandoned packages, or those likely to be under-maintained.

Vulnerabilities. This view includes mapping *nvuln* and *nissues_ratio* to colors. *nvuln* shows the number of vulnerabilities in the package (extracted with *npm audit*). It, therefore, helps to find which packages, or which branches in the dependency tree may be vulnerable to security issues. *nissues_ratio* shows the ratio of closed issues to the total number of issues in the package repository. Problems, improvements, or questions are usually tracked via issues [41]–[43], which are closed when solved. So, the fraction of closed issues is an indicator of how problems are being dealt with in a package, helping to find likely problematic dependencies.

The selected metrics in the approach are essential for managing projects and ensuring that they are secure, reliable, and compliant with legal requirements. The license view can help to identify the different licenses used in the project, which is crucial in ensuring compliance and avoiding legal issues. The replication view can help to identify potential problems caused by multiple versions of the same package and dependencies, while the activity view can help to identify which packages or branches in the dependency tree may be under-maintained or abandoned. Finally, the vulnerabilities view can help to identify packages or branches that may be vulnerable to security issues by tracking the number of vulnerabilities and the ratio of closed issues to total issues in the package repository. To summarize, the selected metrics could provide valuable information to developers, project managers, and legal teams, helping them to manage open-source projects effectively and avoid potential problems.

IV. Use cases

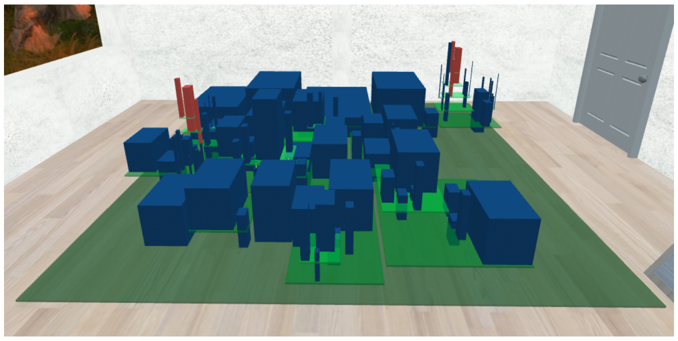
To check the usability of the elevated city metaphor, we produced four scenarios (same setup, with data from four different applications), to better understand and refine the metaphor ourselves. We then run a qualitative experiment with practitioners, using two of those scenarios.

A. Scenarios

We selected four FOSS projects, with a very different number of dependencies. Table I shows the main characteristics of those projects. Figure 8 shows the complete scenes of the elevated cities for the project with the least dependencies (SortingHat UI) and with the most dependencies (GitHub Desktop) among them.

TABLE I: Projects used for the scenes, including a brief description of them, and the total number of their dependencies.

Project	Description	Dependencies
GitHub Desktop	Open source Electron-based GitHub app. It is written in TypeScript and uses React.	2391
Portainer	Lightweight service delivery platform for containerized applications that can be used to manage Docker, Swarm, Kubernetes, and ACI environments.	780
PM2	Production process manager for Node.js applications with a built-in load balancer.	244
SortingHat UI	User interface for the SortingHat application	97


(a) SortingHat UI

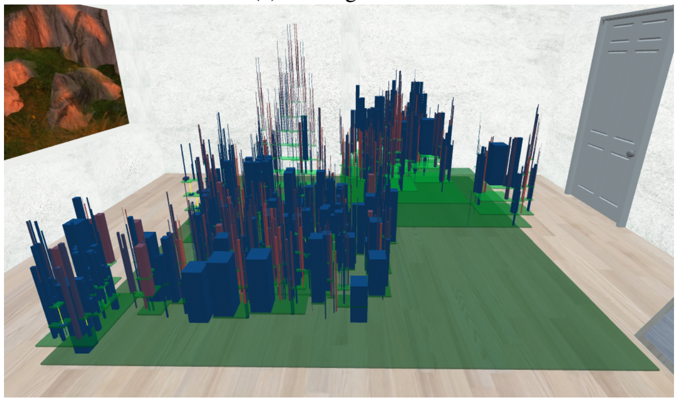

(b) GitHub Desktop

Fig. 8: Pictures of two scenes with the elevated cities for SortingHat UI (left) and GitHub Desktop (right).

More details about the data retrieval for producing these scenes, guidelines for how to replicate them, and the code to visualize the scenes themselves (ready to load in a web browser if served by an HTTP server) are in the replication package, see Section VIII.

V. Experts feedback

We tested the elevated cities visualization with four industrial practitioners, with the aim of receiving qualitative feedback, to learn to which extent it is useful in real-world situations related to software development. For that, we used two of the scenes we developed. The process followed with each subject is as follows.

Training. We started by asking subjects about their experience in VR, along with some demographic questions. Then, they run the "First steps" Meta Quest 2 tutorial, to become used to the controllers and experience immersion. Next, subjects were guided through the experiment scene, to become familiar with movement and interaction. In the scene, we presented the elevated city of the program with the lowest number of dependencies: SortingHat UI, whose data is shown in Table I. Subjects were guided through each of the visualization features, the user interface for changing metrics, and each of the available views described in Section III.

Experiment

The experiment starts when the project in the scene is changed to the medium-sized one, PM2. Subjects are now left to interact freely with the visualization using all four views while providing their feedback in an open conversation. Meanwhile, their voice and their view in VR are recorded, using the facilities of the headset, so that we could later play both, to obtain more detailed information.

After this, some specific questions about the tool were asked to each subject:

- Which view (or combination of views) do you find most useful?
- Which metrics do you find most useful?
- Which parts/features of the metaphor did you find more useful?
- Do you use any other tool for analyzing dependencies?
- Would you change/improve something about the tool?
- Do you have any other comments?

A. Results

We interviewed four subjects from two different companies with strong relationships with software development. Two of the subjects had more than 10 years of experience in programming, the other two had 7 to 10 years of experience. All of them had previous experience with VR and only two of them have previous experience with the *NPM* ecosystem.

License View

All participants unanimously agreed that it is easy to identify the predominant license in the software ecosystem. One participant noted that this was particularly helpful for open-source projects, as it allowed them to quickly identify any sub-dependencies that had licenses that did not fit with their project policy. This feedback highlights the importance of understanding and managing software licenses in open-source projects, which can be a complex and time-consuming task. The license view of the visualization approach provided a quick and easy way for developers to gain insight into the licenses of the software dependencies they were using. In addition, the license view could be used to identify potential legal issues that could arise from using certain software dependencies. For example, if a sub-dependency had a license that was incompatible with the main project license, it could lead to legal conflicts down the line. By using the license view to identify and address these issues early on, developers can ensure that their software projects remain legally compliant and avoid potential legal liabilities.

Overall, the feedback on the license view suggests that it is a valuable view for managing software licenses and ensuring compliance with project policies. By providing a clear and intuitive visualization of software licenses, the approach can help developers make informed decisions about which dependencies to use and ensure that their software projects remain legally compliant.

Replication View

All participants used the replication view to check how often a package was installed more than once and to see if the size of the building had a direct relationship with the replication of packages. One participant highlighted the importance of correlating the installed times of an application with the number of commits as a combination of community and popularity. They suggested that the replication view could be used to identify popular packages that are actively maintained and have a strong community, which could indicate their reliability and potential for future development. The participants agreed that using the metric that shows how many times a package appears in the application gives useful information about the most used dependencies, identifying core dependencies. This information can be used to make informed decisions about which dependencies to prioritize for maintenance and to identify any potential issues with the core dependencies of a software project. In addition, participants also noted that the replication view could be useful for identifying potential performance issues that could arise from the replication of certain packages. For example, if a package is replicated multiple times, it could lead to increased memory usage and slower performance. By using the replication view to identify and address these issues early on, developers can ensure that their software projects remain performant and efficient.

Overall, the feedback on the replication view suggests that it is a valuable view for identifying popular and core dependencies, as well as potential performance issues. By providing a clear and intuitive visualization of package replication, the approach can help developers make informed decisions about which dependencies to prioritize for maintenance and ensure that their software projects remain performant and efficient.

Activity View

The activity view was used for correlating information about big packages, in terms of size, with the community behind them and to identify packages that appeared to be abandoned. One of the subjects, while looking for abandoned packages, noticed that those which are not in the first level of the dependency tree posed a different kind of problem because the application developer has little control over them. This highlights the importance of understanding the complete dependency tree and identifying potential issues in sub-dependencies that could impact the overall stability and performance of the software project. Another participant noticed that by correlating the number of commits with the last activity metrics, some packages that seemed abandoned could in fact be mature software not needing further development. This suggests that the activity view can help developers make informed decisions about which packages to prioritize for maintenance, as well as to identify any potential issues with mature packages that may not require further development. Participants also noted that the activity view could be used to identify packages that are actively maintained and have a strong community behind them. By identifying these packages, developers can make informed decisions about which dependencies to use and can ensure that their software projects remain reliable and up-to-date.

Overall, the feedback on the activity view suggests that it is a valuable tool for identifying potential issues with sub-dependencies and for making informed decisions about which packages to prioritize for maintenance. By providing a clear and intuitive visualization of package activity, the approach can help developers ensure that their software projects remain stable, reliable, and up-to-date.

Vulnerabilities View

Participants also discussed the importance of this view in the context of security. One participant noted that the visualization helped them quickly identify packages with known vulnerabilities, allowing them to proactively take measures to mitigate potential security risks. They also found it helpful for tracking the status of vulnerability fixes and monitoring any new vulnerabilities that may arise in the future. The vulnerabilities view was also used in combination with other views. For instance, participants combined this view with the replication view to identify if packages with vulnerabilities were installed more than once, indicating a higher potential for risk. They also used it in conjunction with the activity view to check if packages with vulnerabilities had low maintenance activity, indicating a potential for security issues in the future. Moreover, the participants found the ratio of issues metric particularly useful in this view. They used it to gain insights into how responsive a package's development community was in addressing security issues. This information, combined with the vulnerabilities and activity metrics, provided a more comprehensive understanding of a package's overall security and maintenance status.

Overall, the vulnerabilities view proved to be a critical tool for ensuring the security and stability of software ecosystems. By providing an easy-to-use and interactive interface, it allowed developers to quickly identify and address potential security vulnerabilities and monitor the status of fixes. Combining this view with other visualizations offered additional insights into the ecosystem's health and provided a more complete picture of potential risks and issues.

Combining the Views

Participants also explored ways to combine different views in order to retrieve more information about the software ecosystem. One popular combination was the replication view and the vulnerabilities view. By cross-referencing these two views, participants were able to check whether a package that was installed more than once had any known vulnerabilities. The replication and vulnerabilities views were also combined with the activity view to check for maintenance problems in packages that appeared frequently and had vulnerabilities. Participants used metrics such as the number of commits, the number of committers, and the issues ratio to determine if these packages were being actively maintained or if they had been abandoned. All of the participants used this combination of views, indicating that it was a valuable way to gain a deeper understanding of the software ecosystem. The ability to combine different views allowed participants to identify potential issues in the software system that might have been missed by using only one view.

These results highlight the importance of providing a flexible and customizable visualization approach that allows developers to combine different views to suit their needs. By providing multiple views and metrics, developers can gain a more comprehensive understanding of their software dependencies and identify potential issues before they become major problems.

During the evaluation of the software dependency visualization approach, participants made several comments on aspects of the approach beyond the color views. One participant expressed surprise at the high number of tall buildings in the visualization, which indicated the presence of old dependencies that might be of interest for further exploration. This observation suggests that the approach can be useful in identifying areas of a software system that may need refactoring or updating. Another aspect of the approach that received positive feedback was the transparency and wireframe features for highlighting unique packages. All participants found these features helpful, although they had different preferences for how they used them. Two participants consistently used the transparency feature in all their responses, while the other half preferred the wireframe feature. Interestingly, both participants who used the wireframe feature noted that they preferred to keep unique packages solid, as this information was more important to them than whether a package was repeated. These comments demonstrate that different developers may have different preferences for visualizing software dependencies and that providing a range of visualization options can be beneficial. The transparency and wireframe features allowed participants to quickly and easily identify unique packages in the system, which helped them to understand the overall structure of the software system. The comments also suggest that developers may be interested in exploring the history and evolution of software dependencies over time, which could be an interesting avenue for future research.

Overall, the feedback from the evaluation of the dependencies ecosystem visualization approach indicates that it is a promising tool for analyzing and managing software dependencies and that there is potential for further development and refinement of the approach to meet the specific needs and preferences of different developers.

VI. Threats to Validity

Internal threats to validity refer to the potential issues that could occur within the study itself, such as flaws in the study design or errors in the data collection process. Some possible internal threats to validity for the initial study are:

- Selection bias: The participants could not be randomly selected, and the sample may not be representative of the larger population, which could affect the generalizability of the results. To mitigate this threat, we interviewed four practitioners from two different companies, holding different positions.
- Experimenter bias: The experimenter may unconsciously influence the participants or the study outcomes, which could lead to biased results. To mitigate this threat, the experimenter was not involved in conducting the study, only present to troubleshoot technical issues with the device. In addition, in the training, the experimenter followed the same steps with the four participants, avoiding bias in previous learning.
- Training bias: Some participants could be more experienced with the projects or have familiarity with *BabiaXR*, then they may perform better on the exploration, and this could potentially confound the results. In this study, none of the participants had previous experience with *BabiaXR* and this visualization, so the training was the same for all of them, carried out in the same way. In addition, in the case of having experience with the project represented, none of the participants had knowledge about the ecosystem of project dependencies, so this threat is mitigated.

External threats to validity refer to issues that could affect the generalizability of the study results to other populations or settings. Some possible external threats to validity for the initial study are:

- Generalizability: The results of the study may not be applicable to other populations or settings, as the sample

may not be representative of the larger population or the study setting may not be typical. To mitigate this threat, we have developed the tool in the web environment, being a universal environment and replicable with any device that understands the *WebXR* standards. In addition, the only requirement to participate is basic notions of the dependency ecosystem, so the range of the population that can participate is wide.
- History: The results of the study may be influenced by external events that occur during the study period, such as changes in the software development process or the availability of new tools or resources. We mitigated this threat by conducting the study in a period in which *BabiaXR* did not undergo any changes, and the data of the selected projects did not change during the study.
- Maturation: The participants may change over time during the study period, which could affect their responses or behavior and thus affect the study outcomes. The study lasted the same length of time for each participant, their answers being free and without having to define anything other than the information that was found in their exploration, if the participant reflected changes in their answers, these are also reflected in the results, mitigating this threat.

VII. Discussion

NPM offers default statistics displayed in plain text or tables on project dependencies, but our approach uses visual representation through artifacts and geometric aspects. While tables and textual analysis may provide statistical insights about dependencies, they are limited in their ability to convey the multidimensionality and complexity of the data. Visual representations, on the other hand, offer a more intuitive way to understand the relationships and dependencies among variables. By enabling users to manipulate and interact with the data in a virtual environment, they can gain a deeper understanding of the patterns and trends within the data. Additionally, visual representations can help to identify outliers, anomalies, and correlations that may not be immediately apparent through textual/tabular analysis. Therefore, we believe that while tables and textual analysis may be useful for presenting basic statistics, they cannot replace the benefits of visual representations for dependency analysis.

Our approach following the metaphor of the elevated city has proven to be useful for industry experts: they managed to extract useful information from it. Focusing on the color of the buildings, and using different views, users access varied and interrelated information about different types of metrics.

In the final feedback survey, we asked participants for the views they found most useful. Three of them agreed that the activity view is important to know if a package is well maintained or if there is a strong community behind it. Combining this with vulnerabilities and package size, problems related to the overall size and vulnerabilities of the application can be dealt with. One participant commented that the license view is very interesting for open-source projects since it gives an overview not only of the different licenses but it points out where licensing problems may be, and how spread they are among dependencies.

Regarding metrics, participants emphasized the vulnerability metrics: the closed issues ratio and the number of vulnerabilities. This appears to be due to the fact that when developing a project in JavaScript, vulnerabilities are reported by *NPM*, but just showing in which packages they were found. Among activity metrics, days since the last commit and the number of commits during the last year were highlighted by participants. The number of times a package is installed was also highlighted by one participant, since installing a package more than once, apart from taking up more physical disk space, can cause other problems.

When asked about key elements of the elevated city in general, participants agreed that changing the view and the color metric in a simple and lively manner was fundamental to detect useful information and correlate it quickly. This was from the beginning an objective of our approach: to be able to quickly visualize information derived from the activity, licenses, vulnerabilities, and replicas in a dependency tree or a part of it. The levels of the neighborhoods to differentiate the different levels of dependencies were also identified as very useful parts by the participants. This is the main difference with the traditional code city. We modified the layout of the city so that these neighborhoods are better identified with the height and with a simple glance you can see the deepest level of dependencies of an application. In addition, when immersed in VR, this observation is even easier, since the participant can observe at eye level the maximum level of dependencies and can use movements such as bending down to find information between levels. This behavior was used by several participants during the experiment.

Only one participant had used a comparable tool to visualize the dependencies of an application. However, that tool only provides the dependencies size using a treemap layout, and only for a specific JavaScript technology (bundles generated with *webpack*), not on any *NPM* dependency.

We also asked subjects about possible improvements, general comments, and other information that the participants might have missed in the elevated city. This question raised interesting issues, such as a feature to detect or isolate direct dependencies, cascade evolution when updating a dependency of the first levels, improving the color of the quarters, providing a new view for those packages that originate quarters, and those which have no dependencies. We expect to have these comments into account in future work.

Summarizing all the information retrieved in this initial study, provided important insights into the tool's potential for managing dependencies ecosystems of projects. We aim for an empirical study for further validation, but the initial feedback from participants gave us an indication of the tool's usefulness and effectiveness. The study highlighted the importance of selecting appropriate views and metrics, such as licenses, replication, activity, and vulnerabilities, for managing the dependencies of the projects. Additionally, it shed light on usability, user-friendliness, and visual design aspects of the tool, which can be used to enhance it further. The initial study played a pivotal role in the early stages of understanding the potential of the tool, making it an essential stepping stone toward a more comprehensive empirical study.

VIII. Conclusions

In this paper, we presented the elevated city, an XR-based approach, based on the city metaphor, for visualizing different aspects of the dependencies of an application. We implemented it for the dependencies of JavaScript applications, relying on web browser technologies, making the approach universal and accessible. The city represents packages as buildings and dependency levels as floating quarters. The visualization includes four views to analyze four different types of information: the type of licenses of the packages, the replication of dependency packages, the activity of the package repositories, and the vulnerabilities found in packages. These views are used to obtain insights about the ecosystem and to analyze and detect issues.

We presented our implementation of the elevated city to some practitioners. Results showed the usefulness of the approach in terms of analysis of licenses, community, vulnerabilities, and use of the packages. The subsequent feedback helped to identify the key points of our approach, as well as other comments that we will take into account in the future. While the initial feedback from participants on the tool is positive, it is essential to conduct a more comprehensive study in the future to validate its usefulness and effectiveness. A well-designed empirical study with larger sample size and a proper research method would provide more significant insights into the strengths and limitations of the tool. It would also help to identify any areas for improvement and to determine if the selected metrics are adequate for managing the dependencies ecosystem. Furthermore, an empirical study could provide more reliable evidence of the usefulness of the tool and its potential impact on managing projects.

Replication package. The data and the analysis obtained for our experiment, including the material needed for replication, are publicly available[4].

Acknowledgements. We acknowledge the financial support of the Swiss National Science foundation (SNSF) for the project "INSTINCT" (SNF Project No. 190113).

References

[1] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," *Asp. Secur. Inc*, pp. 1–26, 2012.

[2] K. Chatzidimitriou, M. Papamichail, T. Diamantopoulos, M. Tsapanos, and A. Symeonidis, "npm-miner: An infrastructure for measuring the quality of the npm registry," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 42–45.

[3] E. Wittern, P. Suter, and S. Rajagopalan, "A look at the dynamics of the javascript package ecosystem," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 351–361.

[4] R. Koschke, "Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 15, no. 2, pp. 87–109, 2003.

[4]Replication Package: `https://doi.org/10.5281/zenodo.7378331`

[5] R. Falke, R. Klein, R. Koschke, and J. Quante, "The dominance tree in visualizing software dependencies," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, 2005, pp. 1–6.

[6] A. Bergel, S. Maass, S. Ducasse, and T. Girba, "A domain-specific language for visualizing software dependencies as a graph," in *2014 Second IEEE Working Conference on Software Visualization*, 2014, pp. 45–49.

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

[8] D. Moreno-Lumbreras, J. M. Gonzalez-Barahona, and A. Villaverde, "BabiaXR: Virtual reality software data visualizations for the web," in *2022 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*. IEEE, 2022, pp. 71–74.

[9] R. Abdalkareem, V. Oda, S. Mujahid, and E. Shihab, "On the impact of using trivial packages: an empirical case study on npm and pypi," *Empirical Software Engineering*, vol. 25, no. 2, pp. 1168–1204, Mar 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09792-9

[10] A. Decan, T. Mens, and P. Grosjean, "An empirical comparison of dependency network evolution in seven software packaging ecosystems," *Empirical Software Engineering*, vol. 24, no. 1, pp. 381–416, Feb 2019. [Online]. Available: https://doi.org/10.1007/s10664-017-9589-y

[11] A. J. Jafari, D. E. Costa, R. Abdalkareem, E. Shihab, and N. Tsantalis, "Dependency smells in javascript projects," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3790–3807, 2022.

[12] R. Kikas, G. Gousios, M. Dumas, and D. Pfahl, "Structure and evolution of package dependency networks," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*, 2017, pp. 102–112.

[13] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" 2015.

[14] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[15] R. Elizalde Zapata, R. G. Kula, B. Chinthanet, T. Ishio, K. Matsumoto, and A. Ihara, "Towards smoother library migrations: A look at vulnerable dependency migrations at function level for npm javascript packages," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 559–563.

[16] A. Zerouali, V. Cosentino, T. Mens, G. Robles, and J. M. Gonzalez-Barahona, "On the impact of outdated and vulnerable javascript packages in docker images," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 619–623.

[17] S. Zhang, C. Demiralp, D. Keefe, M. DaSilva, D. Laidlaw, B. Greenberg, P. Basser, C. Pierpaoli, E. Chiocca, and T. Deisboeck, "An immersive virtual environment for dt-mri volume visualization applications: a case study," in *Proceedings Visualization, 2001. VIS '01.*, 2001, pp. 437–584.

[18] C. Demiralp, C. Jackson, D. Karelitz, S. Zhang, and D. Laidlaw, "Cave and fishtank virtual-reality displays: A qualitative and quantitative comparison," *IEEE transactions on visualization and computer graphics*, vol. 12, pp. 323–30, 05 2006.

[19] B. Laha, D. Bowman, and J. Socha, "Effects of vr system fidelity on analyzing isosurface visualization of volume datasets," *IEEE transactions on visualization and computer graphics*, vol. 20, pp. 513–22, 04 2014.

[20] E. D. Ragan, R. Kopper, P. Schuchardt, and D. A. Bowman, "Studying the effects of stereo, head tracking, and field of regard on a small-scale spatial judgment task," *IEEE Transactions on Visualization and Computer Graphics*, vol. 19, no. 5, pp. 886–896, 2013.

[21] J. J. Chen, H. Cai, A. P. Auchus, and D. H. Laidlaw, "Effects of stereo and screen size on the legibility of three-dimensional streamtube visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 2130–2139, 2012.

[22] P. Milgram and F. Kishino, "A taxonomy of mixed reality visual displays," *IEICE TRANSACTIONS on Information and Systems*, vol. 77, no. 12, pp. 1321–1329, 1994.

[23] R. Skarbez, M. Smith, and M. C. Whitton, "Revisiting milgram and kishino's reality-virtuality continuum," *Frontiers in Virtual Reality*, vol. 2, p. 647997, 2021.

[24] P. Kaiser, P. Vasak, F. Suorineni, and D. Thibodeau, "New dimensions in seismic data interpretation with 3-d virtual reality visualisation for burst-prone mines," 01 2005, pp. 33–45.

[25] A. Anderson and Z. Weng, "Vrdd: applying virtual reality visualization to protein docking and design," *Journal of Molecular Graphics and Modelling*, vol. 17, no. 3-4, pp. 180–186, 1999.

[26] S. Djorgovski, P. Hut, R. Knop, G. Longo, S. McMillan, E. Vesperini, C. Donalek, M. Graham, A. Mahabal, F. Sauer *et al.*, "The mica experiment: Astrophysics in virtual worlds," *arXiv preprint arXiv:1301.6808*, 2013.

[27] Z. Ibrahim and A. G. Money, "Computer mediated reality technologies: A conceptual framework and survey of the state of the art in healthcare intervention systems," *Journal of biomedical informatics*, vol. 90, p. 103102, 2019.

[28] S. Bryson, "Virtual reality in scientific visualization," *Communications of the ACM*, vol. 39, no. 5, pp. 62–71, 1996.

[29] T. Munzner, *Visualization analysis and design*. CRC press, 2014.

[30] S. Few, "Show me the numbers," *Analytics Pres*, 2004.

[31] A. Batch, A. Cunningham, M. Cordeil, N. Elmqvist, T. Dwyer, B. H. Thomas, and K. Marriott, "There is no spoon: Evaluating performance, space use, and presence with expert domain users in immersive analytics," *IEEE transactions on visualization and computer graphics*, vol. 26, no. 1, pp. 536–546, 2019.

[32] R. J. Jacob, A. Girouard, L. M. Hirshfield, M. S. Horn, O. Shaer, E. T. Solovey, and J. Zigelbaum, "Reality-based interaction: a framework for post-wimp interfaces," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, 2008, pp. 201–210.

[33] R. Rosenbaum, J. Bottleson, Z. Liu, and B. Hamann, "Involve me and i will understand!–abstract data visualization in immersive environments," in *International Symposium on Visual Computing*. Springer, 2011, pp. 530–540.

[34] R. J. García-Hernández, C. Anthes, M. Wiedemann, and D. Kranzlmüller, "Perspectives for using virtual reality to extend visual data mining in information visualization," in *2016 IEEE Aerospace Conference*, 2016, pp. 1–11.

[35] C. Knight and M. Munro, "Comprehension with[in] virtual environment visualisations," in *Proceedings Seventh International Workshop on Program Comprehension*, 1999, pp. 4–11.

[36] L. Merino, M. Ghafari, C. Anslow, and O. Nierstrasz, "CityVR: Gameful software visualization," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 633–637.

[37] D. Moreno-Lumbreras, R. Minelli, A. Villaverde, J. M. Gonzalez-Barahona, and M. Lanza, "Codecity: A comparison of on-screen and virtual reality," *Information and Software Technology*, vol. 153, p. 107064, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922001732

[38] K. Kobayashi, M. Kamimura, K. Yano, K. Kato, and A. Matsuo, "Sarf map: Visualizing software architecture from feature and layer viewpoints," in *2013 21st International Conference on Program Comprehension (ICPC)*, 2013, pp. 43–52.

[39] K. Yano and A. Matsuo, "Data access visualization for legacy application maintenance," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 546–550.

[40] M. Misiak, A. Schreiber, A. Fuhrmann, S. Zur, D. Seider, and L. Nafeie, "IslandViz: A tool for visualizing modular software systems in virtual reality," in *2018 IEEE Working Conference on Software Visualization (VISSOFT)*, 2018, pp. 112–116.

[41] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. De Water, "Studying pull request merges: a case study of shopify's active merchant," in *Proceedings of the 40th ICSE SEIP*, 2018, pp. 124–133.

[42] C. Maddila, C. Bansal, and N. Nagappan, "Predicting pull request completion time: a case study on large scale cloud services," in *Proceedings of the 2019 27th ESEC/FSE*, 2019, pp. 874–882.

[43] Y. Yu, H. Wang, V. Filkov, P. Devanbu, and B. Vasilescu, "Wait for it: Determinants of pull request evaluation latency on github," in *2015 IEEE/ACM 12th working conference on mining software repositories*. IEEE, 2015, pp. 367–371.