Visualization and Interaction for Program Comprehension in Virtual Reality

Academic year 2023-2024

Student: Giannaccari Mattia (869877)
USI Advisor: Prof. Dr. Lanza Michele
USI Co-Advisor: Raglianti Marco
UniMiB Co-Advisor: Prof. Dr. Arcelli Fontana Francesca

Double-Degree program:

Università degli Studi di Milano Bicocca
Dipartimento di Informatica, Sistemistica e Comunicazione
Corso di Laurea Magistrale in Informatica

Università della Svizzera Italiana
Faculty of Informatics
Master in Informatics

Abstract

Software systems are among the most complex artifacts created by humanity, characterized by numerous interacting components that must be understood and managed effectively. This thesis explores the integration of software visualization and virtual reality (VR) to enhance program comprehension. We present a novel tool that allows users to visualize and interact with software systems in an immersive virtual environment (VE), focusing on two main objectives: intuitive interaction and informative visualization. Our approach uses metaphors to shape user engagement with virtual objects, creating interactions that mirror real-world actions to minimize cognitive load. By moving away from traditional 2D user interfaces, we enable users to configure visualizations directly within the VR space, fostering a seamless immersive experience.

We discuss the historical context of software visualization and VR technologies, examining relevant research in human-computer interaction and cognitive science that informs our design choices. Through two case studies (a file system model and a Java project model) we show the capabilities of our tool and internally validate its effectiveness in facilitating insightful domain exploration. The findings suggest that integrating VR interactions with software visualization can significantly enhance user understanding of complex systems. This work contributes to the fields of software engineering and VR by providing a foundation for future research and development, offering new perspectives on the visualization and manipulation of software artifacts in VEs.

Contents

List of Figures

List of Tables

Figures Credits

Figure 2.8 https://blackcreek.ca/exhibits/stereoscope-and-stereograph/
Figure 2.9 https://en.m.wikipedia.org/wiki/File:Link-trainer-ts.jpg
Figure 2.10 https://osservatoriometaverso.it/sensorama-la-realta-virtuale-degli-anni-60/
Figure 2.12 https://commons.wikimedia.org/wiki/File:Sega_VR.png
Figure 2.13 https://www.pcgamesn.com/sega-vr
Figure 2.14 https://commons.wikimedia.org/wiki/File:Virtual-Boy-wController.jpg
Figure 2.15 https://uk.pcmag.com/gaming-gear/119528/
Figure 2.16 https://www.wired.com/story/meta-horizon-workrooms-glitchy-company-trials/
Figure 2.21 https://mx2games.com/product/meta-quest-touch-pro-controller/
Figure 4.1 https://get-it-easy.de/en/meta-quest-2-rent/
Figure 4.2 https://www.amazon.it/Pico-All-256GB-Headset-bianco/dp/B0BGLTNTCY
Figure 4.3 https://www.reddit.com/r/Pimax/comments/yr40ub/pimax_portal_keynotes/
Figure 4.4 https://www.playstation.com/it-ch/ps-vr2/
Figure 4.5 https://www.meta.com/it/quest/quest-3/
Figure 4.6 https://unity.com/features/profiling

Chapter 1

Introduction

Software systems are among the most intricate and complex creations ever developed by humans [13], composed of tens of thousands of interconnected components that interact with hardware, humans, and their surrounding environment during execution. Software visualization supports developers in understanding such complex systems by creating visual representations of relevant aspects of a software system under study [5]. This includes visualizing system architecture [6, 14], execution flow [15, 16], code evolution [17, 18], and code dependencies [19, 20] to help developers and researchers understand, analyze, and communicate about software systems. Moreover, software visualization serves as a visual documentation tool that can be easier to understand than textual documentation [21, 22], helping new developers, to quickly get up to speed with the codebase and system architecture.

Over the years, software visualization has evolved significantly. Early work focused on static 2D visualizations, often limited to on-paper representations, which helped document system designs and architectures. As technology advanced, dynamic visualizations that allowed for interaction became possible, offering a more engaging way to explore complex software systems in real time. In the late-1990s, the field has expanded into 3D visualizations, and more recently, virtual reality (VR), which brings new perspectives to how developers can experience and manipulate software. VR allows for the creation of immersive virtual environments (VEs), where users can physically walk through visualizations, interact with system components through natural gestures, and explore software in ways that traditional on-screen 2D or 3D visualizations cannot provide.

The immersive nature of VR offers distinct advantages in program comprehension. Unlike traditional methods that require a cognitive leap to connect abstract representations to real-world experiences, VR allows users to interact with virtual objects as they would with physical ones. For example, picking up and manipulating a virtual object in a VE mimics real-world interactions, making the experience more intuitive. This alignment with real-world behaviors helps reduce cognitive load and leverages the user's prior knowledge about interacting with physical objects. As a result, VR not only enhances visualization but also introduces new possibilities for how users can manipulate and engage with the elements of a software system.

In this work, we focus on two central aspects of VR's application to program comprehension:

- **Visualization**: We review existing literature on software visualization to understand how the virtual environment can be used to better represent software systems. This analysis includes identifying current gaps in visualization approaches, particularly how well these techniques exploit the immersive and spatial capabilities of VR;
- **Interaction**: We examine the different methodologies for interaction within a VE. Traditional VR approaches often involve replicating 2D user interfaces as virtual panels within the environment, but we aim to move beyond these limited methods by exploring more natural, intuitive forms of interaction. Specifically, we investigate how users can manipulate and explore a software system directly through interaction with virtual objects, mapping actions in the VE to meaningful operations on the software system's domain entities.

This research not only builds upon existing work in software visualization but also addresses the emerging need for more advanced, immersive interaction techniques that align with VR's unique

capabilities. By bridging the gap between visualization and natural interaction within VEs, we aim to offer developers more powerful and intuitive tools for understanding and working with complex software systems.

1.1 Contributions

The main contributions of this work are:

- 3D Selection Menu Design: We propose a novel approach for designing graphical user interfaces specifically for VR, moving away from traditional 2D UIs to enhance user interaction and immersion;
- IVAR-NI: A proof-of-concept for manipulating view specification through VR-native interfaces. Our approach minimizes context-switching by allowing users to configure visualizations directly within the VE, enhancing the immersive experience of interacting with the software system;
- VIRTEX: A tool for exploring a software system in VR. It allows visualizing a software system, inspecting its components and mapping cyber-physical interactions onto actions in the reference domain;
- City Layout Extension: We extend the city layout metaphor to create a generic domain layout, allowing any entity within the model to be represented as a building in the visualization, providing a compact and insightful way to visualize software systems;
- Model for a Visualization Tool: We present a mapping of domain entities to visual representations using software metrics, showing how visual properties convey relevant information about the system at a glance;
- Metaphor Utilization: We leverage metaphors to shape user perceptions of the VE and interactions with virtual objects, leading to a more intuitive user experience;
- Case Studies: An internal validation of the tool that shows the potential of this approach. In particular, we show how IVAR-NI and VIRTEX can aid a developer in exploring a file system and refactoring a java project.

1.2 Document Structure

This document is structured as follows:

Chapter 2 - Related Work : In this chapter, we review key areas of research that form the foundation for this thesis. First, we explore the history of software visualization, particularly its role in program comprehension and 3D visualization techniques. Next, we trace the evolution of VR technologies, from early stereoscopic photography to modern advancements. We then delve into the theoretical background of VR, focusing on human-computer interaction, cognitive science, and interaction paradigms specific to VR. Following this, we examine the hardware components of VR headsets and their functions, before concluding with a discussion on the tools, frameworks, and best practices for developing immersive VR applications;

Chapter 3 - Visualizing and Interacting with a Domain in VR : In this chapter we outline our approach to modeling a tool that enables users to visualize and interact with a reference domain in VR. We focus on developing metaphors that shape how virtual objects convey their functionality and how users engage with them, aiming to create interactions that mimic real-world actions and remain intuitive, drawing on principles from human-computer interaction and cognitive science. We leverage software metrics to build our visualizations, allowing for quick comprehension of relevant information;

Chapter 4 - Tool : In this chapter we present the incremental implementation of the tool, beginning with IVaR-NI, a proof-of-concept based on the 3D selection metaphor that allows users to manipulate view specifications and customize domain visualizations. This was followed by VirtEx, which visualizes domains using the view specifications from IVaR-NI and facilitates navigation within the VE. We then integrated both tools into a cohesive solution;

Chapter 5 - Case Study : In this chapter, we present two case studies to demonstrate the capabilities of the tool, which also serve as internal validations and guide future work. This also provides the main insights for the future developments;

Chapter 6 - Conclusions : In the final chapter, we reflect on the contributions of this thesis, summarizing the key takeaways from our work. We also outline potential avenues for future development, highlighting areas where further improvements or new features could enhance the tool's capabilities.

Chapter 2

Related Work

In this chapter, we review key areas of research that provide a foundation for understanding the context of this thesis. The chapter is structured into five sections, each focusing on a distinct area of the related literature:

- In Section 2.1, we present an overview of the history of software visualization, highlighting how developers have always needed visual aids to comprehend programs;
- In Section 2.2, we look at the history of VR technologies, from early 19th-century stereoscopic photography to recent advancements;
- In Section 2.3, we present the theoretical background behind VR, focusing on the findings in human-computer interaction and cognitive science literature that influence VR interaction. Then, we highlight different interaction paradigms, input methods, and user experience challenges specific to VR;
- In Section 2.4, we explain what are the hardware components of a VR system, how they work and in which way the influence the immersion;
- In Section 2.5, we examine the tools, frameworks, and best practices for building VR applications, focusing on the technical aspects of development and the challenges involved in creating effective, immersive VR experiences.

2.1 Software Visualization History

Visualization is the process of representing data, information, or concepts in a visual format, such as charts, graphs, diagrams, or maps. Representation of information in visual form is widely used in many fields, from mechanics to medicine and physics. Some information is easier to represent graphically, as it has an intrinsic geometry or idea within it (*e.g.,* the structure of the nervous system, a gear in a mechanical system) [5]. Some information is harder to represent, being abstract and intangible such as the behavior of a software component [13].

Software is inherently complex and abstract. It is complex because it consists of many different components that communicate with each other, with the hardware on which they run and with the outside world [13]. Moreover, no two components are the same. If they are, one tends to extract the shared part and abstract [13]. It is abstract because there is no immediate, concrete representation of some aspects [5]. The behavior of a software component, for instance, is an abstract concept and can be visualized through graphical abstractions [23].

Software visualization refers to the representation in visual form of some aspects of the software, such as code, execution flow or data flow [15, 16, 24]. This includes visualizing system architecture [6, 14], code evolution [17, 18], and code dependencies [19, 20]. The focus is to help developers and researchers to understand, analyze, and communicate about software systems. Moreover, software visualization serves as a documentation tool [21, 22] that can be easier to understand than textual

documentation, helping new developers to quickly get up to speed with the code-base and system architecture.

Software visualization has become a critical tool in understanding, analyzing, and managing complex software systems [25, 26]. Von Mayrhauser and Vans describe program understanding as a major factor in providing effective software maintenance and enabling successful evolution of computer systems, using existing knowledge about the system to build new knowledge about the system itself [27]. As software systems grow in complexity, the need for effective visualization tools that allows visualizing both static and dynamic aspects of the software systems becomes increasingly vital [24].

2.1.1 Early History (1940s-1970s)

The concept of software visualization dates back to the early days of computing. During this period, programmers relied primarily on textual and mathematical representations of code, which were often cumbersome and difficult to understand. As software systems grew in complexity, the need for more effective ways to understand the code and manage the code-bases became apparent.

Electronic Numerical Integrator and Computer (ENIAC) was the first general-purpose electronic digital computer, built between 1943 and 1945 at the University of Pennsylvania [28]. General-purpose means it could be programmed to accomplish different tasks. Initially, ENIAC was programmed manually by physically rewiring the machine. Programmers would set switches, plug and unplug cables, and adjust knobs to change the configuration of its circuits. Hence a program written for ENIAC already had an explicit visualization (*i.e.,* the wiring diagram of the cables) [1], as show in Figure 2.1.

One of the earliest forms of software visualization was the *flowchart*, introduced by Goldstine and von Neumann in the late 1940s. Flowcharts provided a graphical representation of the logical flow of a program, making it easier for programmers to understand the structure and logic of their code [2], as shown in Figure 2.2. Despite their simplicity, flowcharts laid the foundation for more sophisticated visualization techniques.

Another significant development in the early stages was the creation of *control flow graphs*, which represent the flow of control in a program, as shown in Figure 2.3 [3]. Introduced by Allen in the 1970s, these graphs became a fundamental tool for compiler optimizations and are still widely used in various forms today, such as in [29, 30].

The introduction of *entity-relationship diagrams* (ERDs) by Chen in 1976 also provided a way to model the data structures within a system. ERDs visually represented the relationships between different entities in a database, as shown in Figure 2.4, making it easier to design and understand complex database systems [4].

Flow charts, control flow graphs and ERDs show how, even in the early stages of programming, there was the need for a visual representation of programs, to aid the comprehension of the source code.

2.1.2 First GUI and Collaborative Tools (1980s-1990s)

In the early 80s, Graphical User Interfaces (GUIs) became popular and revolutionized the way users interacted with computers, making them more accessible and user-friendly [31]. This period also saw the emergence of early software visualization tools designed to leverage the capabilities of GUIs.

TANGO is a framework for algorithm animation presented by Stasko in 1990 [32]. Algorithm animation is a technique used to visualize the operations of algorithms, helping users to understand complex algorithms by showing the step-by-step execution through graphical representations. A frame of such an animation is shown in Figure 2.5

Another notable example is the *SeeSoft* tool developed by Eick *et al.* in the early 1990s, a screenshot of which is shown in Figure 2.6. SeeSoft visualized large software systems by representing lines of code as colored pixels, allowing users to quickly identify patterns, anomalies, and areas of interest [33]. This approach marked a significant break from traditional text-based representations, highlighting the potential of graphical visualization techniques.

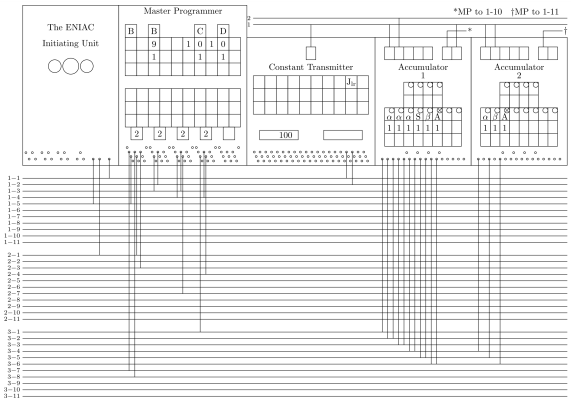

Figure 2.1: A schematic representation of the status of the hardware while programming ENIAC [1].

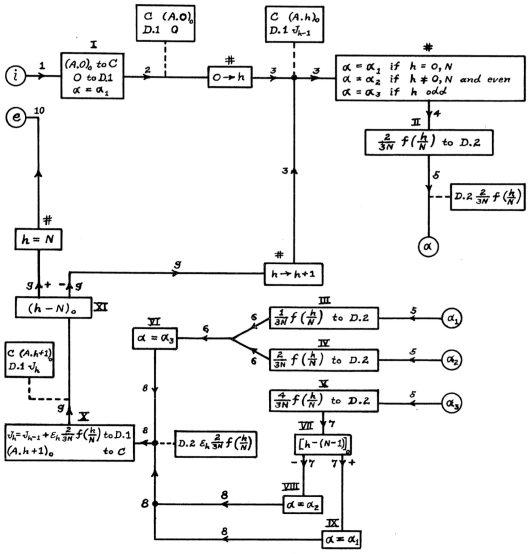

Figure 2.2: The example with which Goldstine and von Neumann introduced the flowchart [2].

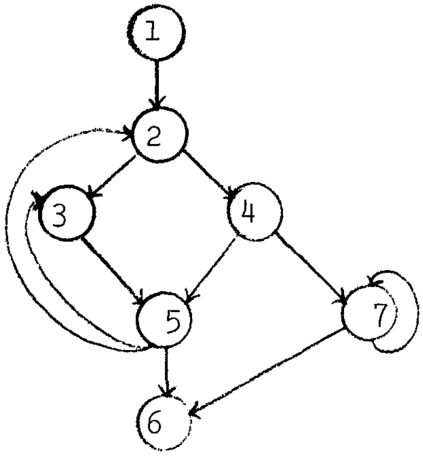

Figure 2.3: The example with which Allen introduced the control flow graph [3].

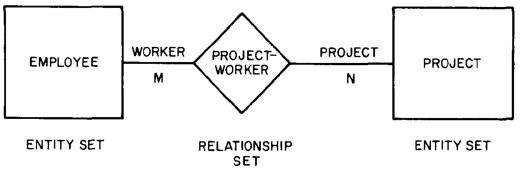

Figure 2.4: The example with which Chen introduced the ERD [4].

During this era, other interactive tools for visualizing the development process were also introduced to help project managers visualize and manage the timeline and dependencies of software projects. *Gantt* charts and *PERT* diagrams for example, visualize the tasks in a project schedule and the dependencies between the tasks. These tools provided a high-level overview of project progress and resource allocation, contributing to more efficient project management, highlighting once again the complexity not only of software systems themselves, but also of the activities involved in its development.

Advent of the Internet in the 1990s and the subsequent rise of the open-source movement had a deep impact on software development and visualization. The proliferation of online collaboration tools and version control systems enabled developers to work together on large-scale projects, often involving hundreds or thousands of contributors.

Version control systems (*e.g.,* git, mercurial) are tools designed to manage and track changes to files over time. They allow multiple users to collaborate on a common shared status of the code-base by recording modifications, maintaining a history of changes, and enabling users to revert to earlier versions if needed. They became particularly essential in software development, where code-bases are constantly evolving and multiple people may be working on the same code simultaneously. This led to the natural emergence and evolution of standardized common languages to support collaborative

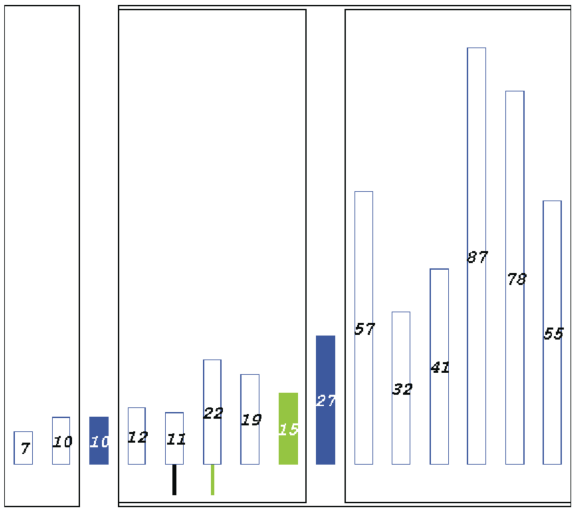

Figure 2.5: A frame of the animation produced by TANGO for the quick-sort algorithm [5].

Figure 2.6: The GUI of the SeeSoft visualization tool [5].

development and code analysis.

Unified Modeling Language (UML), standardized in the late-1990s [1], is a visual language for representing both static and dynamic aspects of a software. It provides a unified framework to represent and communicate complex software systems. It became so important that Watson divided visual modeling into two eras: Before UML and after UML [34]. Developed by Booch *et al.* [35], UML consolidated various modeling methodologies into a single, standardized language, greatly enhancing the software engineering process. It is crucial because it offers a comprehensive and versatile suite of visualization techniques that cater to different aspects of software development, helping with clear and consistent communication among team members, regardless of their location or background. By using UML, developers can create visual representations that encapsulate both the structural and behavioral dimensions of a system, bridging the gap between abstract concepts and practical implementation.

2.1.3 Recent Developments

In the past few decades software visualization has become an integral part of software engineering, with a wide range of tools and techniques available to developers. Some tools visualize the static structure of a software system, for example:

- **Simple Hierarchical Multi-Perspective (SHriMP) Views** is a tool developed by Storey *et al.* that uses nested graph visualizations to represent the hierarchical structure of software systems, allowing users to zoom in and out of different levels of abstraction. This tool supports the comprehension of large and complex software systems by providing multiple perspectives and interactive navigation capabilities [36];

- **CodeCrawler** is a software visualization tool designed by Lanza to support the analysis and understanding of complex object-oriented software systems [37]. It uses visual representations to help developers explore the structure, relationships, and metrics of source code. By using polymetric views, CodeCrawler allows users to map metrics on position, size, and color of an entity, as well as to visualize relationships between different entities.

While SHriMP Views and CodeCrawler focus on static aspects, other tools focus on dynamic aspects such as evolution of the code-base and runtime, for example:

- **EXTRAVIS** is a dynamic analysis tool designed by Cornelissen *et al.* to visualize the evolution and runtime behavior of software systems. It provides visual representations of object

[1] See `https://www.omg.org/spec/UML/1.1`

interactions, method invocations, and other runtime events, aiding developers in understanding the dynamic aspects of their software. It proves particularly useful for performance analysis and debugging [38];

- **Gource** is a tool developed by Caudwell thet visualizes the history of a software project as a tree structure, showing how files and directories evolve over time [39].

Apart from two-dimensional visualizations, researchers also explored three-dimensional ones. For example, the *CodeCity* tool, introduced by Wettel and Lanza in 2008, used a city metaphor to represent software systems as 3D cities. Buildings represent classes, and their height and base size can be mapped to software metrics [6]. Such innovative visualization techniques help developers create custom information-rich visualizations, while allowing navigation and understanding of extensive code-bases more effectively. The CodeCity GUI is shown in Figure 2.7. This approach is significantly different from the one of other solutions. While tools like UML use abstract graphical symbols, other tools employ metaphors to leverage the developer's existing real-world knowledge. These approaches reduce cognitive load and learning curve by minimizing the imposition of new formalisms in favor of intuitive similarities with real-world objects. This approach makes it more intuitive for developers to understand and work with complex software systems [5].

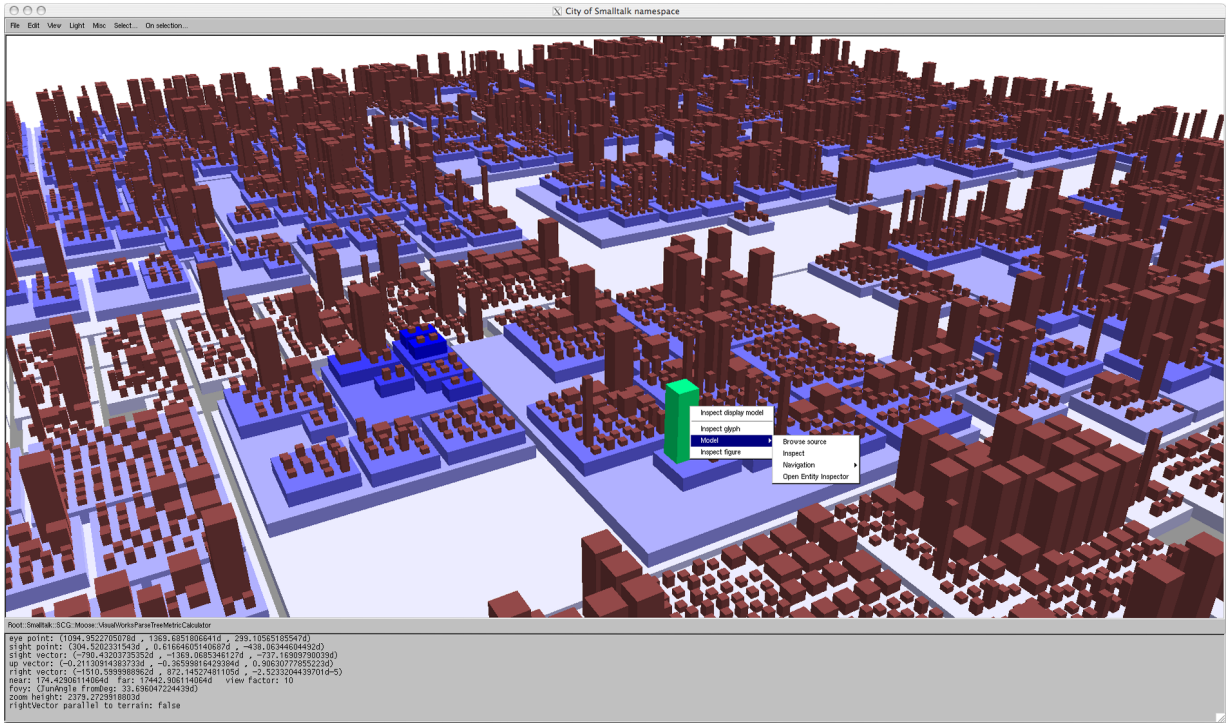

Figure 2.7: The CodeCity visualization tool GUI [6].

Recent developments go even deeper, exploring further aspects of software related not only to code but also to the development process and documentation. DiscOrDance, for example, is a tool by Raglianti *et al.* for the interactive visual exploration of the complete message history of a Discord server [21], which is an informal form of documentation [40]. It uses the concept of *viewspecs* (*i.e.,* configurable view specification), as introduced by [37], which allow customization of the view through a simple internal DSL, in this case using the Pharo[2] syntax.

Modern visualization tools leverage advances in computing power, data analytics, and artificial intelligence to provide more sophisticated and interactive visualizations. One prominent example is *SonarQube*[3], an open-source platform that provides continuous inspection of code quality. SonarQube uses visualization techniques to highlight code smells, bugs, and security vulnerabilities, enabling developers to address issues proactively. Tools like SonarQube demonstrate the ongoing evolution

[2]See `https://pharo.org/`
[3]See `https://www.sonarsource.com/products/sonarqube/`

of software visualization in response to the increasing complexity and demands of modern software development.

The future of software visualization is likely to be shaped by emerging technologies such as VR and AR, although the advantages of VR over non-VR visualizations are still debated [41]. These technologies have the potential to create more immersive and interactive visualization experiences, further enhancing our ability to understand and manage complex software systems. For instance, VR can provide a three-dimensional space for developers to explore software architectures and code-bases in an intuitive and immersive manner. This is leading to new ways of visualizing and interacting with software components [42–47], making the analysis and debugging processes more efficient and effective.

A recent example of such tools is ISA VR, a tool by Hoff *et al.* that enables teams of developers, each using their own VR device, to collaboratively explore software systems in a shared VE [48]. During exploration sessions, they can capture thoughts and insights through shared VR whiteboards using freehand sketching, audio recordings, and in-visualization screenshots. The method works on both local networks and over the internet, making it possible for distributed teams to meet in a VE to examine a system's architecture, discuss ideas, and take notes collaboratively.

2.2 Virtual Reality History

VR has transformed from a futuristic concept in science-fiction movies and video games to a widely adopted technology with applications spanning entertainment, education, healthcare, and much more. From rudimentary proof-of-concepts to the highly immersive systems we see today, VR has been shaped by advances in computing, optics, motion tracking, and human-computer interaction.

2.2.1 Early History (Pre-1950s)

The earliest precursors to VR were non-digital attempts at immersion. Panoramic paintings, which date back to the 1800s, were designed to envelop viewers in large-scale scenes that created the illusion of a real environment.

In parallel, stereoscopic photography was developed in the mid-19th century. The *stereoscope*, invented by Wheatstone in 1832, used two slightly different images of the same scene, one for each eye. The slightly different perspective between the two images creates a three-dimensional effect, leveraging the same visual principles of the human visual system on which modern VR is also based. Such a device is shown in Figure 2.8.

The concept of simulating an environment for training or entertainment purposes began with early flight simulators. In the late 1920s, Link's *Link Trainer* became a key tool for pilot training. This mechanical device used motion and instruments to simulate flying conditions, as shown in Figure 2.9, inspiring later VR systems focused on immersion and simulation.

Figure 2.8: An early stereoscope device for perceiving depth in photos.

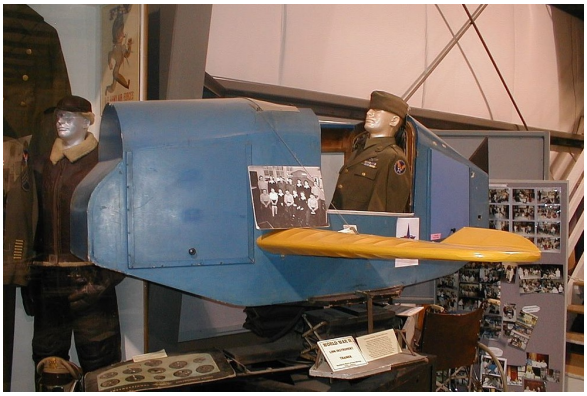

Figure 2.9: The Link Trainer flight simulator for pilots training.

2.2.2 First VR Devices (1950s-1960s)

Heilig is considered one of the pioneers of VR. He invented in 1960 the *Telesphere Mask*, the first known head-mounted display (HMD) [49], which presented stereoscopic 3D images but lacked motion tracking and interactivity. In 1962, he designed *Sensorama*, a machine that stimulates multiple senses simultaneously [50], as shown in Figure 2.10. Sensorama featured stereoscopic 3D visuals, stereo sound, vibrations, and even scents to immerse the user in different experiences, such as riding a motorcycle through Brooklyn. Although it was not interactive, it was one of the first attempts at a multi-sensory immersive experience.

In 1968, Sutherland and his student Sproull developed the first HMD system with motion tracking, shown in Figure 2.11. It was known as the *Sword of Damocles* due to its intimidating appearance and the heavy hardware required to suspend it from the ceiling [51]. Despite its rudimentary graphics and limited functionality, this system endeavors regarding motion tracking and real-time rendering in VR.

Figure 2.10: Sensorama, a machine for multi-sensory immersive experiences.

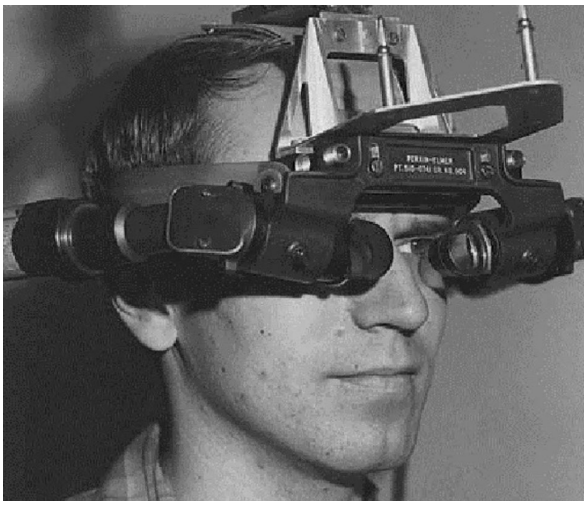

Figure 2.11: The first HMD with motion-tracking, also known as the Sword of Damocles.

2.2.3 VR in Simulations (1970s-1980s)

During the 1970s, VR research was primarily driven by military and academic institutions. The U.S. Department of Defense invested heavily in flight simulators and training systems for soldiers. The creation of more sophisticated simulators such as the *McDonnell-Douglas VITAL* (Visual Imaging Technology for Autonomous Lifelike simulation) marked significant progress in visual and motion systems, allowing for more realistic training environments. NASA also became a major player in VR development during the 1980s. Their Ames Research Center worked on VR displays and motion tracking for training astronauts. NASA's *Virtual Interface Environment Workstation* (VIEW) system was one of the first fully immersive VR setups, using an HMD and gloves to interact with simulated environments.

Throughout the 1980s, VR research expanded significantly, driven by advancements in computing power and human-computer interaction. The term *Virtual Reality* gained popularity in the 1980s by Jaron Lanier, who founded VPL Research, one of the first companies to sell VR headsets (*e.g.,* the *EyePhone*) and gloves (*e.g.,* the *DataGlove*) [51]. Lanier's vision and products helped shape public and professional interest in VR, leading to significant advancements in the 1990s. VR technology, at this stage, was still very expensive and remained primarily accessible only to research institutions and military applications.

2.2.4 The Commercialization of VR (1990s)

The 1990s saw the first major attempt to bring VR to the consumer market, although these efforts largely failed due to technical limitations and high costs. Sega, for instance, announced in 1991

a prototype VR headset for the Sega Genesis, shown in Figure 2.12, which generated significant excitement but was never released. Although the graphics of the games, such as the one shown in Figure 2.13, were not realistic at all, they were still very advanced for that time. Similarly, Nintendo released in 1995 the Virtual Boy, shown in Figure 2.14, a console-based VR system that projected red-and-black 3D graphics, like the game shown in Figure 2.15. However, it was a commercial failure due to poor ergonomics, uncomfortable gameplay, and limited graphical capabilities.

VR also made its way into arcades. Companies like Virtuality offered arcade machines with stereoscopic 3D visuals, motion tracking, and 3D sound. These systems, while expensive and cumbersome, allowed players to enter 3D game environments and manipulate objects with special controllers.

Figure 2.12: The Sega VR headset for the Sega Genesis console.

Figure 2.13: A flight simulator for the Sega Genesis and the Sega VR.

Figure 2.14: The Nintendo Virtual Boy console.

Figure 2.15: A game for the Virtual Boy console.

While consumer VR struggled in the 1990s, research and military applications of VR kept flourishing. The U.S. military continued to use VR for flight and battlefield simulations. Academic research also began to focus more on understanding how VEs could be used for training, psychological therapy, and social interaction. Notable research included the use of VR in treating phobias and post-traumatic stress disorder.

2.2.5 VR's Winter and Resurgence (2000s-2010s)

After the hype and failures of the 1990s, VR entered a period of decline. Consumer interest waned, and investment in the technology slowed. However, research continued in certain fields, including medical

simulations, industrial design, and military training [52–55]. VR began to be applied to therapies such as exposure therapy for phobias [56, 57], showing that the technology had important real-world applications outside of gaming. VR's resurgence began in 2012 when Palmer Luckey introduced the prototype for the Oculus Rift on Kickstarter [4]. The Rift promised a much higher-quality immersive experience than earlier consumer products, thanks to advancements in display technology, motion tracking, and computing power. Oculus's success reinvigorated interest in VR, and the company was acquired by Facebook (now Meta) in 2014 for $2 billion [5].

Following the Rift, other companies entered the VR market. HTC partnered with Valve to create the HTC Vive, which was launched in 2016 and featured room-scale tracking, allowing users to move around in a VE. Sony released the PlayStation VR in 2016, leveraging the PlayStation 4 console to bring VR to a mass audience. Around the same time, mobile VR devices gained popularity. Products like Samsung Gear VR and Google Cardboard allowed users to experience VR using their smartphones. While these systems lacked the full immersion of the Rift or Vive, they offered a more affordable and accessible entry point for many consumers. The Google Cardboard, for example, cost between $20 and $30 when it was released [6].

2.2.6 Recent Developments

One of the key innovations in the 2020s has been the development of standalone VR devices, which do not require a PC or external sensors. The Oculus Quest, released in 2019, was a major step forward in this regard, combining portability with high-quality VR experiences. The Quest 2, released in 2020, became even more successful, further increasing the potential user base for VR technologies. Beyond entertainment, VR has found new applications in fields such as healthcare, education, design, and training. Enterprise VR platforms like Varjo, which offer ultra-high-definition displays, are used for industrial design and engineering. VR is also used for virtual meetings and collaboration, with platforms like Horizon Workrooms [7] aiming to create more immersive remote work experiences, as shown in Figure 2.16.

Figure 2.16: A meeting taking place within Horizon Workroom.

The concept of the metaverse, a persistent, interconnected virtual world, is emerging again. Com-

[4] See `https://www.kickstarter.com/projects/1523379957/oculus-rift-step-into-the-game`
[5] See `https://about.fb.com/news/2014/03/facebook-to-acquire-oculus`
[6] See `https://www.cnet.com/reviews/google-cardboard-review/`
[7] See `https://forwork.meta.com/it/horizon-workrooms/`

panies like Meta and others are investing in developing the metaverse, where users can interact, work, and socialize in shared VEs. Although not a new concept [8], new developments in the metaverse could define the future of VR and augmented reality (AR), blending virtual and physical worlds.

VR has also found a role in research involving software visualization, offering immersive ways to analyze complex software systems and data structures. In traditional 2D environments, understanding intricate relationships within large codebases, debugging, or analyzing system performance can be challenging. VR allows researchers and developers to visualize code in 3D spaces, where they can interact with and manipulate representations of software architecture, dependencies, and flow [19, 43, 58]. This enhances spatial understanding and pattern recognition, enabling users to spot issues, optimize systems, and better grasp the structural dynamics of software, particularly in large-scale or distributed systems [59].

2.3 Virtual Reality Theoretical Background

In this section we present the findings in human-computer interaction and cognitive science that shape how a user perceives the VE and interacts with it. We first present the three "I"s usually used to describe VR technology and then we talk about the purpose of the actions of a user in a VE. We finally present the input methods commonly used for interacting with a VE.

2.3.1 The Three "I"s of VR

So far we have left the definition of certain concepts to the reader's intuition. The very concept of VR can be as intuitive as hard to grasp its depth. Virtual Reality is an oxymoron, because something virtual is inherently not real. Let us now give a more precise definition of what we mean by VR:

> *VR is a user-computer interface that involves real-time simulation of realistic or artificial realities and interactions through multiple sensorial channels. These sensorial modalities involve sight, hearing, touch, smell, and taste [7].*

where *real-time* means that the VR system is able to detect user input and modify the VE accordingly. From the previous definition is clear that a virtual experience is interactive and it also gives an implicit definition of immersion (*i.e.,* a mental involvement in something). VR is often defined as a triad of "I" [7], with Interaction and Immersion being just two of the three, indeed the most intuitive to grasp. The third "I" is the Imagination, as shown in Figure 2.17. It refers to the ability of the VR tool to solve a particular problem, as well as to the ability of our brain to fill the gaps in the imperfect sensorial information it receives [7]. For instance, there is no triangle in Figure 2.17, there are just three circles with three lines between them. This is an example of the Gestalt principle of *closure* [60]. Our brain uses the imagination to group the three lines in a single triangle behind the three circles, filling the gaps in the geometry.

Interaction and immersion are tightly coupled, mutually affecting each other, and influencing the imagination aspect. This section is structured as follows:

- In Sub-Section 2.3.1 we give the definition of affordance, feedback and ergonomics, all aspects that influence the effectiveness of the interaction with the VE;
- In Sub-Section 2.3.1 we give a definition of immersion and presence, then we explain how visual fidelity, interaction fidelity and responsiveness influence immersion;
- In Sub-Section 2.3.1 we give an idea of what cognitive science is about and how Gestalt psychology can help filling the gaps between the imperfect sensorial information received by the VE;
- In Sub-Section 2.3.2 we present how using metaphors that mimic the real world can help the user interact with the VE in an intuitive way;

[8] See `https://en.wikipedia.org/wiki/Second_Life`

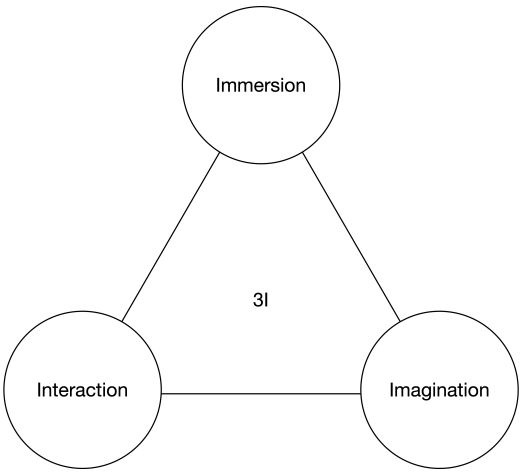

Figure 2.17: The three "I"s of VR [7].

- In Sub-Section 2.3.3 we present some of the well-known design guidelines from the literature;
- In Sub-Section 2.3.4, we present different interaction methods for VR.

Interaction

Effective interaction design in VR relies on theoretical concepts from human-computer interaction, such as affordances, feedback and ergonomics.

> *Affordance means that things may suggest through their design attributes what you can do with them [61].*

It describes the ways in which a user can interact with a system based on its design (*e.g.,* a button affords pressing, a handle affords pulling). Affordances help users understand the potential actions available without needing explicit instructions, making systems more intuitive to use. Affordance frameworks provide the starting point for designing intuitive interactions in VR [62, 63].

> *Feedback is the system's response to user's actions, informing the user that the input has been received [64].*

It provides confirmation and guide the next steps. They can be visual (*e.g.,* a button changing color when pressed), auditory (*e.g.,* a sound indicating completion), or tactile (*e.g.,* a vibration). Effective feedback helps users understand the system's state and behavior, making interactions smoother and reducing errors. VR can benefit from not only visual but also vibrotactile feedback. It has been shown that haptic feedbacks greatly improve user's performance when performing a task VR [65]. It also improves immersion, giving a stronger sense of presence, as we will discuss in Sub-Section 2.3.1.

> *Ergonomics refers to the design of systems, devices, and interfaces that prioritize user comfort, efficiency, and safety [66].*

It involves creating tools that align with human physical and cognitive capabilities, minimizing strain or discomfort during use. Ergonomics considers factors like the layout of controls, the ease of interaction, and the physical fit of hardware or interfaces to ensure optimal user experience, performance, and well-being over prolonged use.

Interacting with the VE includes traveling within it, selecting and manipulating 3D objects and controlling the system status. Interactions can occur through physical movement into the real world, by means of the controllers or by using hand gestures. This great variability, combined with the need

to avoid physical load to complete certain actions and the physical impairment of some users, makes the interactions aspect even more delicate [67, 68]. VR devices have evolved over time precisely to make the most of affordances, feedbacks and ergonomics.

The concept of interacting with a virtual GUI in a way that mimics real-world interactions dates back to the 1960s. Ivan Sutherland, the same who developed the above mentioned Sword of Damocles, developed a software known as *Sketchpad* in 1963, the forerunner of modern GUIs. It enabled drawing on a display using a pen in exactly the same way as on paper, as shown in Figure 2.18, pioneering human-machine interaction [69].

Figure 2.18: Ivan Sutherland using the Sketchpad [8].

Throughout the following decades, advancements in computing power, sensor technology, and interface design propelled the evolution of interaction in VR. Devices such as the Data Glove and early motion-tracking systems, developed by pioneers like Jaron Lanier and his team at VPL Research in the 1980s, introduced the concept of using hand movements and gestures to manipulate objects within VEs [51]. These innovations laid the groundwork for more intuitive and natural interactions in VR.

Data gloves, like those developed by VPL Research, were among the earliest devices designed for tactile interaction in VR. These gloves used sensors to detect finger movements and gestures, allowing users to manipulate virtual objects with their hands. Similarly, today's hand-held controllers have evolved to offer precise tracking and intuitive button layouts, enhancing user interaction in VR applications.

Immersion

Immersion and *presence* are often confused or used interchangeably but they represent two distinct concepts [70].

> *Immersion is the objective quality of what the technology provides. A system is more immersive when it delivers sensory displays (across all modalities) and tracking that accurately maintains the fidelity of real-world sensations [70].*

> *Presence is a human subjective psychological response to the immersion in a VR system [70].*

Immersion is objective and hence measurable. A system can be more immersive than another one. Considering only the visual immersion, it is directly affected by how the VE is displayed and updated [71]. For instance:

- The display resolution is the quality at which the image is displayed, affecting the realism of the VE;
- The refresh rate is the number of times per second that the image on the display is refreshed, affecting responsiveness;
- The realism of lights and shadows affects the realism of the whole VE;

Advances in motion-tracking technology have been crucial in enhancing the sense of presence in VR environments [72]. From early optical systems to modern tracking solutions using cameras and sensors, these technologies enable accurate positioning and movement detection within VEs.

Interaction fidelity also affects immersion [73]. It has been shown that as the appearance of a robot becomes more human-like, emotional affinity increases, as shown in Figure 2.19 [9]. Once it reaches a point of near-human realism, the slight imperfections create discomfort. The *uncanny valley* refers to the unsettling feeling people experience when a humanoid figure or robot looks almost, but not quite, human. Beyond this valley, if the likeness becomes indistinguishable from a real human, the sense of unease diminishes.

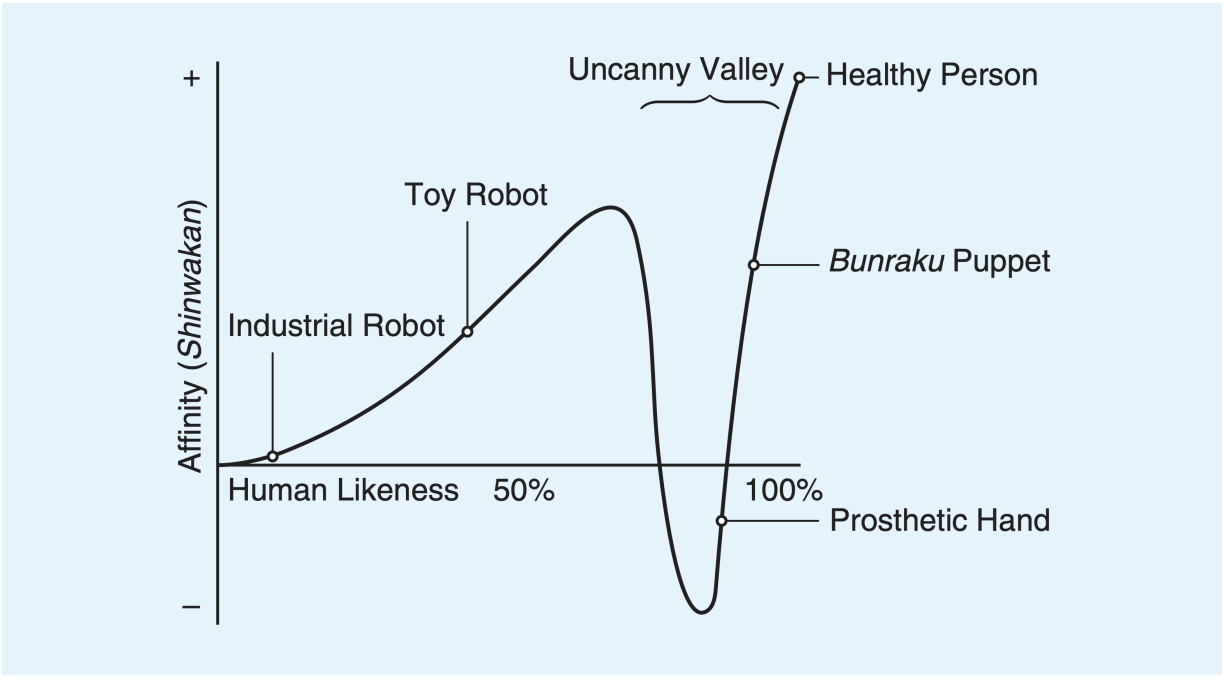

Figure 2.19: The plot of the uncanny valley [9].

This is also true for interaction: Users perform better in low-fidelity and high-fidelity interactions than in mid-fidelity interactions [73].

While immersion is an objective aspect, directly influenced by the properties of the system, presence is subjective. Different people can experience varying levels of presence, even when using the

same immersive system. Similarly, different immersive systems can evoke similar levels of presence in different individuals. While presence and immersion are conceptually distinct, they are likely closely related in practice. A key aspect of studying presence is to explore this relationship and understand how different factors contribute to an individual's sense of *being there* in a VE [70].

Imagination

While in a VE, our brain fills the gaps when sensory information are incomplete or ambiguous [7]. VR systems strive to provide realistic and immersive environments, but anyways the human brain often receives limited or imperfect information, such as lower resolution graphics, less tactile feedback, or incomplete sensory data.

Active imagination is fundamental in overcoming disbelief to achieve true immersion in VR. This involves the observer consciously placing themselves in a receptive mindset, similar to the process of cooperating with a hypnotist. While this phenomenon is not new (people naturally engage in this when reading a captivating book or watching a film) it requires a heightened level of attention and imagination in VR. Users must actively construct mental models of what they see and hear while deliberately ignoring cues that indicate the VR experience is not real [74].

Obviously, the way in which a virtual experience is delivered greatly influences the effort a user has to make to achieve this. A realistic VE for example would require less effort. Exploiting the capabilities of the brain of subconsciously filling some gaps makes the whole virtual experience more realistic with less explicit effort required by the user.

2.3.2 Intentionality and Naturalness

Intentionality and naturalness are related not to *how* an interaction takes place, but to *why* it takes place. An *intention* is in fact defined as:

> *An intention is a specific action required to achieve a goal [61].*

Thus, intentionality concerns the relationship between action and meaning [75]. This involves creating interfaces where users can intuitively link an interaction to the desired effects [76], in order to understand what interaction is needed to obtain the desired output. We can define intentionality as:

> *Intentionality refers to the purposeful direction of a user's actions or behaviors towards achieving a specific goal when interacting with a system [61].*

To achieve intentionality, it's crucial to design interfaces that provide clear and perceptible affordances and feedbacks. While affordances guide users in understanding how to interact with different elements [77], a feedback reinforces the user's understanding and expectations of the action's outcome.

Feedback plays a significant role in supporting intentionality. When users perform an action, immediate and clear feedback helps them confirm that their intention has been recognized and executed correctly by the system. For example, when a user deletes a file, a confirmation message or sound can reassure them that the action has been successfully completed. Consistency across the interface further supports intentionality. When similar actions have similar results throughout the system, users can rely on their previous experiences to predict outcomes, which makes the interaction more efficient. For example, if a swipe gesture is used to scroll through content in one part of an application, it should work similarly in other parts.

Designing for intentionality also involves understanding the context in which users operate. This means considering the users' goals, their environment, and the specific tasks they aim to accomplish. Selecting a specific object in VR, for example, can be challenging. When using ray-casting, the user interacts with object through a ray projected from the controller. As the ray collides with an object, the user can interact with that object. This approach is useful for interacting with distant object,

but it also makes hard to aim at very distant object. Even a small movement of the controller, in fact, would result in a large displacement of the beam, making it hard to hit the target precisely. A possible solution that uses affordances is the use of the well-known magnetic ray metaphor shown in Figure 2.20. It allows the ray to magnetically snap to the closest object, keeping it selected even when the ray is no longer pointing exactly at the object [10].

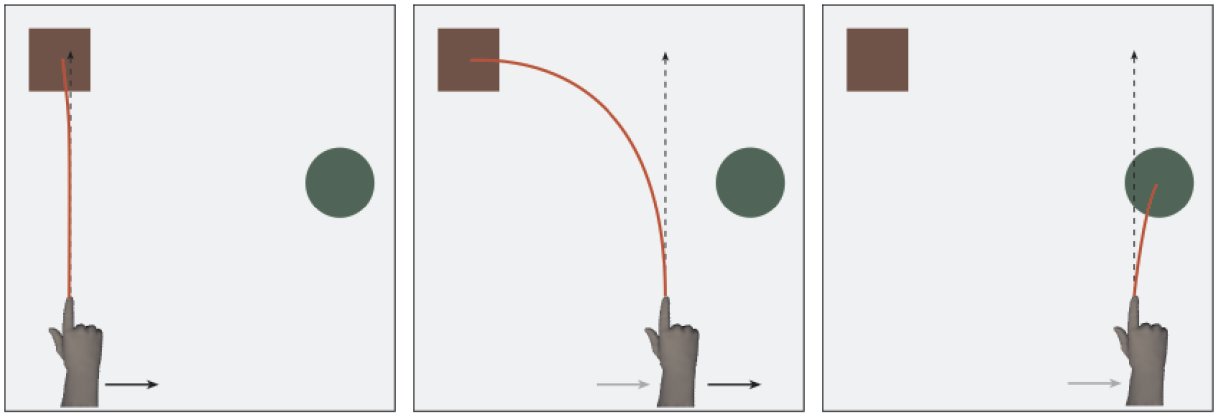

Figure 2.20: How the ray moves between the virtual objects using the magnetic ray metaphor [10].

Naturalness refers to the degree to which interactions with a digital system feel intuitive and closely mimic real-world behaviors and experiences:

> *Naturalness is how people use an interactive product: How naturally it feels in their hands, how naturally they understand its function from their existing skills, how natural do they feel while using it, and how naturally does it fits into their usage context [78].*

Naturalness is hence about designing interfaces that allow users to interact with technology in a way that feels seamless and instinctive, reducing the learning curve and enhancing user satisfaction. Natural interactions recall familiar gestures, behaviors, and sensory cues to create an experience that feels organic.

Hand-tracking technology, for instance, is a prime example of promoting naturalness in interaction. By allowing users to manipulate virtual objects with their hands, as they would in the physical world, the technology reduces the abstraction layer between the user and the system. Eye-tracking technology also contributes to naturalness by enabling gaze-based interaction. This technology can make navigation and selection processes more intuitive. For example, in a VR environment, a user can select objects or navigate menus simply by looking at them, mirroring the way we use our gaze to direct attention in the real world. Voice interaction enhances naturalness by enabling users to perform tasks using spoken commands. This can be particularly effective in hands-free scenarios, such as when users are driving or cooking. For instance, a user can ask a virtual assistant to set a timer, play music, or search for information, using natural language without needing to touch any device.

Naturalness is also supported by realistic feedback mechanisms. Haptic feedback, which provides tactile sensations in response to user actions, can make interactions with digital elements feel more like interactions with physical objects. For example, when typing on a virtual keyboard, slight vibrations can simulate the feeling of pressing real keys.

2.3.3 Well-Known Design Guidelines

Guidelines for 3D and VR interaction focus on improving the usability of a system by correctly implementing affordances, feedback and ergonomics, as well as intentionality and naturalness. This approach also minimize the physical effort required to perform an action and improves accessibility. General guidelines include [79]:

- **Affordances**: Clearly indicate interactive elements through visual cues, such as highlighting or animation;

- **Feedback**: Provide immediate and informative feedback to user actions to reinforce interaction outcomes;
- **Consistency**: Maintain consistent interaction patterns throughout the experience to reduce cognitive load and facilitate learning;
- **Accessibility**: Ensure that interactions are accessible to users with diverse abilities (*e.g.,* physical, cognitive and sensory abilities) and preferences through adaptable interfaces and inclusive design practices.

Some guidelines are specific to VR. The user moves through physical space while observing a virtual space, so guidelines are needed to ensure safety and reduce motion sickness. These include:

- **Comfort**: Design for comfort to minimize motion sickness and physical discomfort during extended VR sessions;
- **Presence**: Enhance the feeling of presence through realistic interactions and environments that respond predictably to user actions;
- **Safety**: Consider physical safety within VEs, especially in dynamic or interactive scenarios, by providing clear boundaries and warnings.

2.3.4 Interaction Methods

VR technology enable users to interact with VEs through a variety of methods, including physical hand-held controllers, hand gestures, voice commands, and eye tracking. This diversity of interactions opens the door to a multitude of possible configurations, allowing for a tailor-made experience.

Controllers Interaction

Controllers are one of the most common interface between users and virtual worlds. These devices are designed to translate physical actions into digital responses, offering tactile feedback and precise motion tracking capabilities. For instance, the Oculus Touch Plus controllers shown in Figure 2.21, feature ergonomic grips, trigger buttons, and capacitive sensors. These inputs detect fingers' position and allow users to grasp, point, and interact with virtual objects in a natural and intuitive manner. Interaction through buttons may initially be less intuitive than interaction with hands alone, but after a short settling-in time, they can provide much more functionality than hands alone [80].

Figure 2.21: The Oculus Touch Plus controllers shipped with the Meta Quest 3.

Ergonomics is crucial to prevent user fatigue during extended sessions [80]. Buttons, triggers, and joysticks are strategically positioned to facilitate easy access and intuitive operation. The design

should ensure that interactions feel responsive and seamless, enhancing user engagement and immersion. Recent innovations include adaptive controllers that provide users with physical disabilities, enhancing inclusivity in VR experiences. Haptic feedback systems simulate tactile sensations, such as the sensation of touching a virtual object or the recoil of a virtual weapon, thereby deepening the sense of realism [65].

Hands-Tracking and Gestures

Hands-tracking technology allows interaction in VR by hands, enabling users to manipulate virtual objects with no need for physical controllers. Sensors and cameras detect hand movements and gestures, translating them into digital actions, as shown in Figure 2.22. For example, the Leap Motion sensor [9] enables users to sculpt virtual clay, paint digital canvases, or perform intricate surgery simulations with precise hand movements.

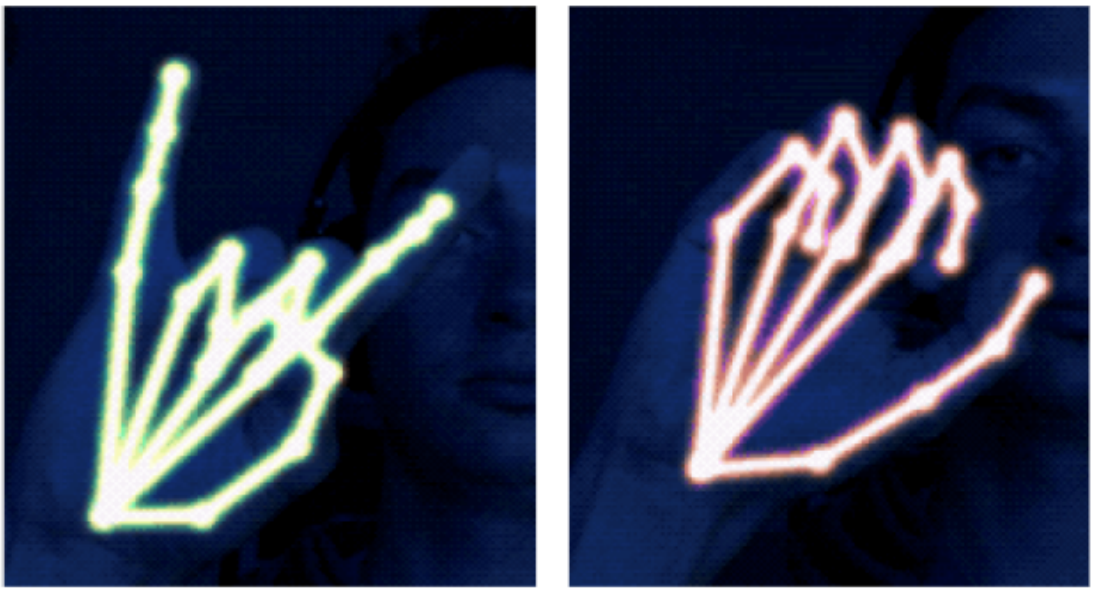

Figure 2.22: Real-time hands-tracking through the MediaPipe algorithm [11].

Hand tracking enhances immersion by enabling intuitive and natural interactions. Users can pick up, move, and rotate virtual objects using gestures that mirror real-world actions. In social VR applications, hand gestures facilitate non-verbal communication, such as waving, pointing, and thumbs-up gestures, fostering more lifelike interactions between avatars. Although this type of interaction may be more intuitive, it has been shown to be less efficient than interaction via controller in ray-cast optimization settings, while it is equivalent to controllers in close interaction settings [80].

Eye-Tracking

Eye tracking technology monitors the user's gaze within VR environments, allowing for dynamic interaction and enhanced visual experiences [81]. By detecting where the user is looking, eye tracking systems are used to adjust focus, depth of field, and object interaction based on gaze direction. For example, in a VR game, eye tracking can be used to aim a weapon simply by looking at a target, or to trigger environmental changes by focusing on specific objects.

The new Apple Vision Pro has made eye-tracking interaction its strong point, providing an environment that can select for interaction the visual element the user is looking at.

Eye tracking can enhance realism by further improving foveated rendering, namely where the highest visual detail is rendered in the user's direct line of sight, while peripheral areas are rendered

[9] See `https://www.ultraleap.com/`

at lower resolutions. This optimization significantly improves performance and reduces computational load, resulting in smoother VR experiences.

Eye tracking is also instrumental in usability research, analyzing user attention and engagement in VEs. For instance, eye-tracking has been used to track what the user focuses on, producing the heat map shown in Figure 2.23 [12].

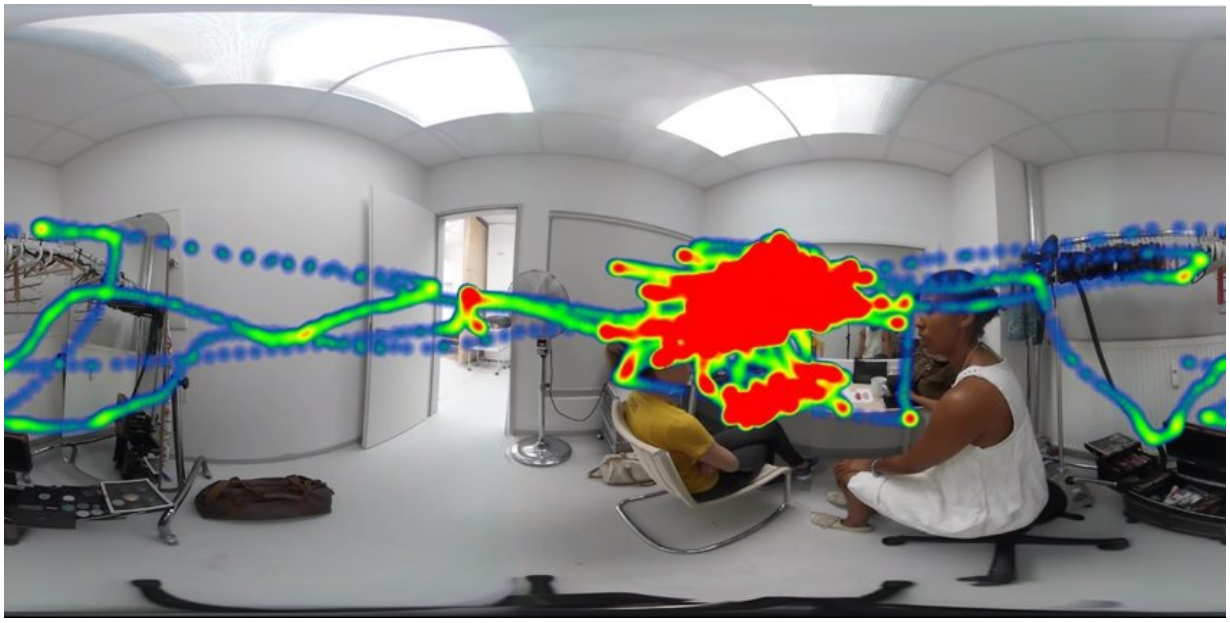

Figure 2.23: Heat map of the areas on which the user focuses attention [12].

It is also suitable for enhancing accessibility features by allowing users with disabilities to navigate interfaces using gaze-based interactions, thereby expanding the inclusivity of VR technology. Despite the benefits, using eye tracking to require the user to perform explicit actions in an unnatural way can lead to fatigue [82].

Voice Interaction

Voice interaction empowers users to control and interact with VEs using spoken commands. Leveraging advanced speech recognition algorithms, VR systems can interpret verbal instructions in real-time, executing corresponding actions within the digital space. Voice interaction offers hands-free operation, freeing users from the constraints of physical controllers and further enhancing multitasking capabilities within VR environments.

Natural language processing capabilities enable complex interactions, such as dictating text or interacting with AI-driven virtual assistants. Last but not least, voice recognition overcomes the problem of typing in a VE, which is particularly uncomfortable [83].

2.4 Virtual Reality Technologies

As we described in Section 2.2, the base concept behind the latest HMDs is still the same as that behind the stereoscope invented in 1832. It laid the foundation for the concept of immersive visual experiences by tricking the brain into perceiving depth and dimensionality. Moreover, the first HMDs with head movement tracking were already being experimented with in 1968. Despite its rudimentary graphics and cumbersome size, the Sword of Damocles introduced key concepts such as motion tracking, real-time rendering and interactive visuals.

VR technology creates a simulated environment that users can interact with in a seemingly real or physical way. The primary components of VR systems include:

- **HMD**: Devices that provide stereoscopic 3D visuals and are often equipped with motion tracking to adjust the VE as the user moves their head;

- **Input devices**: Ranging from hand-held controllers to sophisticated haptic gloves that provide tactile feedback, they enhance the sense of presence within the VE;
- **Motion tracking sensors**: Sensors are used to track the user's movements and translate them into the VE. They can be built into the HMD or be external stand-alone devices like cameras and infrared sensors;
- **Software**: VR software is crucial for creating immersive experiences. It includes the 3D models, animations, graphics and physics engines, and interaction logic that make up the virtual world.

Achieving high levels of immersion and realism is a primary goal in VR research, crucial for creating convincing VEs. Advances in graphics rendering techniques, such as ray tracing, have significantly improved VR environments. NVIDIA's RTX technology allows for real-time ray tracing, producing highly realistic lighting and shadows. A recent study evaluated the impact of photorealistic graphics on user immersion, finding that increased visual fidelity enhances the sense of presence and overall experience [84]. Research into environmental interactions focuses on how users engage with the virtual world. Locomotion into VE is a tricky part and can influence the whole experience, from the well-known VR-sickness to disorientation [85]. Accessories like the *Virtuix Omni* treadmill allow users to walk and run in VR, providing a full-body immersive experience.

In this section we explore hardware and software VR technologies currently available, as well as the technical specification that we use to assess which VR technologies suit our needs.

2.4.1 VR Headsets

A VR headset is a head-mounted device that provides immersive virtual experiences by simulating a three-dimensional environment. It consists of a pair of displays (or sometimes a single display) and a pair of small optics between the eyes and the displays, as shown in Figure 2.24. Depth is simulated by presenting two slightly different images to each eye through the displays and the lenses. The headset's sensors track the user's head movements, allowing the VE to respond in real-time, maintaining the illusion of being in a different place. The combination of visual, auditory, and interactive elements works together to create a sense of presence, making the user feel like they are actually inside the VE.

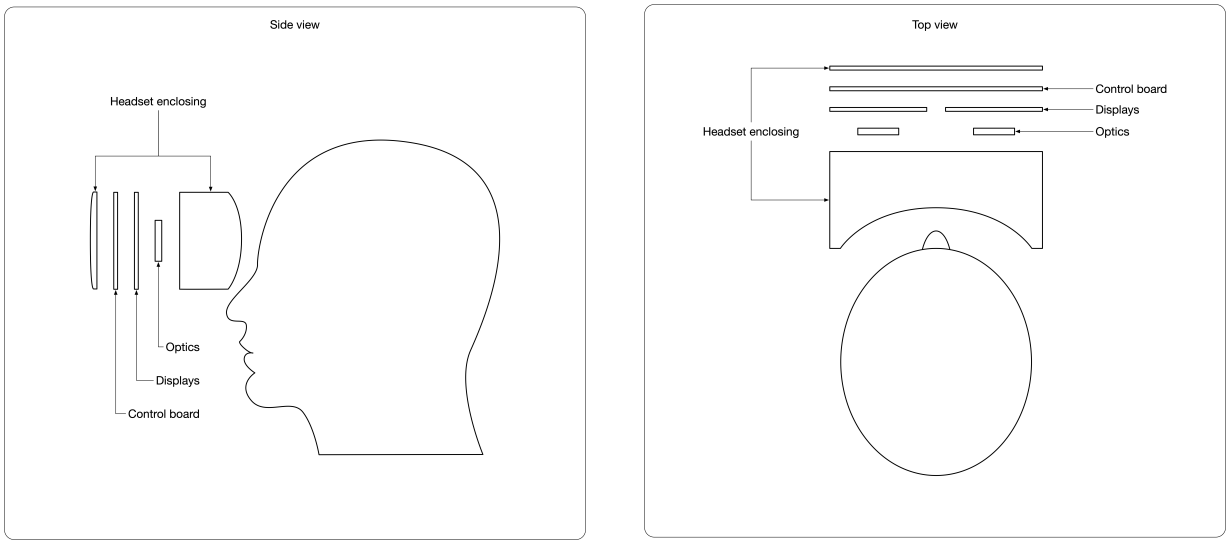

Figure 2.24: Hardware components of an head-mounted display.

The following is a list of the key components of a VR headset, together the their main technical characteristics that we use to assess the available technology:

- **Display(s)**: The display is the most crucial part of a VR headset, showing the VE. It can be a single screen or separate screens (one for each eye), that display slightly different images, creating a stereoscopic effect to simulate depth. Technical specifications that characterize the

displays are the resolution (*i.e.,* the vertical and horizontal size in pixel) and the refresh rate (*i.e.,* the times that the displays change the showed image each second);

- **Lenses**: Lenses sit between the display(s) and the user's eyes. They focus and reshape the image for each eye, creating a 3D effect by presenting slightly different perspectives to each eye. They are one of the most sensitive components and greatly influence the quality of the image perceived by the user. There are different types of lenses (*e.g.,* fresnel, aspherical and pancake) which influence image quality;
- **CPU and GPU**: The rendering performances and responsiveness of the device depend on the computational power of the mainboard components. High-end headsets use more powerful hardware to guarantee high performances;
- **Sensors**: Gyroscopes, accelerometers and cameras track the position and orientation of the user's head, allowing the VE to adjust accordingly. Sensor can be built into the headset (*i.e.,* inside-out tracking) or external stand-alone sensors placed around the room (*i.e.,* outside-in);
- **Audio**: Built-in headphones or audio jacks provide stereo or spatial audio, which enhances the sense of immersion by simulating sound from different directions;
- **Input methods**: Besides controllers, VR headsets can include other input methods like voice commands, eye tracking, and hands gesture recognition through built-in cameras;
- **Connectivity**: VR headsets can be standalone (*i.e.,* untethered), connecting wirelessly to the internet and other devices, or tethered, requiring a wired connection to a more powerful PC or gaming console.

2.4.2 Optics for VR applications

Optics are a critical component in VR headsets because they directly influence the visual experience and thus the immersion that users fee. To make virtual worlds feel real, the optics must present a clear and focused view that mimics how our eyes perceive depth and spatial relationships. They also influence the FoV: A wider FoV helps users feel surrounded by the virtual world, enhancing immersion. Without the right optical design, users would see a narrow or distorted view.

Moreover, optical lenses can introduce distortions (like shape distortion or chromatic aberrations) that make the VE look unnatural. High-quality optics reduce distortions, ensuring that what the user sees is lifelike. Poor optics can lead to eye strain or discomfort as the eyes struggle to focus on virtual objects. The following are the aspects of VR Influenced by optics:

- **Image Quality**: The sharpness and clarity of the image are directly related to the optics. Lenses in VR headsets magnify the screen for the user, and poor-quality lenses can result in blurry images or reduced detail;
- **Comfort**: Good optics help reduce eye strain and discomfort, especially during extended use. If lenses don't focus light properly or cause misalignment, users can quickly feel fatigued or even develop headaches;
- **Depth Perception**: Optics in VR contribute to how well depth is perceived, a critical factor for making the environment feel real. The interplay of lenses and the distance between the eyes help create convincing stereoscopic 3D effects;
- **Peripheral Vision and FoV**: Optics can either enhance or limit the user's peripheral vision, which is key to full immersion. Narrow FoV breaks immersion, while wide FoV improves the sense of presence;
- **Lens Aberrations and Artifacts**: Aberrations, such as chromatic aberration (where colors fringe at the edges of objects), can affect the user's perception. Proper lens design minimizes such artifacts.

The function of the optics in a VR headset is to bend and focus light in specific ways to ensure that images from a flat display appear as if they are coming from different distances and directions in 3D space. The lenses magnify the display to fill the user's FoV, ensuring they don't see the edges of the display and that the image appears as if it's surrounding them.

The most common optics for VR applications include fresnel lenses, aspheric lenses and pancake lenses, each one providing different quality and levels of comfort.

Fresnel Lenses

Fresnel lenses are made up of a series of concentric grooves, each acting as a small prism that bends light toward the user's eyes, as shown in Figure 2.25. These lenses are thinner and lighter than traditional lenses but can still focus light effectively.

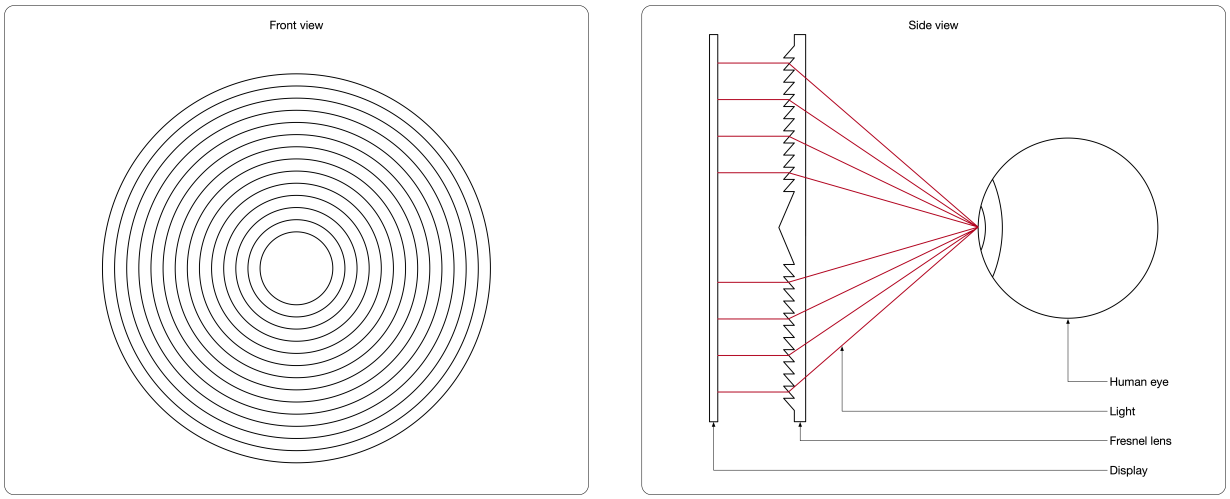

Figure 2.25: How the light passes through a fresnel lens.

Fresnel lenses are commonly used in VR headsets because they are lightweight, reducing the overall weight of the headset, and they help to maintain a wide FoV without requiring a bulky design. However, fresnel lenses can introduce visual artifacts like "god rays" (light streaks across the screen) and slight blurriness at the edges.

Aspheric Lenses

Aspheric lenses have a more complex shape than traditional spherical lenses, as shown in Figure 2.26, allowing them to focus light more accurately onto a flat surface. This reduces optical aberrations, especially at the periphery of the lenses.

These lenses offer clearer image quality with less distortion, particularly at the edges of the lens. This makes them ideal for VR, where a large portion of the user's view comes from peripheral vision. However, aspheric lenses are more expensive to manufacture than simple spherical lenses.

Pancake Lenses

Pancake lenses use a combination of polarization and light folding to reduce the optical path length. They rely on the principle of light reflection and refraction to fold the light path multiple times within a very compact space, as shown in Figure 2.27.

Pancake lenses allow for much thinner headsets by dramatically shortening the distance between the display and the user's eyes. This design improves form factor without sacrificing optical performance. However, they can reduce brightness due to light losses during the folding process, and they are generally more expensive and complex to manufacture.

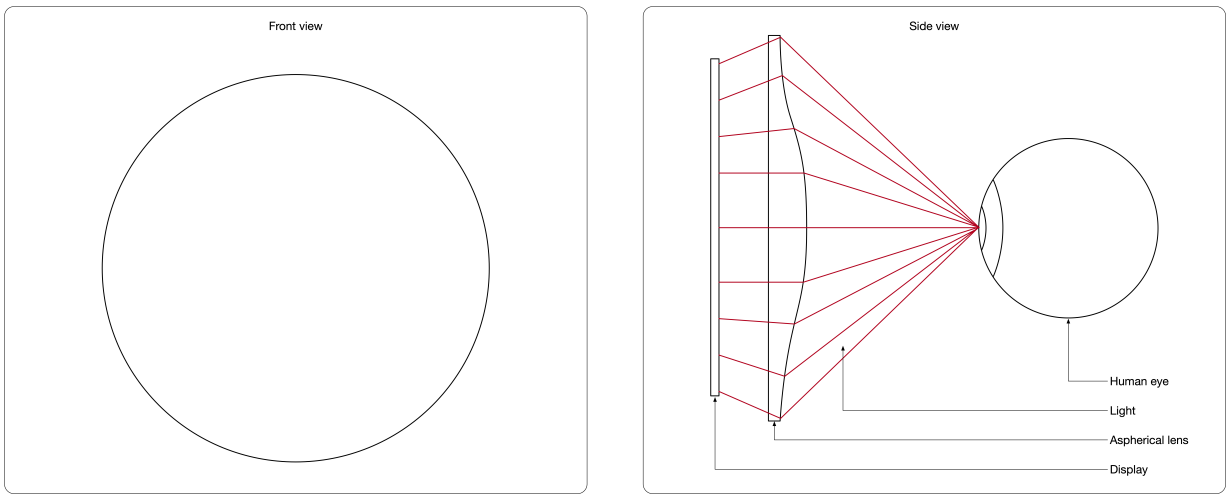

Figure 2.26: How the light passes through an aspheric lens.

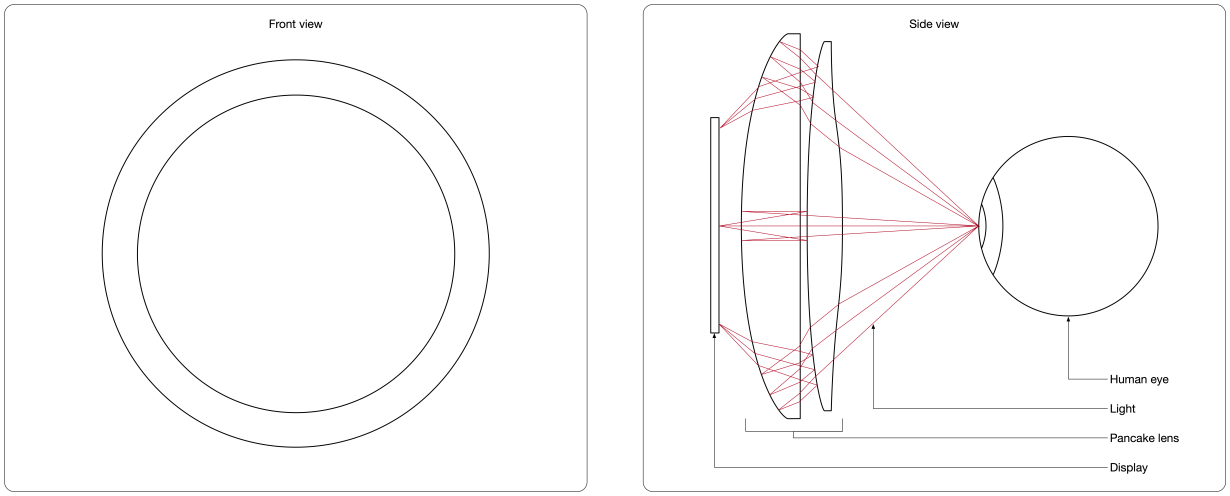

Figure 2.27: How the light passes through the two lenses composing a pancake lens.

Optics Comparison

In order to assess the different kind of optics, we rely on Table 2.1 [10] that summarizes the advantages and disadvantages of each type of optics.

	Fresnel Lenses	Aspherical Lenses	Pancake Lenses
Cost	Low	Medium-High	High
Weight	Light	Medium	Light
Image Quality	Moderate	High	Very High
Distortion	Potential Issues	Minimal	Minimal
Field of View	Wide	Moderate-Wide	Moderate-Wide
Eye Fatigue	Moderate	Moderate	Minimal

Table 2.1: Summary of the advantages and disadvantages of fresnel, aspheric and pancake lenses.

2.4.3 VR Controllers

Interactions in VR can be achieved through either bare hands or hand-held devices, each offering a different experience. A VR controller is a hand-held device used to interact with VEs within a

[10] See `https://vskel.com/3-types-of-vr-headset-lenses/`

VR headset. They are equipped with many sensors, buttons, and input methods that allow users to perform actions and manipulate objects into the virtual world. The physical buttons and triggers give users a strong sense of control, while haptic feedback enhances immersion. On the other hand, hand-tracking offers a more intuitive and natural interaction, allowing users to engage with virtual objects using gestures. This approach feels more seamless and immersive, as it closely mimics real-world movements. However, it often lacks the precision and feedback that controllers provide, which can be a limitation for more complex or precise tasks [80].

It was also shown that for direct interaction, there is not much difference between interaction through controllers and through hands. In contrast, for ray-cast interaction, controllers perform much better than hands in terms of precision, physical exertion and preference [80].

Key components of a controller are:

- **Motion sensors**: Accelerometers measure the acceleration forces to detect the controller's movement and orientation, gyroscopes track the rotation and angular velocity of the controller, magnetometers help correct orientation and track the controller's position relative to the Earth's magnetic field;
- **Buttons and triggers**: On the controller there are several input buttons for selecting options, performing actions, and interacting with the VE. Triggers are usually located under the index fingers, used for actions like shooting or grabbing objects;
- **Thumb-sticks and touchpads**: Thumb-sticks are placed to be used with the thumb and allow for directional input, commonly used for navigation within the VR environment. Touchpads can detect touch and swipe gestures, providing an alternative method for interaction and navigation;
- **Capacitive touch sensors**: Detecting the position of the user's fingers on the controller allows for more precise interactions, such as finger tracking, also providing a visual feedback about the position of the fingers on the controllers;
- **Haptic feedback**: Providing a tactile feedback (*i.e.,* vibrations) simulates the sense of touch and enhance the immersive experience by conveying different sensations based on virtual interactions.

2.4.4 Motion-Tracking

Motion tracking is the technology that allows VR systems to detect and interpret the movements of a user in the real world and translate them into the VE. There are two primary approaches to motion tracking in VR:

- **Inside-Out Tracking**: In this system, cameras or sensors are mounted on the VR headset itself. The headset "sees" the surrounding environment and tracks the user's position within the space. Inside-out tracking is more convenient and portable since no external hardware (*e.g.,* sensors, base stations) is needed;
- **Outside-In Tracking**: This method uses external sensors or cameras positioned around the room to track the position and movement of the VR headset and controllers. Outside-in tracking can provide more precise tracking, especially for large-scale movements and room-scale experiences.

While talking about the *movement* of the user into the physical space, we refer to translations and rotations. Since we live in a world with three spatial dimensions, we can move and rotate along three different axes. Each axis is called *degree of freedom* (DoF) [11].

> *Degrees of freedom refer to the number of ways in which a user can move or rotate within the VE.*

[11] See `https://varjo.com/learning-hub/degrees-of-freedom-in-vr-xr/`

The number of DoF influences how freely and realistically a user can interact with the VE. VR systems usually track three or six DoF, as shown in Figure 2.28.

Three DoF refers to the ability of the VR system to detect the rotation in three different directions, but without any positional tracking (*i.e.,* movement through space). The three degrees of rotational freedom are:

- **Pitch**: Tilting your head up and down (like nodding "yes");
- **Yaw**: Turning your head left and right (like shaking your head "no");
- **Roll**: Tilting your head side-to-side (like tipping your head toward your shoulder).

Six DoF refers to both rotational and positional tracking, allowing users to move freely through space as well as rotate their heads. The three additional degrees of freedom are:

- **X-axis**: Moving left and right (side-to-side movement, known as strafing);
- **Y-axis**: Moving up and down (vertical movement, like jumping);
- **Z-axis**: Moving forward and backward (moving closer to or away from objects).

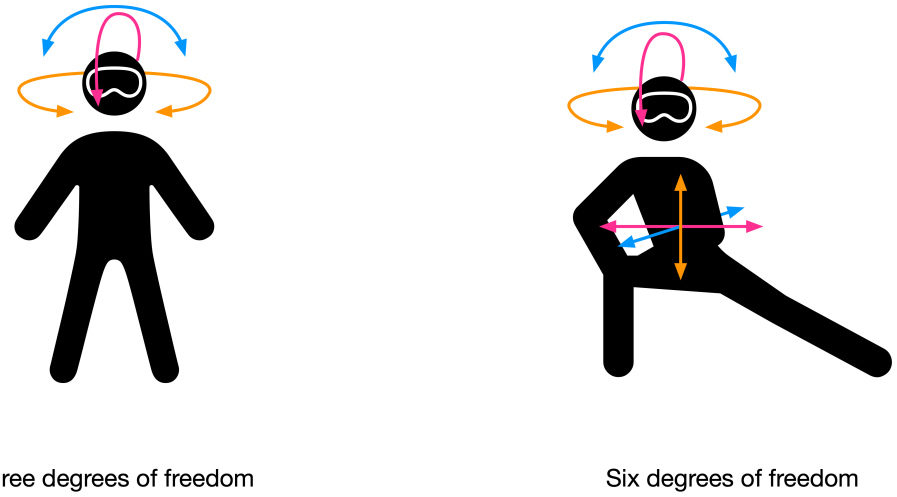

Figure 2.28: Three degrees-of-freedom on the left, six degrees-of-freedom on the right.

Six DoF enhances immersion significantly because users can move freely within the virtual world, which feels more natural compared to being restricted to just head movement. It makes activities like exploring or interacting with objects feel realistic. With accurate motion tracking, the VR system can adjust the VE according to the user's movements, reducing motion sickness.

Motion tracking and degrees of freedom are central to how immersive and interactive a VR experience is. With higher DoF and more sophisticated motion tracking, users can engage more deeply with the virtual world, performing actions and movements as they would in real life. Systems with six DoF and robust motion tracking offer a superior, more natural VR experience compared to those with limited tracking capabilities.

2.4.5 Other VR Accessories

VR accessories can enhance the immersive experience by expanding the range of interactions and sensations users can experience within VEs.

The HTC Vive Tracker is an innovative accessory that broadens the scope of what can be tracked and used as input devices in VR. By attaching these trackers to various objects, such as sports equipment, props, or full-body tracking suits, users can bring a wide array of physical items into the virtual space. This capability allows for more immersive and interactive VR experiences, as the tracker

seamlessly integrates with the HTC Vive and Vive Pro systems. For example, attaching a tracker to a baseball bat allows the user to swing it in a VR baseball game, adding a layer of realism and physical engagement that traditional controllers cannot offer.

VR treadmills, such as the Virtuix Omni and KAT Walk, take physical immersion a step further by allowing users to walk and run within VR environments. These devices support 360-degree movement, enabling users to explore virtual worlds in a more natural and physically engaging way. By simulating real-world locomotion, VR treadmills enhance the sense of presence and physicality in VR experiences. This is particularly beneficial for gaming, where players can move through expansive virtual landscapes, and for training simulations, where realistic movement can improve the effectiveness of the training.

Haptic gloves, such as those developed by HaptX, add another dimension to VR interaction by providing tactile feedback to the user's hands. These gloves use a combination of force feedback and microfluidic actuators to create realistic touch sensations, allowing users to feel the texture, shape, and resistance of virtual objects. This capability is invaluable for applications requiring fine motor skills and precise interactions. For instance, in surgical training, haptic gloves can simulate the feel of surgical instruments and tissues, helping trainees develop the necessary tactile skills in a controlled VE. Similarly, in virtual prototyping, designers can manipulate and assess virtual models with their hands, gaining insights into how products will feel and function in the real world.

2.5 Developing for VR

VR applications are a blend of 3D rendering, real-time interaction, and VE perception. Unlike traditional desktop or mobile development, VR introduces unique challenges such as managing user inputs that go beyond keyboard and mouse interactions. The goal of VR development is to create a convincing, interactive experience where users feel fully immersed in the virtual world. Both immersion and interaction are critical and directly affect usability.

VR technology must respond to user movement in real-time, whether it's turning their head to look around or physically moving within the virtual space. High-performance rendering is hence necessary, as VR experiences need to run at high frame rates (usually 90 to 120 FPS) to avoid motion sickness, which can occur if there are performance dips or frame drops [86]. Motion sickness, or *VR sickness*, is a common issue caused by a mismatch between the visual input and the user's physical movement [87]. Techniques such as teleportation for movement, gradual acceleration, and reduced rotational speed can help mitigate this issue. Developers need to be mindful of how users move through virtual spaces and how their movements align with real-world physics to avoid disorienting them [88].

Input methods are another delicate aspect of VR development. VR headsets are typically paired with hand controllers, though some devices now support hand-tracking. Developers must account for diverse input forms, from gestures to gaze-based selection, ensuring that users can interact naturally with their surroundings. Designing intuitive interactions for the virtual space requires a fundamental shift from how traditional GUIs are designed, as VR experiences need to feel both intuitive and immersive [89].

The type of VR experience a developer aims to create also plays a significant role in shaping the design approach. For example, creating a 360-degree video is quite different from building an interactive VR tool. In 360-degree video, the user can look around in all directions, but there is no interaction. In contrast, interactive VR allows users to engage with objects, move around, and interact with the VE in real-time, which introduces a whole new layer of complexity.

2.5.1 Software Technology for VR

VR software has significantly advanced, offering immersive and interactive experiences across various fields. This technology includes sophisticated physics and graphic engines, rendering techniques, GUIs, and efficient content creation and management systems, all of which work together to create engaging VEs. Central to VR technology are physics and rendering engines, which ensure high frame rates and low latency to maintain immersion [72]. Modern engines employ techniques such as foveated rendering, which prioritizes detail in the user's focal area while reducing detail in the periphery, thus

optimizing performance without sacrificing visual quality. Real-time ray tracing has further improved VR graphics, providing realistic lighting, shadows, and reflections that enhance the sense of realism in virtual worlds [72].

GUIs and interaction design are crucial for intuitive VR experiences. Unlike traditional 2D GUIs, VR UIs must operate within a three-dimensional space [90]. This leads to the development of spatial menus, gesture-based controls, and voice commands [89, 91]. Gesture recognition, powered by advanced sensors and machine learning, enables users to perform actions with hand movements, making interactions in applications like virtual training more intuitive [90]. Voice commands add another layer of interaction, allowing users to control VR environments through speech.

Multi-user VR environments present additional software challenges, including networking and latency management. Platforms like *VRChat* and *AltspaceVR* enable users to interact in shared virtual spaces, requiring robust networking solutions to ensure real-time interactions. Predictive tracking and network optimization algorithms help mitigate latency, providing smooth and synchronized experiences.

Even if they are not closely related, content creation and management are also very important in VR. Tools like Blender and Autodesk Maya are used for 3D modeling and animation, enabling the creation of detailed virtual objects and characters. These models are then integrated into VR environments, where they are animated and brought to life. Sound design also plays a critical role, with spatial audio techniques simulating how sound behaves in the real world, enhancing immersion by providing directional audio cues. Content management systems (CMS) for VR help organize, store, and retrieve assets, which is particularly important even in small-scale projects. These systems offer version control, collaborative tools, and cloud storage, ensuring that development teams have access to the latest assets and project files, helping in having an efficient workflow and collaboration.

2.5.2 SDKs, Game Engines and Profiling Tools

Different VR platforms provide different tools and devices to bring these experiences to life. Each platform has its own SDKs and hardware capabilities, requiring to adapt the applications for different devices. For instance, Oculus devices offer both standalone and PC-tethered modes, allowing for a range of performance options depending on the target audience.

VR SDKs and libraries serve as the starting point of the development process, offering essential tools and APIs that allow developers to interface with VR hardware. OpenXR has emerged as a standard for VR and AR development, enabling cross-platform compatibility. Instead of developing specific versions of an application for different VR platforms, OpenXR allows developers to write code that works across multiple devices, saving time and resources. However, these frameworks are often limited and fail to fully exploit the capabilities of the hardware devices. On the other hand, platform-specific SDKs such as Oculus SDK, which offers deep integration with Meta's VR devices, come with pre-built components and interaction systems that make it easier for developers to manage device-specific features such as hand-tracking and spatial audio.

Among the most commonly used game engines for VR development are Unity and Unreal Engine. Unity, known for its ease of use and extensive community support, is ideal for developers who want a flexible and robust engine for VR projects. It supports numerous VR SDKs, including Oculus SDK, SteamVR, and OpenXR. Unity's XR Interaction Toolkit simplifies handling common VR interactions, such as object manipulation and teleportation. In contrast, Unreal Engine is favored for its graphical fidelity and powerful rendering capabilities, making it a preferred choice for AAA game and highly realistic VR simulations.

Being performance a critical factor in VR development, there is also the need of analyzing frame rate and hardware utilization. Tools like the Oculus Performance head-up display, Meta Quest Developers Hub's performance analyzer and Unity's profiler help developers optimize the application for smooth performance.

2.6 Reflections

We presented an overview of the history of software visualization, from the early on-paper representations to modern 3D collaborative tools. This highlights how developers have always needed visual aids to better comprehend the abstract aspects of software. 3D visualizations allow the developers to build information-rich views, using visual properties such as size, shape and color to convey domain properties.

VR also has a long history and has been used extensively in many research fields, especially simulation-related ones, precisely because of its interaction capabilities. The key concepts behind VR are immersion and interaction:

- Immersion depends directly on the VR system's ability to present a VE as close to reality as possible, both in terms of visual quality and rendering speed. Poor quality and bad performance can lead to a sense of discomfort that breaks the sense of presence. Given the importance of quality and performance aspects, we reviewed the main hardware properties of VR system that mainly influence these aspects, in order to understand how to assess a VR system;
- Interaction lays its foundations in human-computer interaction and cognitive science. These two areas study how an interaction is carried out in an intentional and natural way, as well as how it is perceived by the user. Guidelines such as affordance, feedback and ergonomics guide the implementation of the interactions to make them result as intuitive as possible. The key is to develop virtual interactions that mimics real-world interactions, in order to leverage the knowledge that a user already has about the world they live in.

Interaction in a VE usually occurs with two categories of virtual objects: 3D objects and GUIs. Since 3D objects simulate real-world objects, interactions with such objects is inherently intuitive. Picking and throwing an object, for example, is an intuitive interaction based on real-world physics. On the other hand, interacting with a GUI relies on the knowledge that the user has about other on-screen UIs, thus with 2D objects. These objects fall short of the interaction capabilities that VR provides, especially in terms of degrees of freedom. In addition, the majority of visualization tools in VR, only allow customization through on-screen 2D UIs, forcing the user to perform a context-switch every time they have to go back and forth between the configuration GUI and the visualization.

In the next chapter, we show how we model a visualization tool for VR by leveraging the theoretical background of this chapter. As notable results of the modeling, we propose a novel approach for designing GUIs specifically for VR and an extension of the well-known city layout.

Chapter 3

Visualizing and Interacting with a Domain in VR

This chapter shows how we model a tool that allows a user to visualize a reference domain (*e.g.,* a software system) in VR and interact with its components.

In designing our approach, we first focus on developing the metaphors that shape how the virtual objects convey their function, how the users interact with them and how the users perceive the VE. The primary goal is to create interactions that mimic real-world actions, allowing users to engage with software elements in a way that feels familiar. To achieve this, we drew inspiration from principles of human-computer interaction and cognitive science, ensuring that the virtual interactions are both intuitive and intuitive.

In parallel, we focused on minimizing reliance on traditional 2D UIs, which are commonly used in many existing VR tools. These flat, screen-bound interfaces often disrupt the immersive experience by forcing users to shift between interacting with the 3D VE and navigating 2D menus. To address this, we designed interaction techniques that allow the user to configure the visualizations directly within the VE, eliminating the need for context-switching and maintaining the user's immersion.

While interaction is the central focus, the visualization of the reference domain also plays a crucial role in this approach. By visualizing domain entities, we provide users with information-rich, intuitive views that help in navigating the architecture of the domain. Visual properties of a virtual object (*e.g.,* size, shape,) are mapped to attributes of the domain entities that the virtual object represents. This enables users to grasp relevant information about the system at a glance. Through this integration of intuitive interaction and enhanced visualization, our approach offers a powerful tool for exploring and manipulating a given reference domain.

The domain of software systems is particularly interesting, and there are already many software visualization techniques that exploit 3D representations. In this domain, the attributes of the domain entities we consider are expressed by software metrics. A software metric is a quantitative measure that evaluates different aspects of a software component (*e.g.,* quality, complexity, maintainability). Thus, software metrics provide objective data, allowing developers to make informed decisions, track progress, identify problems, and assess performance. To make a few concrete examples of metrics that we consider, we can map onto visual properties the number of lines in a text file, the size of the file in Megabytes, and the number of references to a Java methods.

This chapter is structured as follows:

- In Section 3.1, we present the metaphors that we designed to shape how the user perceives the VE and how they interact with the virtual objects. We then highlight the importance of the city layout and extend it to a general model. Finally, we present a novel approach for designing VR-native GUIs based on the interaction with 3D objects;
- In Section 3.2, we first present the theoretical concepts of cognitive science behind visualization. Then, we explain how they can be applied to software visualization. Finally, we present an example of software visualization in which we apply such concepts;

- In Section 3.3, we show how we use software metrics as the properties of the domain entities to visualize a software system;
- In Section 3.4, we first show how we model the representation of a generic reference domain and the visual domain. Then we show how we map domain entities to visual representations;

3.1 Metaphors for Interaction

According to the Cambridge Dictionary of English [1]:

> *A metaphor is an expression that describes a person or object by referring to something that is considered to have similar characteristics to that person or object.*

Thus, a metaphor serves as a bridge between the familiar and the unfamiliar, allowing users to intuitively understand new or abstract concepts by relating them to known objects or experiences [92]. By leveraging familiar metaphors, users can more easily grasp complex concepts and actions without extensive explanation, thus improving usability [92].

In this section we present five metaphors that we implemented to shape different aspects of the interactions.

3.1.1 City Layout

The city layout is a well-known metaphor in software visualization [6]. It arranges the entities of a domain in the VE in a way that resembles a city, as shown in Figure 3.1. Different clusters of related components, such as those belonging to the same package, are represented as districts, while individual components are akin to buildings within these districts. This organization intuitively conveys the hierarchical relationships among components through the superposition of nested elements.

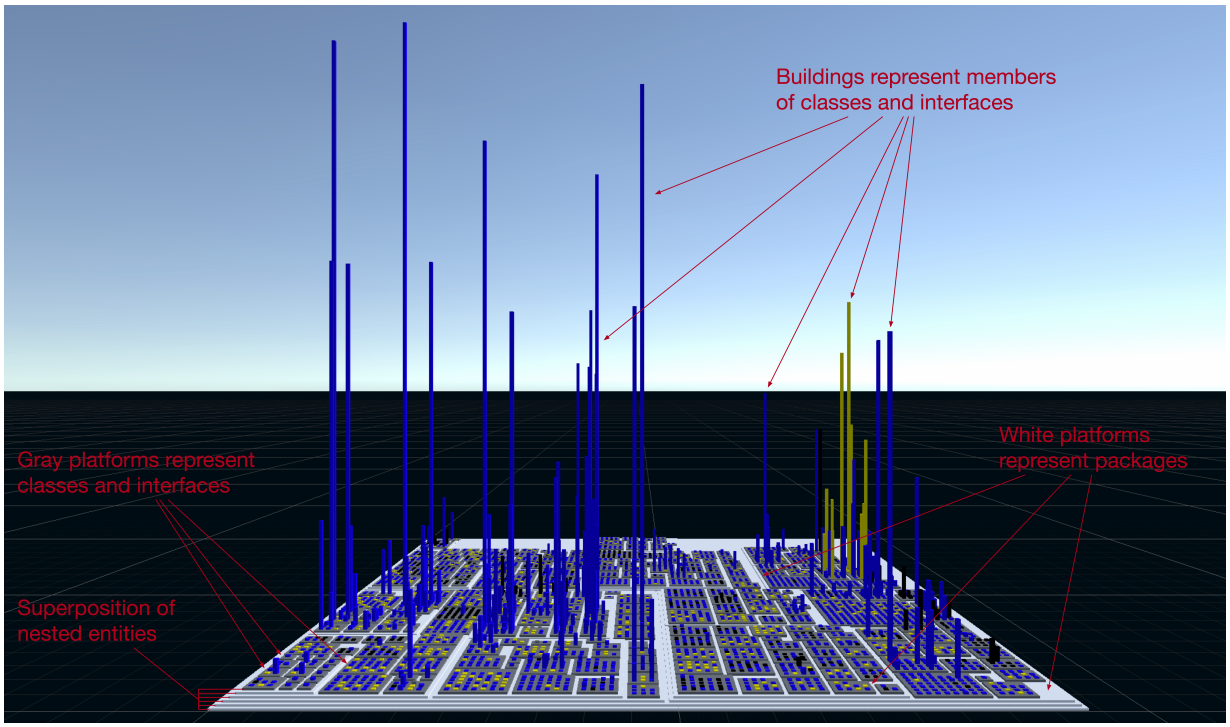

Figure 3.1: A software system visualized through the city metaphor.

Although this metaphor is usually applied to software systems, we have extended it to a generic domain. Hence, buildings and districts can represent any entity in the domain and the hierarchy can be any given relation.

[1] See `https://dictionary.cambridge.org/dictionary/english/metaphor`

To give an example, this metaphor can also be used to visualize a file-system. In order to distinguish between binary files, text files and java files, We build the following mapping:

- A binary file is mapped onto a black hope of unit radius;
- A text file is mapped onto a red 1-by-1 base parallelepiped. Its height is proportional to its number of lines;
- A java file is mapped onto a green 1-by-1 base parallelepiped. Its height is also proportional to its number of lines.

An example of the mapping is shown in Figure 3.2. We then build two city layouts based on different relations: The one given by the containment and the one given by the file type, as shown in Figure 3.3. The type is obtained by mapping the file extensions to a taxonomy of the file types.

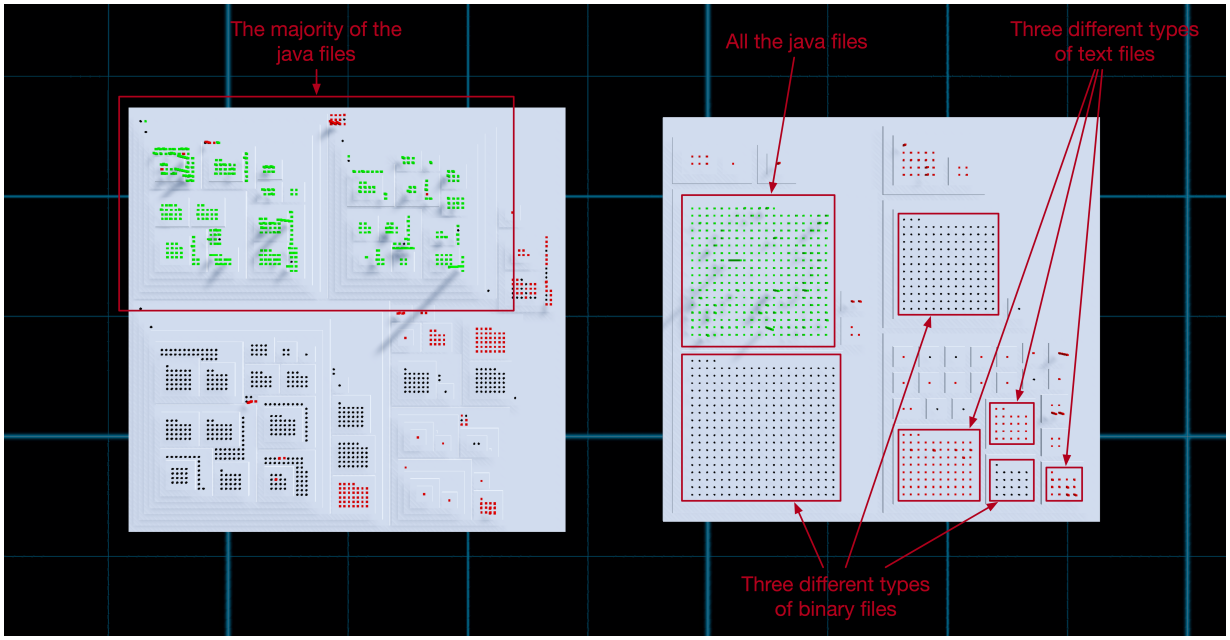

Figure 3.2: The different visualizations of binary, text, and java files.

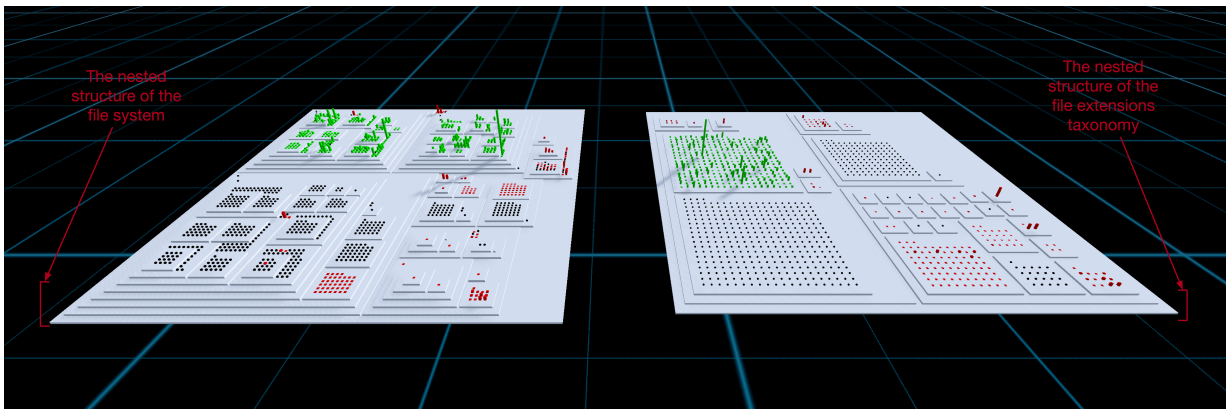

Figure 3.3: The two hierarchies given by the containment and the file type relations.

The city metaphor offers users a familiar and intuitive spatial layout for navigating large and complex domains. Just as people orient themselves in real-world cities by recognizing landmarks, the same principle applies in VR, where different areas of the city correspond to various domains or categories within the system.

In the context of software development, one district may represent code libraries, while another might represent testing suites. Users can travel between these districts to easily locate and interact with specific glyphs, such as debugging tools or modules.

3.1.2 3D Selection Menu

We designed this metaphor with the purpose of migrating from traditional 2D interfaces designed for a display to 3D interfaces designed specifically for VR interaction. In particular, the 3D selection menu aims to replace the traditional drop-down menus of 2D UIs. It operates like a physical selection mechanism where users "pick" an item represented by a 3D object and positions it on a platform above the others, as shown in Figure 3.4.

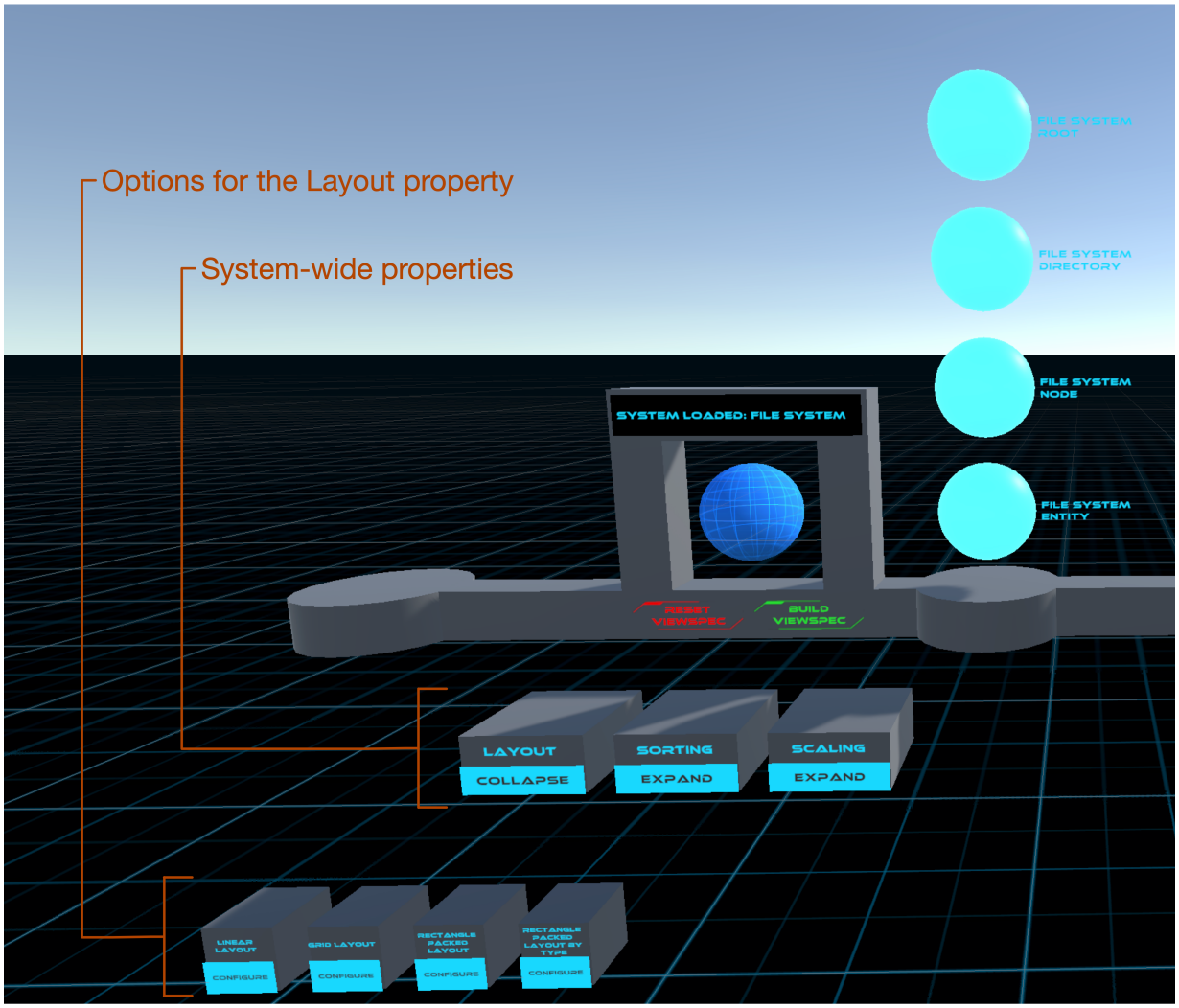

Figure 3.4: The cubes at the bottom are the available options for the layout. Selecting a specific layout is done by picking the glyph depicting it and placing it on the "layout" platform.

By translating the process of selection into a spatial 3D interaction, the user can engage with the system in a more immersive and embodied manner. Studies in embodied cognition [93] suggest that interacting with virtual objects in ways that mimic physical actions enhances user engagement and comprehension. In VR, physical gestures such as pointing or grabbing 3D objects have been found to improve user interaction efficiency [93], also creating uniformity between the system objects they interact with and the GUI.

In addition, the 3D selection menu present two additional benefits:

- The virtual objects representing the available options are not limited to text. Consider the case of a 3D selection menu used to allow the user choose between options that have a concrete representation (*e.g.,* colors, shapes). Being in a 3D environment, the available options can be visualized in 3D instead of using text, as shown in Figure 3.5;
- The available options can be combined together to build new options, acting as building blocks for new options.

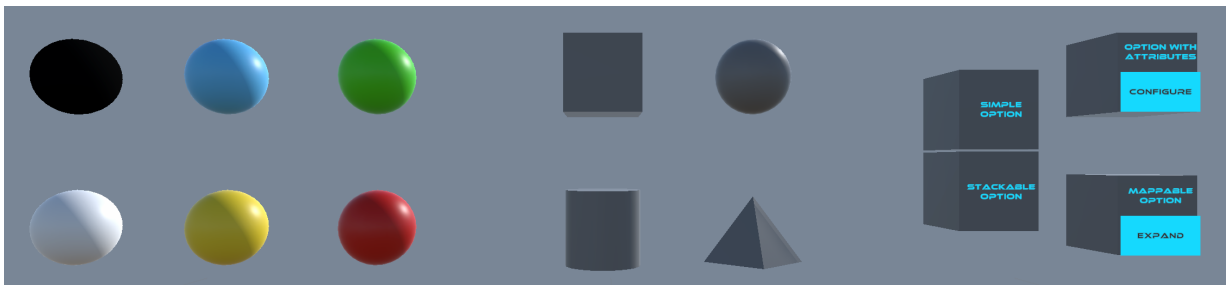

Figure 3.5: The objects that represent available options for the 3D selection menu.

This metaphor gives novel perspectives for building UIs designed specifically for VR, moving away from the traditional way of interacting with 2D windows. To show the potential of this new direction, we developed a proof-of-concept (*i.e.*,IVAR-NI) that applies the 3D selection metaphor to the manipulation of viewspecs.

This topic is of great interest to the scientific community and we got positive feedbacks from the VISSOFT scientific community on which we published the work [89].

3.1.3 Lines Connecting Related Items

We visualize relationships between object of the domain through lines between their visual depiction. Just as a cable connects two elements that can communicate with each other, the metaphor of physically connecting glyphs helps users understand relationships and dependencies between different system elements. This concept can be enriched, for example, with the direction, the size or the color of the line and even with the shape of the connector, making it clear which objects can be connected together and for what purpose.

Research in information visualization shows that graph-based representations (*e.g.,* node-link diagrams) enhance the user's ability to perceive and interpret complex relationships in a system [94]. These relationships become even more tangible when represented in a VR environment, allowing users to see dependencies from different perspectives and interact with the connections themselves.

In the java project case study we present in Chapter 5, for instance, the lines between components show packages referencing a selected object, as shown in Figure 3.6.

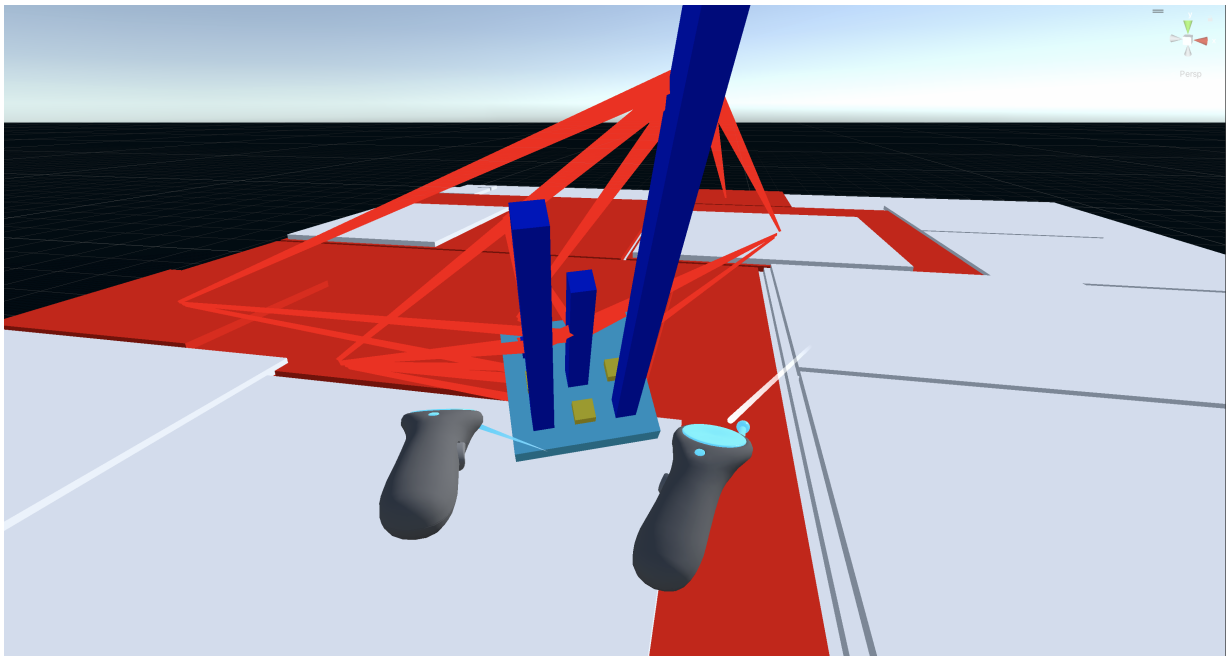

Figure 3.6: Visualizing the references from a package to a selected object.

3.1.4 Inspection Tool

The inspection tool allows retrieving details and insights about the objects of the domain. Each physical hand-held controller has a 3D model depicting it into the VE. Each 3D model is provided with a virtual ray casted from the controller that the user can use to interact with the virtual objects. When a user brings the ray to a glyph, a virtual window (*i.e.,* the tool-tip) is shown on top of the controller, containing some information about the object in the domain depicted by that glyph. An example of the tool-tip is shown in Figure 3.7 (A). By clicking a button on the controller, the user can show another larger virtual window (*i.e.,* the advanced tool-tip) with extended information, as shown in Figure 3.7 (B).

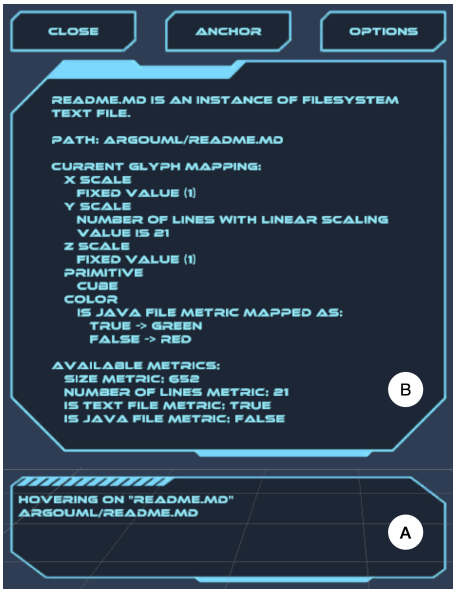

Figure 3.7: The inspection tool. The tool-tip (A) shows basics information while the advanced tool-tip (B) shows extended information on demand.

The metaphor of the inspection tool is inspired by the tools an electrician uses to read the values of a circuit. It acts as an information retrieval tool, offering in-depth insights on demand without cluttering the main view with excessive details. Cognitive load theory suggests that overloading users with too much information at once can disrupt learning and performance [95]. By using an inspection tool, users can choose when and what additional information they need, ensuring that only relevant data is presented when required.

3.1.5 Moving Items in Space

Interaction with the virtual objects mimics the interaction with physical objects in the real world. Users can pick them up and moving them within the VE. The direct manipulation of objects in a VE is a well-established concept in human-computer interaction [96], offering users a greater sense of agency and control. Studies have shown that VR applications benefit from allowing users to engage spatially with objects, as this interaction is highly intuitive and reflects real-world physics [97].

In the java project case study we present in Chapter 5, for instance, a class can be moved from one package to another simply by grabbing, moving and dropping it on the new package. This interaction is not only more natural than the same action performed via contextual menus in an IDE, but implemented in VR it also allows two-handed interactions: A user in a VE interacts with it using two controllers. Each controller functions independently of the other, and they can be used to perform different tasks at the same time. For instance, while a user is moving a class from one package to another, they can also use the other controller for inspecting and moving other components.

3.2 Cognitive Science for Software Visualization

Our brain processes visual information much faster than text [98], and images can often communicate meaning more effectively [99]. This is why visualization is critical in fields ranging from education to data analytics and software engineering. The effort needed to accomplish comprehension is defined as:

> *Cognitive load is the processing load imposed on the performer by a particular task, or the perceived mental effort the performer invests in a task [100].*

Visualization helps reduce cognitive load by turning abstract or quantitative data into graphical formats like charts, maps, or 3D representations, making the data easier to analyze and comprehend [101], hence reducing the time needed for the brain to process the information.

Software visualization is visualization applied to the domain of software engineering and development. The theory behind software visualization is grounded in cognitive science, which seeks to understand how humans process and comprehend information. Visualization, in this context, acts as a bridge between the mental model developers have about the software and the actual software system itself [102]. Mental models shape how people perceive the tasks or concepts they encounter, their capabilities and themselves:

> *A mental model is an internal representation that individuals form based on their interactions with the world, other individuals, and the objects they engage with [103].*

They help predict the outcome of an interaction and explain the behavior of a system. By influencing how people interpret information and make decisions, mental models play a crucial role in learning, problem-solving, and interaction with both physical and digital environments [103].

A larger mental model leads to a higher cognitive load, since the amount of information to be processed all at once is greater. Interactive visualizations allow users to manipulate the visualized data to explore different aspects of the system. Zooming in and out of large codebases allows changing the level of abstraction and filtering out irrelevant data. Changing the level of abstraction allows to simplify complex information without losing its essence. Different levels of abstraction allow developers to focus on specific areas without being overwhelmed by the small details of the entire system. Filtering allows to hide irrelevant data, avoiding visual noise.

Dynamically adjusting what aspects of the system are being monitored greatly enhances the developer's ability to explore and comprehend the software system. These techniques are especially important in the modern context of large-scale systems, which often have hundreds or thousands of interacting components. The ability to interact with visual representations allows users to identify specific problems more quickly than by screening static representations [95].

3.3 Software Metrics

We use software metrics for assessing the properties of the objects in the domain.

> *A software metric is a quantitative measure that helps assess various attributes of a software system, such as its quality, performance, complexity, or maintainability [104].*

Software metrics play a crucial role in the software development lifecycle, as well as in code comprehension [105]. By providing objective data, they allow developers to make informed decisions based on measurable characteristics, improving both the development process and the final product. Software metrics are important because they provide an empirical basis for evaluating software, which can otherwise be subjective.

For instance, a developer might feel that a particular module is "complex" or "efficient", but without a way to quantify it, that evaluation is open to interpretation. Metrics help put numbers

to subjective concepts, making it easier to track progress, detect potential problems, and benchmark performance.

Metrics are also valuable in tracking the quality of the software itself. For example, defect density measures the number of bugs or defects found per unit of code. A high defect density may suggest that the software is unstable or poorly designed. Tracking this over time helps ensure that as new features are added, quality remains under control. Similarly, testing metrics like test coverage provide insight into how much of the code is being tested. If a large portion of the code remains untested, the risk of undetected bugs increases.

However, calculating even a few metrics on a large software system can produce an enormous amount of data, making it very hard to use them or to get information from them. For this reason, combining metrics with software visualization can provide an insightful mix to extract the information needed to perform a given task, such as bug-fixing a piece of code.

The metrics we implemented in this work are:

- **Size**: The size of a file system entity (*i.e.,* a file or a directory) in Megabytes. It can be measured on `FileSystemEntity` instances;
- **Number of Children**: The number of children of a node in a tree hierarchy. This metric counts the outgoing `ContainmentEdge` of the given node. This metric can be measured on `FileSystemEntity`, `JavaPackage`, `JavaClass`, `JavaInterface`, `JavaEnum`, and `JavaMethod`, `JavaConstructor` instances;
- **Number of Lines**: The number of lines of a text file. It can be measured on `FileSystemTextFile` instances;
- **Number of References**: The number times a java symbol is referenced in a java project. It can be measured on `JavaSymbol` entities.

We also implemented categorical properties to assess categorical aspects of the domain entities. The categorical properties implemented in this work are:

- **File Type**: The type of the file according to a taxonomy that categorize the file extensions. It can be associated to `FileSystemFile` instances;
- **Is Text File**: Wether a file is a text file or not. It can be associated to `FileSystemFile` instances;
- **Is Java File**: Wether a text file is a java file or not. It can be associated to `FileSystemTextFile` instances;

3.4 Model

A software system can be viewed at different levels of abstraction. At the file system level, it appears as source code organized into files and folders. At a higher level, it can be seen as software components (*e.g.,* classes, fields, methods) structured within nested modules. Both of these examples follow a tree-like structure. However, when additional relationships (*e.g.,* dependencies between components, inheritance, utilization) are considered, the system's structure becomes more complex and no longer fits into a tree. A graph structure, which better represents these relations, is more challenging to explore and understand. To manage this complexity, visualizing the graph is essential.

We model the domain as a generic graph, where both nodes and edges are visualized through a graphical representation (*i.e.,* glyphs). The visual characteristics of the glyphs (*e.g.,* shape, color) are mapped to specific properties of the domain entities represented by nodes and edges. The domain properties we consider are expressed by software metrics. A software metric is a quantitative measure that evaluates different aspects of a software component (*e.g.,* quality, complexity, maintainability). Thus, software metrics provide objective data, allowing developers to make informed decisions, track progress, identify problems, and assess performance. To make a few concrete examples of metrics that

we consider, we can map onto visual properties the number of lines in a text file, the size of the file in bytes, and the number of references to external Java methods inside a each class.

For the representation of the reference domain, we abstracted as much as possible from specific representations in order to build a modular visualization tool on which any specific domain could be loaded by extending the core domain. We hence model the domain as a generic graph in which both nodes and edges can be visualized. Visualization is obtained by mapping the core domain on a visual domain, as shown in Figure 3.8.

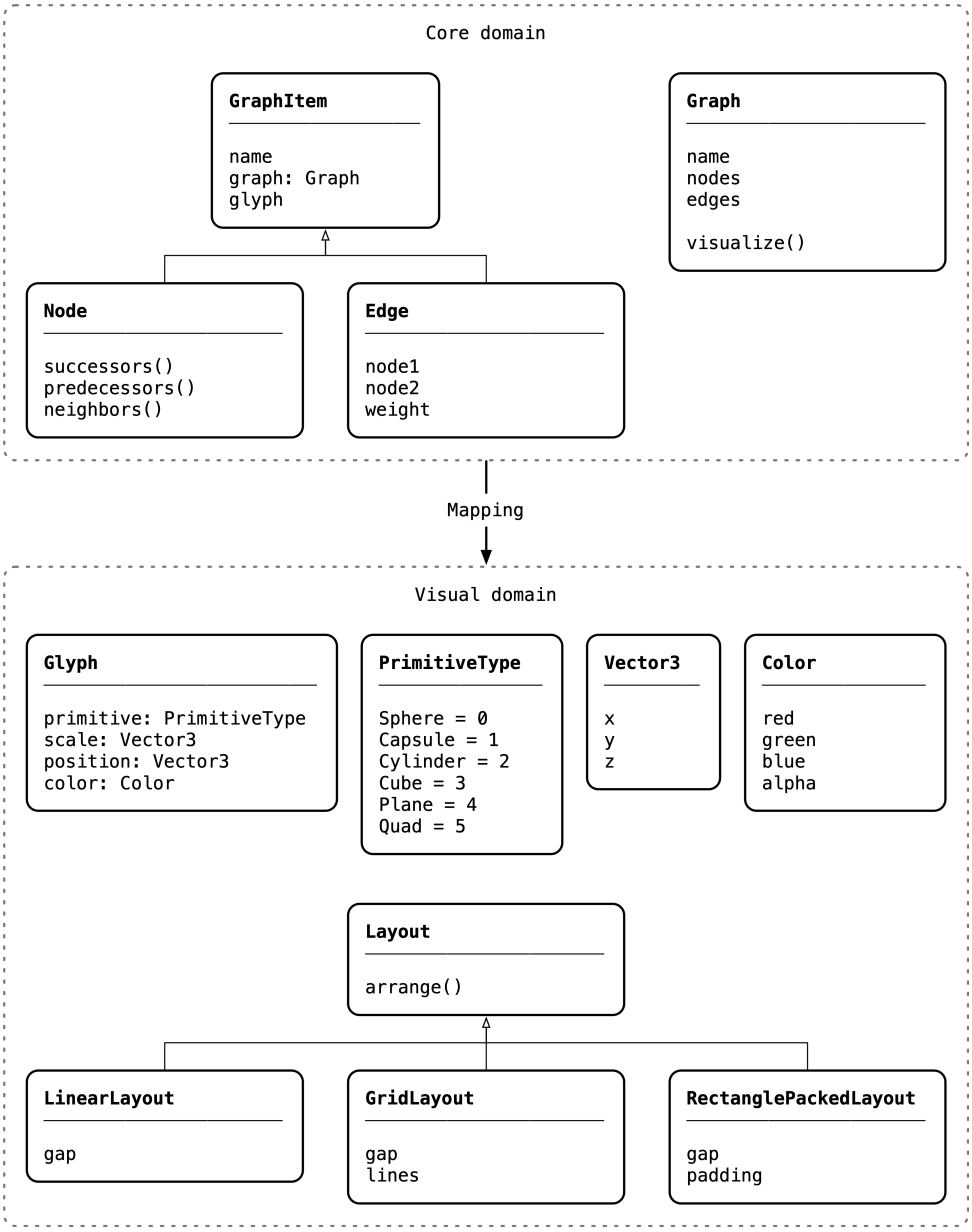

Figure 3.8: The mapping of the core domain to the visual domain.

The core domain consists of four classes that provide the functionality needed to represent a graph and navigate between nodes:

- `GraphItem` acts as a common interface for nodes and edges. It contains basic informations such as the name of the item, the graph to which it belongs and the glyph that depicts the item;
- `Node` contains some methods that help exploring the graph. Since successors and predecessors can be computationally heavy to compute on the entire graph, they are lazy loaded when needed and invalidated when certain operations are performed on the node (*e.g.,* moving a file invalidates the parent predecessor);

- `Edge` represents non-directed edges;
- `Graph` contain nodes and edges that belong to the graph, as well as some utility methods to explore the graph. The `visualize()` method allows the graph to visualize itself.

We model the visual domain taking inspiration from the Unity model, in order to make the representation of data in the back-end and front-end more uniform, thus simplifying the exchange of data between them. It consists of:

- A `Glyph` which corresponds to a GameObject in the Unity model. It is characterized by a `primitive` (*i.e.,* the 3D shape), a three dimensional `scale` vector (*i.e.,* the 3D size), a three dimensional `position` vector and a `color`. Please note that Unity uses x and z coordinates on the horizontal plane and y coordinate for height and elevation. We model all the three dimensional vectors as instances of the `Vector3` class;
- The `PrimitiveType` enum reflects the way Unity handles primitives;
- Layouts are defined on the common interface `Layout` and may have different attributes depending on the functionalities they implement. Layout's attributes allow the user to customize part of the visualization (*e.g.,* the gap between the items in a grid layout).

The `ViewBuilder` is the component responsible for mapping the core domain to the visual domain. It starts from a template called *View Specification* (or viewspec, for short) and builds the visualization. The concept of using a template as a starting point for the visualization is not new and has already beed explored in other research works:

> *A view specification is a template for a view. It consists of a mapping between the domain model and the visual model, a layout, a sorting method for domain objects, a global scaling method for the view [21].*

We model a viewspecs as shown in Figure 3.9. It consists of the classes needed for building the visual depiction for the nodes and edges and for arranging them into the VR:

- `ViewSpec`: The class representing a view specification. It has the following attributes [2]:
  - A `glyph_mapping` component. It is responsible for building the glyph of a given domain entity (*i.e.,* a node or an edge);
  - A `layout` for arranging the glyphs into the VE;
  - A `sorting` method for the entities of the system;
  - A `system_scaling` method for scaling the entire system depiction as a whole.
- `GlyphMapping`: The class representing a mapping between a type of object in the system and a visual representation. For instance, to visualize files and folders in a file system, you need to create two mappings that define how each type should be represented. One mapping will specify how files are visualized, and another will determine how folders are visualized. Thus, the `glyph_mapping` property of the `ViewSpec` is a map from the types of entity in the domain to their visualization;
- `SystemScaling`: We allow the user to scale the system visualization as a whole, enabling both table-scale and room-scale explorations. Table-scale explorations allow the user to have a compact visualization to fit on a virtual table. This visualization is better for visualizing the system from the outside. Room-scale exploration allows the user to explore the visualization by walking into it. Walking in the physical space in which they are located is translated into movement into the VE. `LinearScaling` and `FitInARoom` allows the user to scale the visualization by a scaling factor or to fit into a physical space of given size.

[2] We use camel case for classes and snake case for attributes, according to python notation convention.

- `Sorting`: The abstract class representing the sorting methods for the entities of the system. This allows entities arranged in some layouts to follow a given order. For example, when using a linear layout, the entities can be placed from the newest one to the oldest one. Also note that some layouts (*e.g.,* the city layout) does not support sorting, as they arrange the entities on the base of specific properties (*e.g.,* minimizing the wasted space).

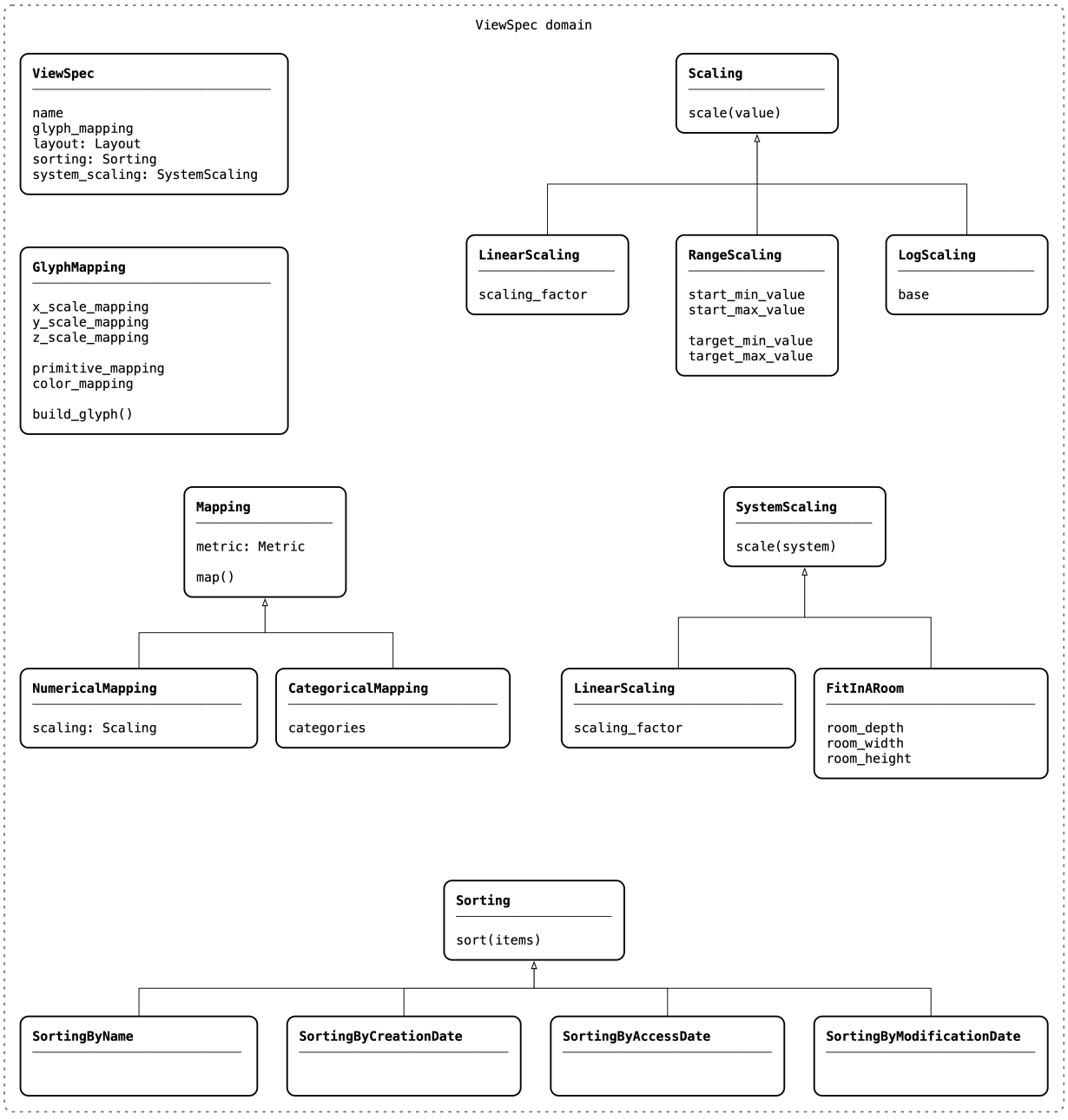

Figure 3.9: The model of a view specification.

Each attribute of a `GlyphMapping` is an instance of the class `Mapping`. As we further explain in Section 3.3, we use *software metrics* to measure certain properties of the domain objects. For now, consider metrics simply as properties of a given object in the domain.

Such properties can be numerical values (*e.g.,* the number of lines in a text file, the size of a directory) or categorical values (*e.g.,* a file can be either a 'text file' or a 'binary file'). A `Mapping` maps the value of a property to a specific visual attribute.

Also glyphs have both numerical and categorical properties. The height of a glyph is, for instance, a numerical visual property while the shape is a categorical property. Domain numerical properties are mapped to visual numerical properties, while categorical domain properties are mapped to categorical visual properties.

Numerical properties can also be scaled through a `Scaling` object. `Scaling` is the abstract class that define the interface for the scaling methods for values:

- `LinearScaling` scales an input value by a scaling factor;
- `RangeScaling` maps an input value from the interval of all the values to a different interval;
- `LogScaling` scales an input value logarithmically.

In Chapter 5 we present two case studies to show how we visualize a reference domain in VR and how we shape the interaction inside the VE. In Section 3.4.1, we show how we model a file system and in Section 3.4.2, we explain how we model a java project.

3.4.1 Model for the File System

The model for the file system case study extends the core model in order to represents files and directories and the tree structure between them, as shown in Figure 3.10. In particular:

- `FileSystemNode` extends the `Node` class and represent a generic node in the file system graph. It can be:
  - A `FileSystemEntity` which is the abstraction for files and directories. Action on the entities (*e.g.,* move or rename a file or a directory) are exposed through methods of this node;
  - A `FileSystemNodeType` which is used to represent the type of a file or a directory. Each file system entity has a type given by its extension (or no extension). We build a categorization of the entities through a taxonomy of the file extensions.
- A `FileSystemEdge` represent the relations between the nodes of the file system. A relation can be either a containment between a directory and a file or the type of the entity through its extension;
- A `FileSystem` is an extension of the `graph` class.

The categorization of the entities can be used to label the entities and allows to collect fine-grained data about the file system. For instance, directories such as the .git directory and files such as the .gitignore files can both be grouped as 'git stuff'.

3.4.2 Model for the Java Project

The model for the java project case study also extends the core model in order to represents java entities (*e.g.,* classes, methods, packages) and the relations between them (*e.g.,* containment, reference, extension), as shown in Figure 3.10. In particular:

- A `JavaNode` is the abstraction of all the entities scraped from a java project. It can be a primitive type, a class, an interface, an enum, a package, a variable, a field, a method or a constructor;
- A `JavaEdge` is the abstraction of all the relations between the nodes. It can be a containment (*e.g.,* a method is contained in a class), a reference (*e.g.,* the call of a method by another method), a type (*e.g.,* a field is an instance of a certain class), an extension (*i.e.,* one class extends another class), an implementation (*i.e.,* a class implements an interface). It can also be a weighted edge that counts the number of references from a node to another one. For instance, if a methods uses three times a field, there is such an edge between them with weight 3;
- A `JavaProject` is an extension of the `graph` class;
- We implement the classes into the `LSP domain` (Language Server Protocol domain) to make the backend communicate with the language server. `Location`, `Range` and `Position` identify the position of a `JavaSymbol` in a java file. We discuss what a language server is and how we use it in Chapter 4.4.5.

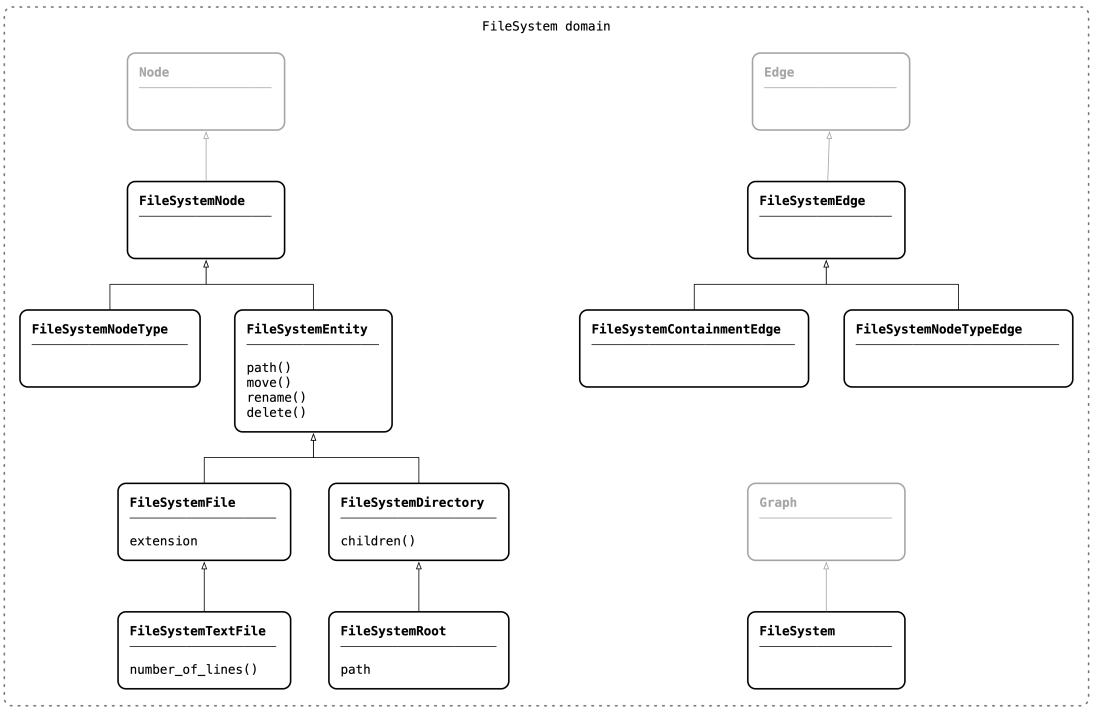

Figure 3.10: The domain developed for the file system case study.

3.5 Reflections

To summarize, we first highlighted the role of metaphors in shaping the way a user perceives the VE and interacts with the virtual objects. Then we presented the metaphors that we designed. This led us to two main insights:

- The city layout provides a compact and insightful way of visualizing a software system. Given its relevance, we extended it to a generic domain, allowing any entity of the model to be visualized as a building in the city visualization;
- The 3D selection menu provides new perspectives in designing VR-native interfaces. It has the potential to evolve into a new approach of building GUIs specifically designed for VR, moving away from traditional 2D UIs. This topic is of great interest to the scientific community and we got positive feedbacks from the VISSOFT [3] scientific community on which we published the work [89].

Then, we presented the main concepts from cognitive science behind visualization, highlighting how the use of metaphors helps reduce cognitive load. After, we introduce the software metrics that we use for the visualization. Finally, we present both a domain model and a visual model for our visualization tool, showing how we map domain entities on visual depictions through the software metrics.

The concepts introduced in this chapter lay the foundation for the implementation of the tool that we present in the next chapter.

[3] See `https://vissoft.info/`

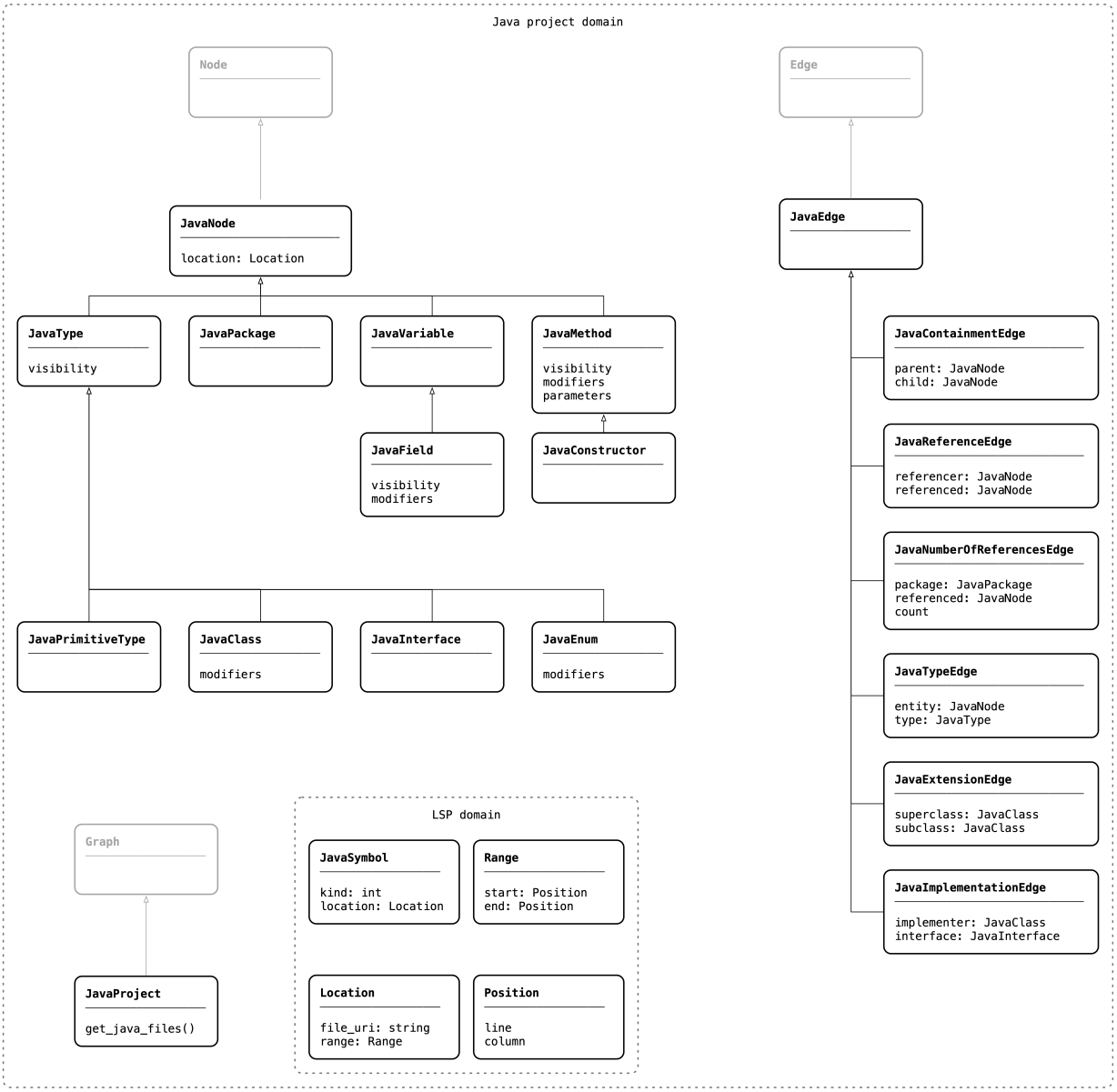

Figure 3.11: The domain developed for the java project case study.

Chapter 4

Implementation

We implemented the tool presented in this chapter incrementally:

1. We started from the 3D selection metaphor and developed IVaR-NI, a proof-of-concept to explore which features could make Interactive VR-Native Interfaces. IVaR-NI focuses on the manipulation of view specifications, allowing to customize the visualization of a domain;
2. Then we developed VirtEx, a tool that allows ti visualization of a domain through the view specifications built using IVaR-NI. It also provides the means navigate into the VE and to inspect the virtual objects.

Then, we started integrating IVaR-NI and VirtEx into a single tool that we present as follows:

- In Section 4.1, we present a comparison of five of the most popular VR headsets in terms of hardware capabilities. This guided us in the selection of the materials for the implementation;
- In Section 4.2, we present the hardware and software technologies we used for implementing our solution;
- In Section 4.3, we present the an overview of the tool, showing the high-level architecture of the front-end and the back-end, as well as how they communicate;
- In Section 4.4, we explore the implementation details of the back-end components, from how the domain is instantiated, to how it is served to the front-end. We also present how the back-end can be extended for the two case studies that we present in the next chapter;
- In Section 4.5, we explore the implementation details of the front-end components, from the tools provided to the user to explore the VE, to the components that run under the hood to make everything work properly.

4.1 Hardware Evaluation

In assessing the available hardware technology, we take into account several aspects, each one of them affecting the final experience in terms of interaction and immersion. As explained in Section 2.3.1, immersion is influenced by the quality of the rendering while interaction depends on the input available methods. Hence, we evaluate five main aspects:

- **Image quality**: The quality of the image is evaluated considering the resolution (*i.e.,* the size in pixels) of the display, the refresh-rate (*i.e.,* times per second in which the display changes image), the optics (*i.e.,* the type of lenses), the technology of the display (*i.e.,* the kind of display and how many of them are used) and the field-of-view (FoV, *i.e.,* how much of the VE is visible at any given time);

- **Input methods**: We take into account the tracking of the user into the VE, the kind of controllers paired with the headset and whether the headset supports hands-tracking and eye-tracking or not;
- **Computational power**: We take into account the capabilities of the CPU and the GPU, as well as the amount of RAM;
- **Connections**: We evaluate if the headset can be used as a stand-alone device as well as connected to a computer.

4.1.1 Comparison of VR Headsets

Before proceeding with the implementation of the tools, we performed a comparison of several VR headsets to decide which one to purchase in order to test the tools. We already had a Meta Quest 2 in our lab but we wanted a more powerful device to test heavier visualizations.

We made an initial selection based on price, thus looking for VR headsets priced between 300.- CHF and 600.- CHF. Another factor taken into consideration was whether or not hand-held controllers were included when purchasing the headset. The selected headsets are shown in Table 4.1.

Since we already had a Meta Quest 2 to compare the candidates with, and some of us already used the PlayStation VR 2, we used them as a yardstick. Thus we included them in the comparison table. To name a winner, we consider three main characteristics (*i.e.,* the display, the input methods and the performance). The following list present a summary of the results:

- **Display**: The Meta Quest 3 and the Pico 4 are the only ones with pancake lenses and this aspect is enough to exclude the other candidates. While the former has a slightly lower resolution, the latter has a significantly lower refresh-rate. Moreover, the Meta Quest 3 has a slightly wider FoV. In conclusion, the Meta Quest 3 has a better display;
- **Input**: The Portal QLED View does not feature hand-tracking, while the Meta Quest 3 and the Pico 4 do. While the Pico 4 has one more camera for tracking, the Meta Quest 3 feature a depth sensor. This point is a draw since the hardware alone is not enough to evaluate the tracking;
- **Performance**: The Meta Quest 3 is the headset with the newest hardware. Both CPU and GPU are more powerful than the ones on the other headsets. In conclusion, the Meta Quest 3 has the better computing hardware.

The Meta Quest 3 is the best among the compared VR headsets in terms of display quality and performance, hence we choose the Quest 3 as out testing platform.

4.2 Materials for the Implementation

In this section we present the hardware and software technologies we used to implement the tool that we present in the following sections.

4.2.1 Frontend Software

The software technologies we use in the frontend revolve around Unity and its supporting toolkits. It enables the creation of 3D environments and interactions in the VR space. The technologies used are:

- **Unity**: Unity served as the primary development platform for the frontend. Its interactive development environment allowed us to quickly develop prototypes of 3D objects and then save them as prefabs and reuse them later via the scripting language. We designed the frontend using Unity's scene editor, where all assets (3D models, textures, scripts) can be managed. We implemented interactions and behavioral logic using C# scripting;
- **Oculus XR Plugin**: It is a Unity plug-in that provides display and input support for Oculus devices;

	Quest 2	Quest 3	Pico 4	Portal QLED View	PlayStation VR 2
Manufacturer	Meta	Meta	Pico	Pimax	Sony
Release date	Oct. 13 2020	Oct. 10 2023	Oct. 18 2022	Oct. 30 2022	Feb. 22 2023
Retail price	€299.99	€549.00	€420.00	€599.99	€659.00
Resolution per eye	1832x1920	2064x2208	2160x2160	1920x2160	2000x2040
Display refresh-rate	120 Hz	120 Hz	90 Hz	144 Hz	120 Hz
Optics	Fresnel lenses	Pancake lenses	Pancake lenses	Aspheric lenses	Fresnel lenses
Display technology	Single Fast switch LCD	Two LCD displays	Two LCD displays	Single QLED	Two OLED displays
Field-of-view	97°	110°	104°	100°	110°
Tracking	6 DoF Inside-out via 4 integrated cameras	6 DoF Inside-out via 4 integrated cameras and depth sensor	6 DoF Inside-out via 5 integrated cameras	6 DoF Inside-out via 4 integrated cameras	6 DoF Inside-out via 4 integrated cameras
Controllers	Two Oculus Touch (3rd generation) with 6 DoF	Two Meta Quest Touch Plus Controllers with 6 DoF	Two Pico 4 Controllers with 6 DoF	Two Pimax Portal Controller with 6 DoF	Two Playstation VR2 Sense Controller with 6 DoF
Hands-tracking?	Yes	Yes	Yes	No	No
Eye-tracking?	No	No	No	No	Yes
Operating System	Android	Android	Android	Android	N/A
Chipset	Qualcomm Snapdragon XR2	Qualcomm Snapdragon XR2	Qualcomm Snapdragon XR2	Qualcomm Snapdragon XR2	N/A
CPU	Octa-core Kryo 585 (1 x 2.84 GHz, 3 x 2.42 GHz, 4 x 1.8 GHz)	Octa-core Kryo (1 x 3.19 GHz, 4 x 2.8 GHz, 3 x 2.0 GHz)	Octa-core Kryo 585 (1 x 2.84 GHz, 3 x 2.42 GHz, 4 x 1.8 GHz)	Octa-core Kryo 585 (1 x 2.84 GHz, 3 x 2.42 GHz, 4 x 1.8 GHz)	N/A
GPU	Adreno 650	Adreno 740	Adreno 650	Adreno 650	N/A
Memory	6 GB	8 GB	8 GB	8 GB	N/A
Storage	128 GB	128 GB	128 GB	256 GB	N/A
Untethered mode?	Yes	Yes	Yes	Yes	No
Tethered mode?	Yes	Yes	Yes	Yes	Yes

Table 4.1: Comparison of the hardware specifications between the five selected VR headsets.

Figure 4.1: The Meta Quest 2 its two Oculus Touch controllers.

Figure 4.2: The Pico 4 its two Pico 4 controllers.

Figure 4.3: The Pimax Portal QLED View and its two Pimax Portal controllers.

Figure 4.4: The PlayStation VR2 and its two Sense controllers.

- **XR Interaction Toolkit**: It is a package that handles user interactions within the VR environment. This toolkit simplified the management of input from the Meta Quest 3's controllers, providing pre-built interaction models such as ray-casting, direct object manipulation, and GUI handling. It also offers out-of-the-box support for interacting with virtual objects, making it easy to incorporate real-world physics and behavior into the VR environment;
- **OpenXR Plugin**: It is a plug-in based on an open, royalty-free standard created by Khronos that aims to simplify AR/VR development by enabling developers to target a diverse array of AR/VR devices;
- **C# Scripting**: We used C# for the frontend logic. It is Unity's default programming language for creating behaviors (*i.e.,* scripts attached to 3D objects) and handling interactions. We use C# scripts to manage how the user interacts with virtual objects, how data is sent to the Python backend, and how the VR environment retrieve data from the backend outputs. These scripts also handle communication with the backend REST API.

4.2.2 Frontend Hardware

We use a Meta Quest 3 VR headset for testing, developing the software on a MacBook Pro. Since Meta doesn't support tethered mode on macOS, we se the Quest 3 in untethered mode, allowing for greater freedom of movement during exploration. It offers high-quality visual rendering, hand-tracking, and controller interaction, making it an excellent platform for VR development. The setup includes the headset and two controllers, as shown in Figure 4.5.

4.2.3 Backend Software

We developed the backend software using Python and some of its libraries. One of the reasons we chose python is its reflexivity, which allowed us to inspect the code of the domain model, as further

Figure 4.5: The Meta Quest 3 head-mounted display and its two Meta Quest Touch Plus hand-held controllers.

explained in the following chapters. The materials we used, including some libraries and frameworks to handle communication and layouting, are:

- **Python**: It served as the backend for computational tasks, data scraping and data processing. The backend handles inputs from the VR environment, process them, and return results to the Unity frontend in real time. Python's flexibility allowed for rapid development and testing of backend features;
- **Flask**: A lightweight web server that we use to expose certain Python functions as REST APIs. This allows Unity to make HTTP requests to the backend to get data and execute actions on the domain model;
- **rectangle-packer**: A library for rectangle packing that arranges items within a container to minimize wasted space. It's useful for layouting because it optimizes space and reduces clutter.

Another key component of the backend is the Eclipse JDT Language Server (LS) that we use to interact and manage java code, as well as to retrieve document symbols and perform some refactorings. The version we used is v3.8.26 and it runs on top of Java 21.0.1.

4.2.4 Backend Hardware

The backend software run on a 16-inch 2019 MacBook Pro, a high-performance laptop powered by a 9th-gen Intel core i7 CPU, an AMD Radeon Pro 5300M GPU and 16GB of RAM.

4.2.5 Profiling Tools

The Unity Profiling Tool [1] is a powerful utility integrated into Unity, designed to help developers optimize the performance of their games and applications. It provides detailed insights into how different parts of a game are using system resources, enabling the identification and troubleshooting of performance bottlenecks.

[1] See `https://unity.com/features/profiling`

The Unity Profiler can monitor the performance of a game in real-time as it runs in the Unity Editor or on a target device. This helps developers understand how their game behaves on different platforms. It can analyze:

- **CPU profiler**: It breaks down CPU usage by processes, showing which scripts and systems are consuming the most processing time. It helps pinpoint areas where code optimizations or system adjustments are needed;
- **GPU profiler**: Tracks GPU performance to help developers understand the cost of rendering tasks such as lighting, shadows, post-processing effects, and more. This is crucial for optimizing graphics-heavy games;
- **Memory profiler**: Monitors memory allocation, showing how much memory is used by scripts, meshes, and other assets. It helps identify memory leaks and unnecessary memory usage that could slow down performance;
- **Rendering profiler**: Provides detailed information about rendering processes, including draw calls, overdraw, and shader execution, helping developers fine-tune their rendering pipeline;
- **Physics profiler**: Provides insights into how physics calculations are performed, tracking the cost of physical interactions such as collisions, rigid body dynamics, and more.

The Profiler also provides a timeline view where you can visualize the sequence of operations across different components (*e.g.,* CPU, GPU, rendering), helping you see how different tasks are executed over time, as shown in Figure 4.6. This is the feature that we used the most, as it provides an overview of the time spent in each frame and a detailed view of what processes take longer during a frame. This view allows assessing which components slow down the rendering, thus reducing performances.

Figure 4.6: The timeline view of the unity profiler, showing hardware resources utilization and how the time it takes for a frame to be rendered is divided between the various components.

4.3 Tool Architecture

The core tool consists of two main software components, the back-end and the front-end, as shown in Figure 4.7. The front-end is the component responsible for visualizing the model and managing the

interactions with the user. The interaction takes place through the hardware components of a VR headset (*i.e.,* a Meta Quest 3). The back-end is the component that implement the model's logic. It is responsible for scraping and instantiating the domain and providing the data to the front-end. The communication between the front-end and the back-end takes place through a REST API service that runs in the back-end. It exposes the data to the front-end and allows the interactions from the user to be signaled to the back-end.

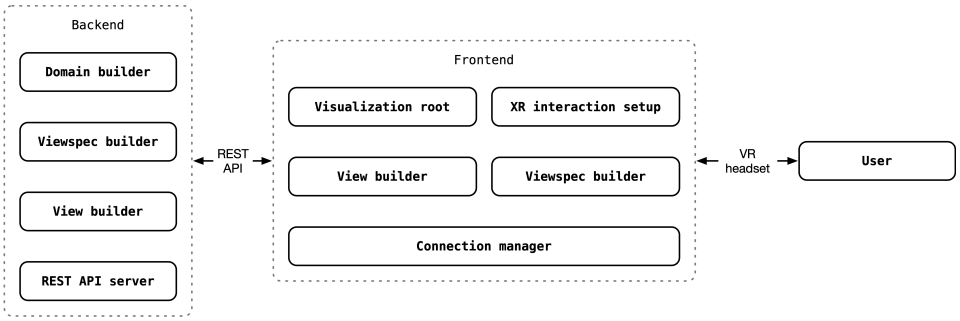

Figure 4.7: The high-level architecture of the tool.

For the implementation of the case studies presented in Chapter 5, we extend the core tool with the additional features needed to build the domain to visualize. In particular, for the file system case study, we extend the tool with the ability to read and interpret data from the file system, as shown in Figure 4.8. For the java project case study we extend the core tool with the ability to exchange data with a language server and to parse the data collected from it, as shown in Figure 4.9.

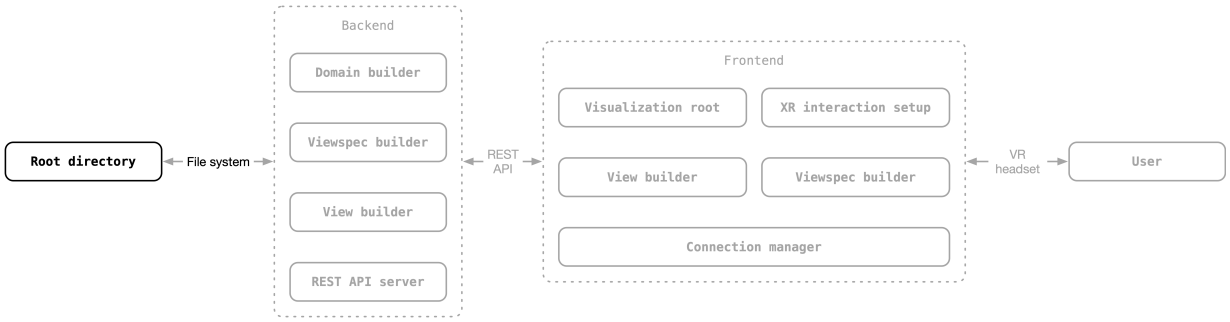

Figure 4.8: The core tool extended for the file system case study.

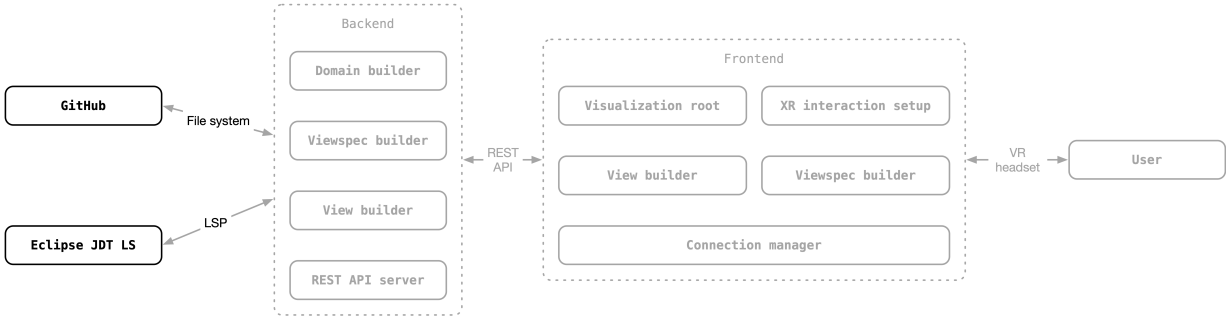

Figure 4.9: The core tool extended for the java project case study.

In Section 4.4, we further explore in detail the structure and functioning of each component of the back-end. In particular we explain the details of four high-level components:

- The **domain builder** collects the data to build the domain (*e.g.,* the files and folders of a file system, and the relations between them);
- The **viewspec builder** reflectively analyze the code of the domain model and provides the frontend with all the data needed to build a viewspec;

- The **view builder** maps metrics to graphical properties, creating glyphs, and arranges the glyphs in the VE according to a predefined layout;
- The **REST API server** exposes backend functionalities to the frontend.

In Section 4.5, we further explore in detail the structure and functioning of each component of the front-end. In particular we explain the details of five high-level components:

- The **connection manager** is responsible for managing the connection to the back-end. It requests data, signal interactions, serialize and deserialize data;
- The **viewspec builder** is the front-end counterpart of the viewspec builder component in the back-end. It allows users to build a view specification (*i.e.,* a specification of how the data is visually depicted);
- The **view builder** builds the visualization. For each object of the domain, it instantiates the glyph that depict it according to a viewspec;
- The **XR interaction setup** manages the hardware associated with VR devices;
- The **visualization root** acts as an entry point for the visualization.

4.4 Back-End Architecture

The core back-end part of the tool consists of four main components, as shown in Figure 4.10. The domain builder manages the domain model, the view builder manages the visual model, the REST API server manages the communication with the backend and the viewspec builder allows the customization of the view specifications.

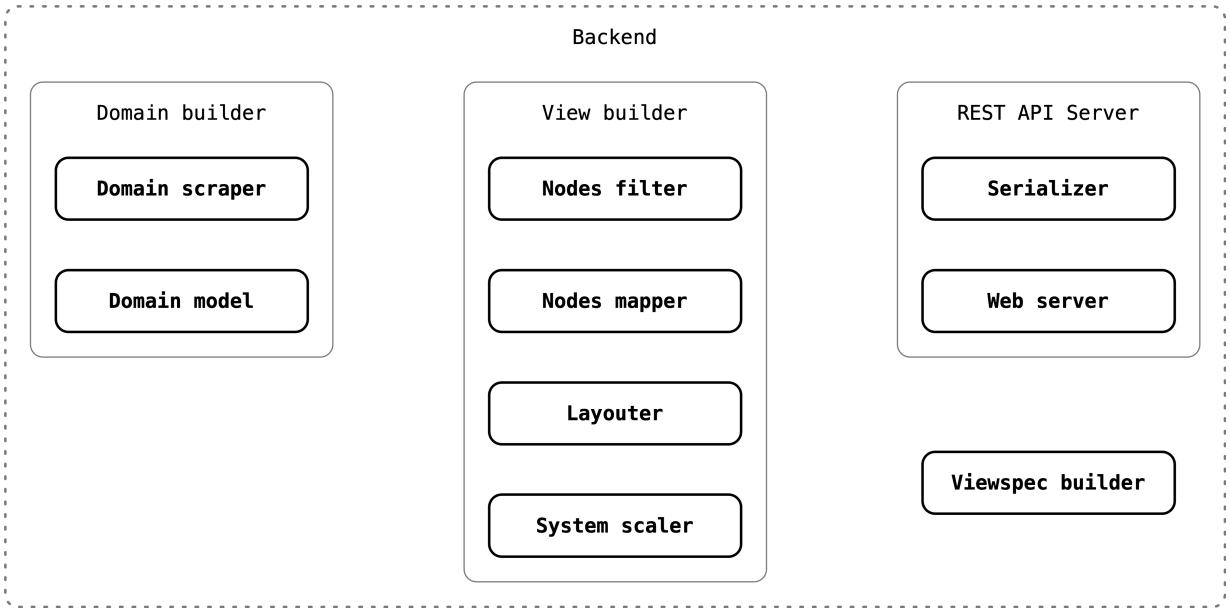

Figure 4.10: The core back-end architecture.

4.4.1 Domain Builder

This component of the architecture provides the base tools for scraping data from various sources, processing it, and converting it into a structured domain model (*i.e.,* a graph-based model). When the core back-end is extended, the extension can rely on these tools to scrape the domain. It consists of the following sub-components:

- **Domain Scraper**: This component extracts data from the domain. It is responsible for gathering raw data from external sources;
- **Domain Model**: This is the final representation of the domain data, structured in a way that can be further processed or queried by other components of the system.

4.4.2 View Builder

This component is responsible for processing the data provided by the Domain Builder and preparing it for the visualization. It has the following components:

- **Nodes Filter**: This module filters out irrelevant or unnecessary nodes based on specific criteria. It ensures only useful information is passed on;
- **Nodes Mapper**: Once the data is filtered, this component maps the remaining nodes to a glyph that can be visualized;
- **Layouter**: This component handles how the data will be visually arranged, preparing it for presentation in a particular layout;
- **System Scaler**: The system scaler adjusts the size of the visualization as a whole, as explained in Section 3.4.

4.4.3 REST API Server

This part of the architecture exposes the processed data via a RESTful API, allowing other systems or clients to interact with it. It has two sub-components:

- **Serializer**: This component transforms the domain model data into a format (*i.e.,* JSON) that can be sent over the network via the API;
- **Web Server**: The web server acts as the interface between the backend and clients, handling incoming API requests and delivering the serialized data in response.

4.4.4 Viewspec Builder

This component is part of IVAR-NI and collaborates with both the domain builder and the view builder. It reflectively analyses the code of the domain model and the visual model to provide the front-end with the capabilities of building a view specification [89]. For example, if the core domain is extended with a new layout (*e.g.,* a spiral layout), the new layout option is automatically added to the viewspec builder in the front-end. Moreover, if the new layout has customizable parameters (*e.g.,* the distance of the line from the center) it is automatically provided to the front-ent, allowing the user to customize it.

4.4.5 Tool Extension for the Case Studies

As we explain in Section 4.3, the core tool is extended with new features to implement the two case studies presented in Chapter 5. In particular, for the case study on the file system, the tool is extended with the ability to manipulate the file system, for the case study on the java project, the tool is further extended with a Language Server (LS) that can manipulate java code.

The tool extensions reside in the back-end architecture. The ability to manipulate the file system is built into the file system model itself, as shown in Figure 4.11. We implement the following features:

- Move, rename and delete files and directories;
- Get the children of a directory;
- Build the file system path of files and directories by walking from the root of the file system tree to the desired node.

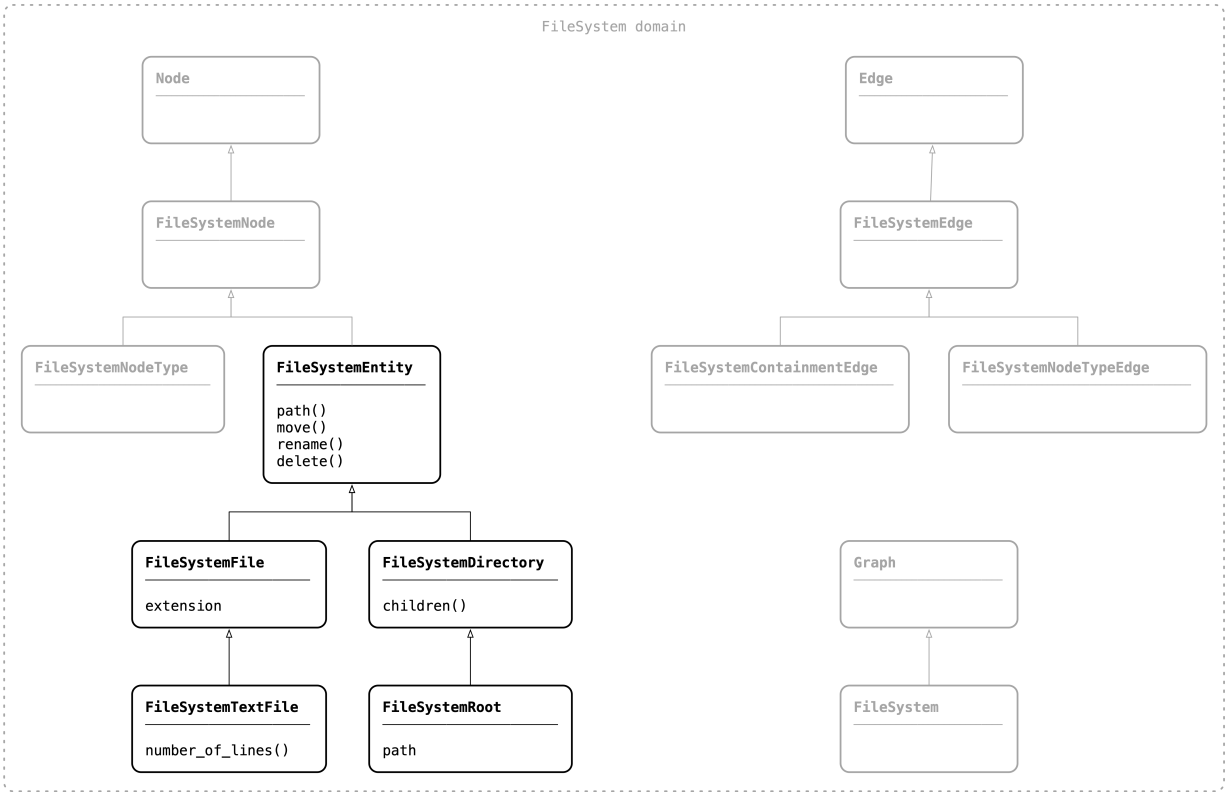

Figure 4.11: The file system operations built into the file system model.

To instantiate the graph representing the domain of the file system, the domain builder starts by instantiating a root directory node of a given root directory. It then analyses all its children, instantiating a new node according to the type of the children (*i.e.,* directory, file, text file). Every time it instantiate a child node, it also instantiate a new containment edge between the parent directory and the child node. The process is then repeated for each sub-directory found in the current directory.

For the java project case study, we further extend the back-end with the components needed to interact with a java project by means of a Language Server (LS).

> *A language server is a service that provides language intelligence tools (*i.e., *programming language-specific features like code completion, syntax highlighting and marking of warnings and errors, as well as refactoring routines)* [2].

The communication between the back-end and the LS is carried out via the LS middleware component, as shown in Figure 4.12. This component acts as both a client for the LS and a server for the other components of the domain builder. It exposes the functionalities of the LS by abstracting the low-level messages with method calls. We implement the following methods:

- `DocumentSymbols`: This method extracts the java symbols from a given java document (*i.e.,* a java file). A symbol is a generic meaningful element in a programming language that can be identified and interacted with by a development environment (*e.g.,* variables, methods, classes). Version 3.17 of the LSP is able to identify 26 different types of symbols [3];
- `SymbolReferences`: The references (*e.g.,* a method call) from the entire project to a specific symbols;
- `HoverString`: The hover string contains contextual information (*e.g.,* the package of a class, the path of a method) to be shown when the mouse is hovered on a specific symbol in the IDE served by the LS. To extract additional metadata from this string, we parse it. Unfortunately

[3] See `https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/`

the hover string is not homogeneous between the different types of symbols and must be parsed differently from one symbol to another;

- `CodeActions`: The actions used to manipulate the java symbols (*e.g.,* a refactoring action);
- `Signature`: The string containing the name and parameters list of a method or constructor. It also needs to be parsed to extract information.

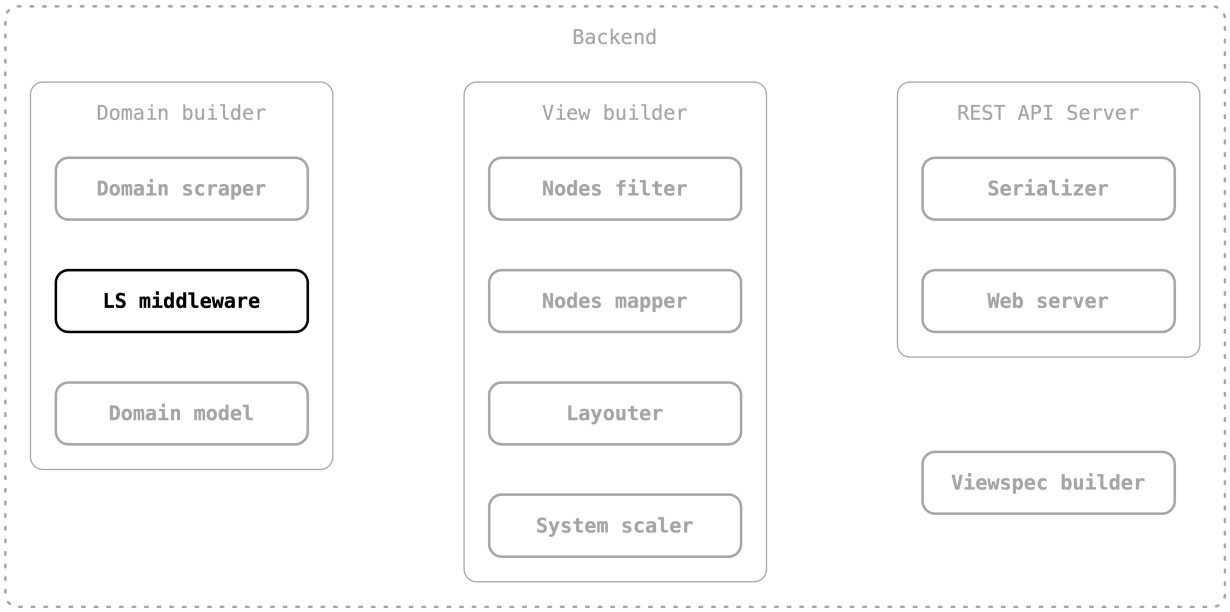

Figure 4.12: The back-end extended with the LS middleware component.

For the java project case study we use the Eclipse JDT Language Server [4]. The communication between the LS middleware and the JDT LS requires the implementation of the Language Server Protocol (LSP) [5].

> *The language server protocol is an open, JSON-RPC-based protocol for use between source code editors and language servers* [6].

Our implementation of the LSP includes the classes needed to query the LS in order to gather the java symbols from a java file. In particular:

- `JavaSymbol`: This class represents a generic symbol in a java file;
- `Position`: A position is a pair of line and column used as coordinates into a text file;
- `Range`: Is a portion of a text file delimited between the `start` position and the `end` position;
- `Location`: This class identifies the location of a symbol in a java file. The `uri` is the path of the java file in the file system. The `range` is the portion of text in which the symbol is present in the file;

4.5 Front-End Architecture

The architecture of the front-end, shown in Figure 4.13, consists of a multitude of components. Some of them have a visual representation in the VE and some others run under the hood to make everything work properly. In order to make it easier to distinguish them, we colored the former light grey. We

[4] See `https://github.com/eclipse-jdtls/eclipse.jdt.ls`
[5] See `https://microsoft.github.io/language-server-protocol/`

now present an high-level overview of the main components, and then we go in the details of each one of them:

- The **XR (eXtended Reality) Interaction setup** is a mix of scripts from the Unity XR plug-in, custom scripts and visual components. It is mainly responsible for managing the input of the hardware controllers and to output the virtual scene to the displays of the HMD. Moreover, it has a set of tools that the user can use to interact with the VE;
- The **Connection manager** queries the back-end for collecting the data to visualize and signals to the back-end the interactions from the use, to make the model respond accordingly;
- The **Viewspec builder** is a set of interactive tools within the VE that allow users to design and customize the visualization of a system to their preferences;
- The **View builder** is responsible for instantiating game objects given a domain;
- The **Visualization root** is the entry point of the visualization. It is responsible for positioning itself (and thus the entire system) within the scene.

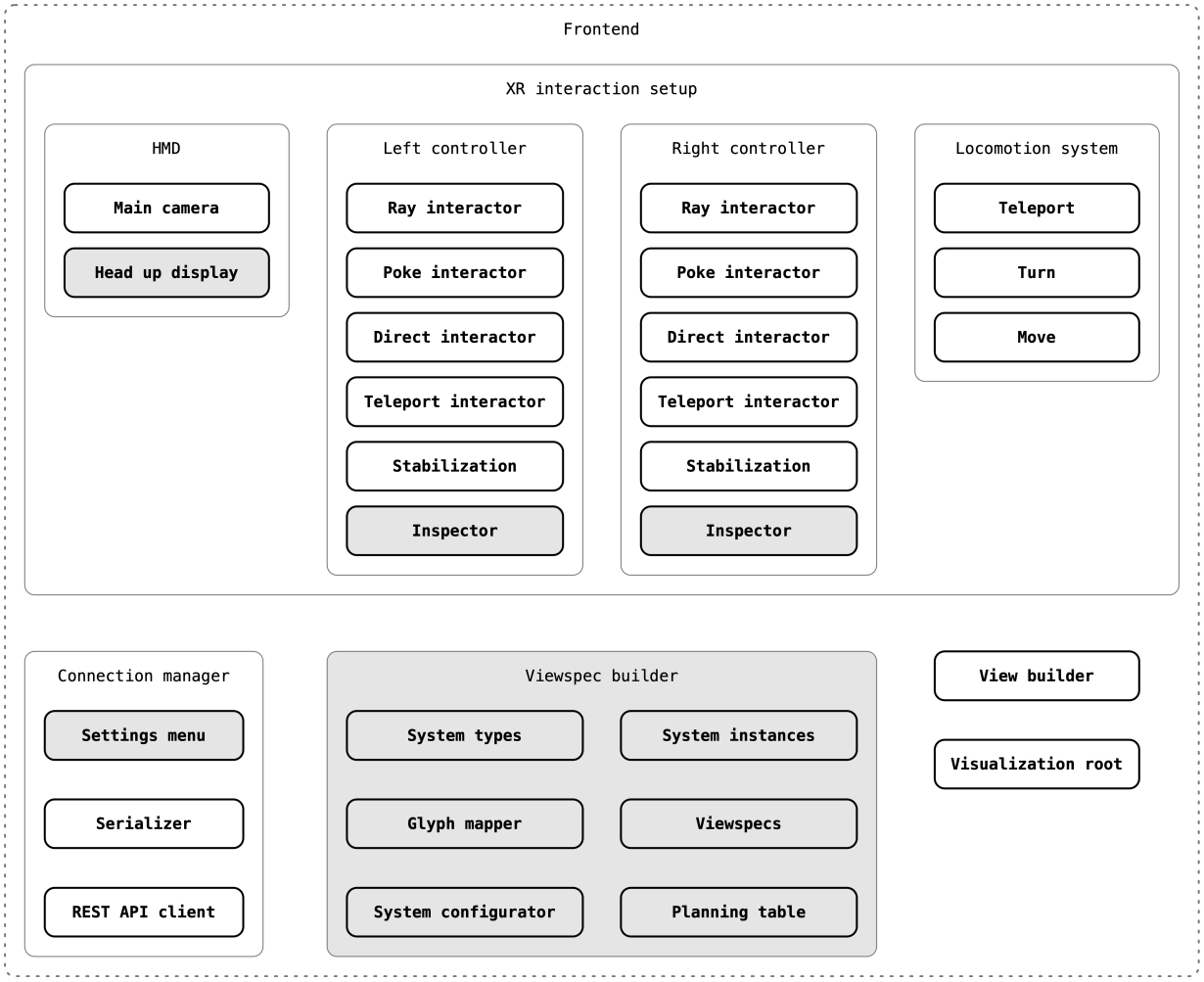

Figure 4.13: The architecture of the front-end.

4.5.1 XR Interaction Setup

The XR interaction setup consists of four main components:

- The **Head-Mounted Display (HMD)** is responsible for the output of the rendered VE to the user through the displays of the VR headset;

- The **Left controller** and **Right controller** are responsible for reading the input from the two hand-held VR controllers. They are equipped with the tools needed for traveling the VE and interacting with the virtual objects;
- The **Locomotion system** manages the ways in which the user can travel into the VE.

Head-Mounted Display

The HMD is responsible for presenting the rendered VE to the user. The main camera captures the virtual scene and generates two images, one for each eye, with slightly different perspectives to create a sense of depth. The main camera also allows for certain visual quality settings, to be adjusted. In particular, we tweaked both the level-of-detail [7] and the anti-aliasing [8] to find the right compromise of quality and performance.

> *The level-of-detail (LOD) refers to the complexity of a 3D model representation.*

> *Anti-aliasing is a computer graphics technique that smoothes jagged edges on curves and diagonal lines. It helps to make digital images appear more realistic by eliminating the "staircase" effect that often appears on curved or angled lines.*

The head-up display (HUD) is a virtual panel displayed in front of the user and attached to their movement of the head. It provides information about the status of the system and user controls, as shown in Figures 4.14, 4.15, 4.16. It allows users to adjust some settings, thus customizing the interaction with the VE (*e.g.,* how traveling occurs), to manage the connection to the back-end, and to view system logs. It also allows the user to view a mini-map of the system and to teleport to a specific location. The HUD also provides real-time display of frames-per-second (FPS) at which the rendering is occurring, giving insight into the system's performance.

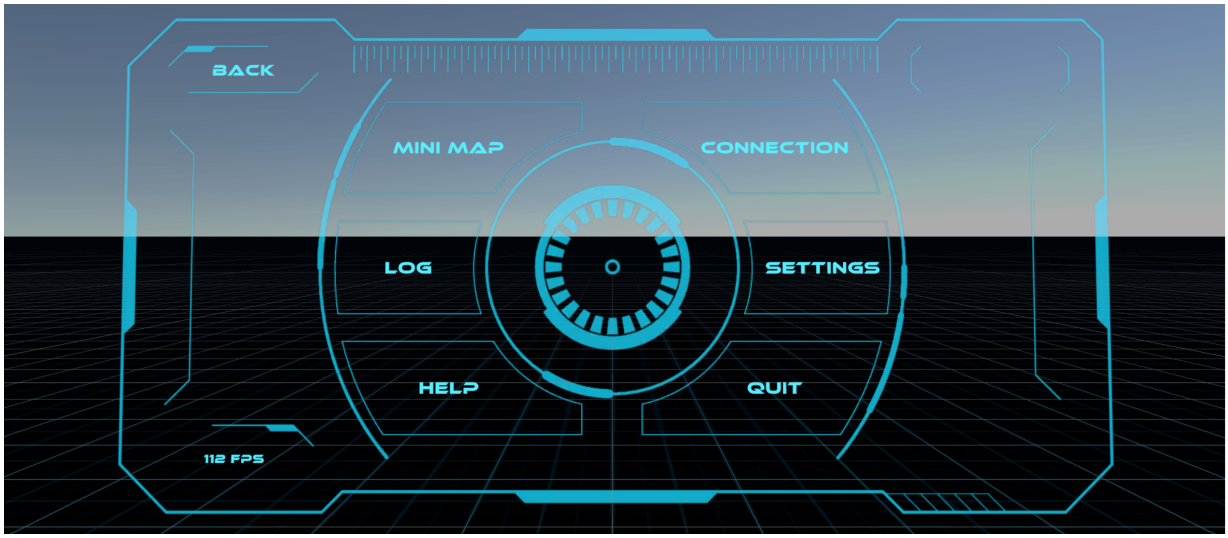

Figure 4.14: The main screen of the HUD. Real-time rendering FPS on bottom-left corner.

Left and Right Controllers

The two controller components share an identical structure, with the only differences being the mirrored 3D model depicting them and settings adjusted to manage left- and right-hand interactions. Both controllers are equipped with several interaction tools:

[7] See `https://en.wikipedia.org/wiki/Level_of_detail_(computer_graphics)`
[8] See `https://www.lenovo.com/us/en/glossary/what-is-anti-aliasing/`

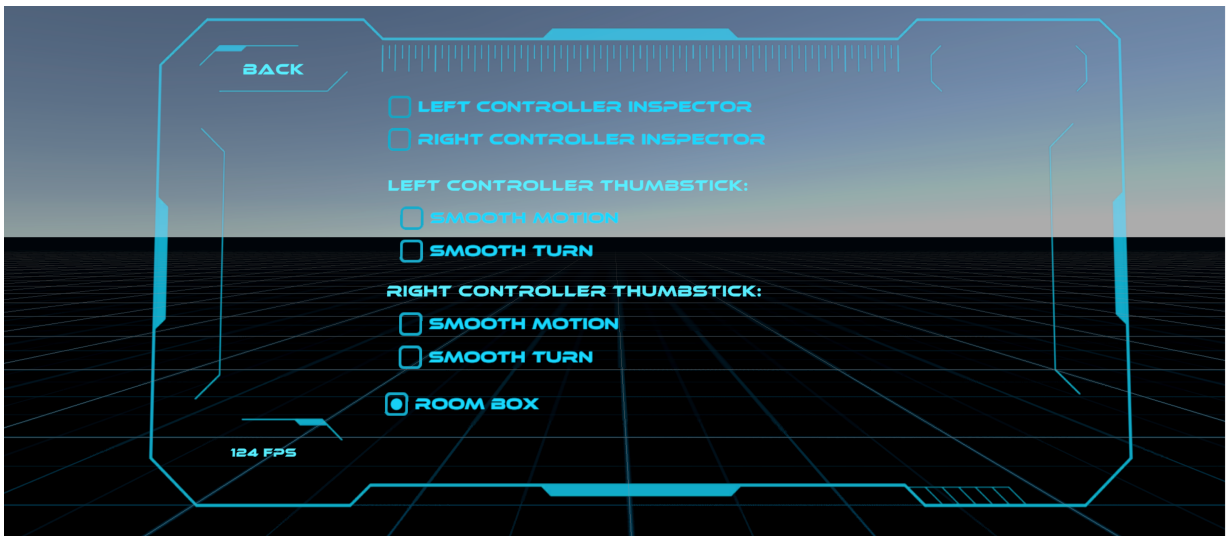

Figure 4.15: The controllers settings screen in the HUD.

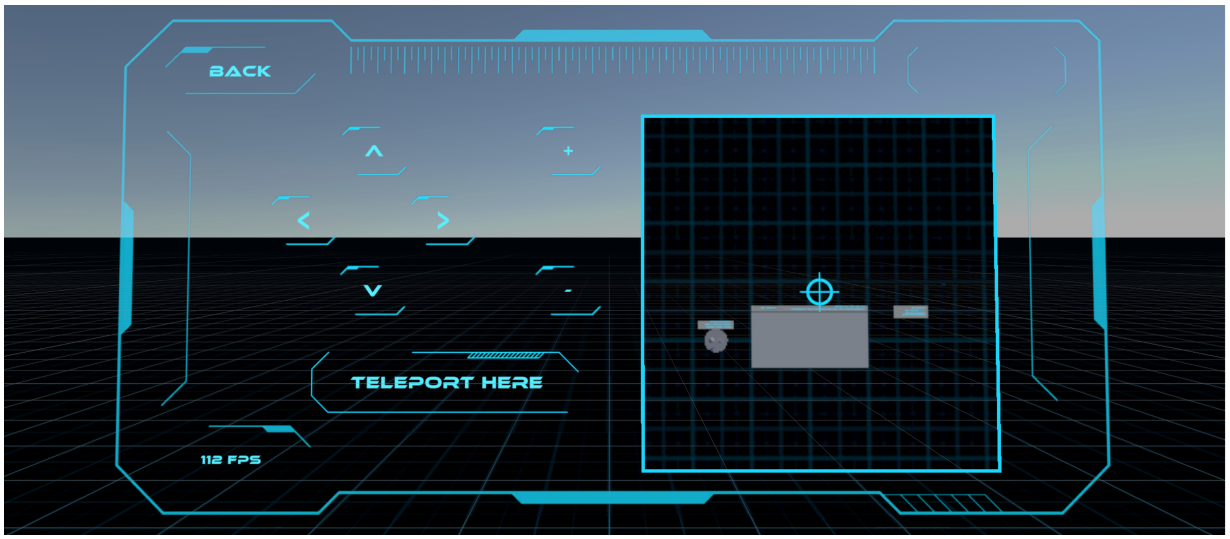

Figure 4.16: The mini-map screen in the HUD.

- The **Ray interactor** enables users to grab or hover over objects from a distance using ray-casting (*i.e.,* a ray projected from the controller);
- The **Poke interactor** and the **Direct interactor** implement the same interactions of the ray interactor, but for close-range manipulation, allowing users to directly touch or poke objects;
- The **Teleport interactor** communicates with the locomotion system's teleport component, as well as with any objects that include a teleport area component or teleport anchor component. Teleport components attached to the 3D objects allow these objects to be a teleport destination;
- The **Stabilizer** addresses the challenge of interaction at a long distance. Small rotations or movements of the controller can cause significant displacement of the ray cast from the controller, making precise targeting difficult. The stabilizer mitigates this by dampening small movements and magnetically snapping the ray to the nearest object. This makes it easier to select and interact with small or distant objects;
- The **Inspector** component allows users to retrieve detailed information about objects in the scene on demand, as described in Sub-Section 3.1.4. It includes a small virtual display called a tool-tip, which displays brief information about an object, such as its name or path. By pressing a button on the controller, users can bring up a larger virtual display (*i.e.,* the advanced tool-

tip) that provides extensive information, such as how specific metrics are mapped to the object's visual properties.

Locomotion system

The locomotion system enables different methods for users to navigate into the VE, offering options that help minimize motion sickness. These include:

- **Teleport**: This technique allows users to move instantly to a new location. It is proven to reduce the discomfort that can come from continuous movement [85];
- **Turn**: This method can be either continuous or implemented as snap-turns (*i.e.,* rotations by fixed increments). Snap-turning is commonly used to avoid motion sickness that might result from smoother, continuous turning [85];
- **Move**: Continuous movement, which is similar to traditional video-game controls, allows users to move freely in any direction using the controller's thumb-sticks.

4.5.2 Connection Manager

The connection manager is responsible for downloading the data from the front-end and signaling the interactions from the user. Data from the back-end are deserialized to be used and interactions are serialized before being sent to the back-end. A settings menu displayed into the HUD allows the user to change the IP address and the port of the back-end, as shown in Figure 4.17. The REST API client is the component that actually sends the web requests to the APIs of the back-end REST API server.

Figure 4.17: The connection menu part of the HUD.

4.5.3 Viewspec Builder

The viewspec builder consists of a set of tools that enables users to configure the visualization directly within the VR environment. This set of tools is placed in a VE called *lobby* shown in Figure 4.18.

The idea behind the viewspec builder is to make the users interact with the same 3D objects that represent the model's components (*e.g.,* cubes, spheres), rather than using 2D interfaces displayed on virtual panels. This set of tools includes the glyph mapper, the viewspecs area, the instances area and the planning table.

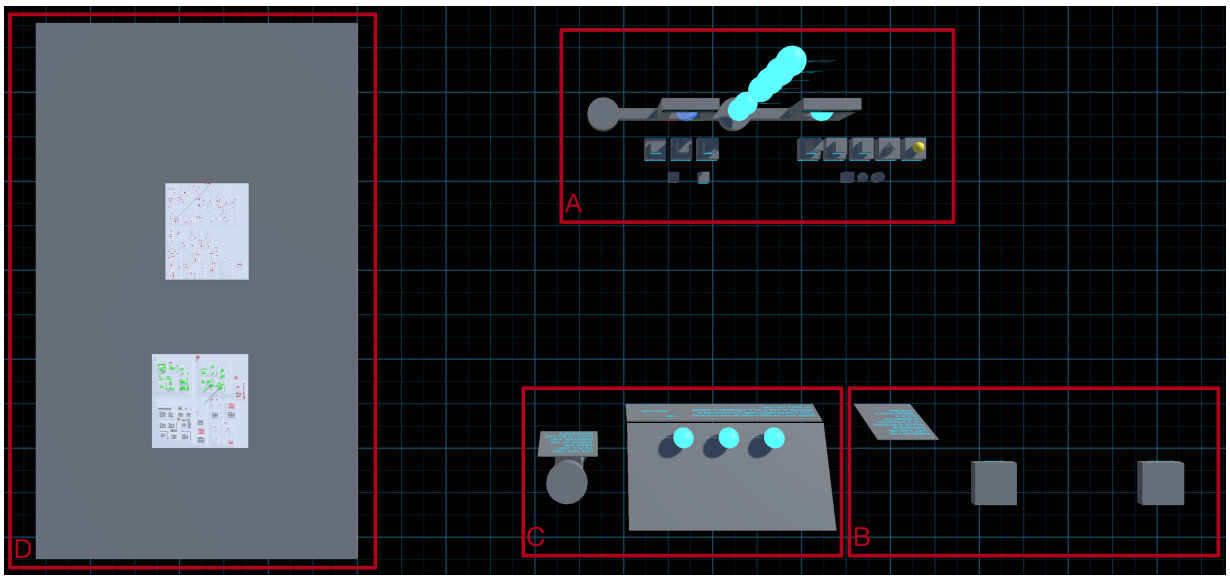

Figure 4.18: The lobby and the set of tools to build a viewspec. The glyph mapper (A), the viewspecs area (B), the instances area (C), the planning table (D).

Glyph Mapper

In Figures 4.19, 4.20, 4.21, 4.22, we show the interface of the glyph mapper and how the user can build a view specification following these steps:

1. The glyph mapper displays all the types of domain on which the user can build a visualization (*e.g.,* file system). A type of domain cannot be directly visualized, as it is not a specific instance of a domain. It is rather used to create a viewspec with which the user can visualize the instances of that specific type. The user can select a type of domain as described in Sub-Section 3.1.2: They grab the glyph representing the chosen system and place it in the domain selection window;
2. Once the type of domain is selected, the user can choose the entities of the domain that they want to visualize (*e.g.,* files of a file system). The selection happens in the same way of the selection of a type of domain. Selecting a type of domain also allows the user to configure some domain-wide options, such as the layout in which to arrange the domain components.
3. Every time a user select a domain entity, the glyph mapper shows the visual options on which the user can map the domain options (*i.e.,* x scale, y scale, z scale, primitive and color);
4. By interacting with the *expand* button of a property, the user can visualize the options that can be selected for that property (*e.g.,* the available layouts for the layout property);
5. In the same way, the user can visualize the domain properties that can be mapped on the visual properties and thus build the viewspec. The selected options are placed on top of the platform representing the property. Some options can be composed by stacking one on top of the other, some of them can be configured through a configuration menu, finally some others can map categorical options to categorical properties.

Viewspecs Area, Instances Areas and Planning Table

The viewspecs area displays all the available viewspecs. Each viewspec is represented as a platform on which the user can place a system instance. The instances area displays all the system instances available for visualization. As the user places a system instance on the viewspec platform, the tool builds a preview of the system instance visualized through that viewspec, as shown in Figure 4.23.

The preview of the visualization can then be used to plan the final visualization scene through the planning table, as shown in Figure 4.24. The visualization scene is the VE in which the user enters for the exploration.

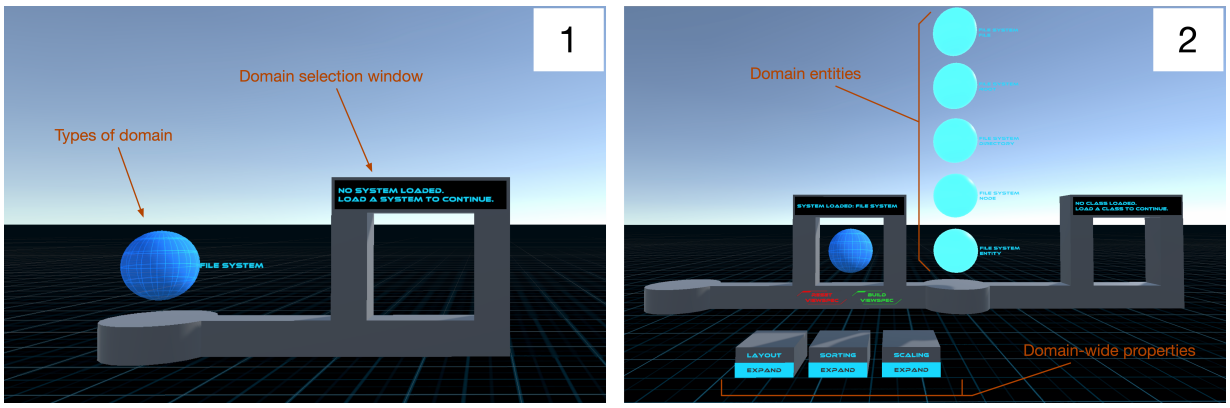

Figure 4.19: The user selects a type of domain by placing it in the domain selection window (1). Domain entities and domain-wide properties are shown accordingly (2).

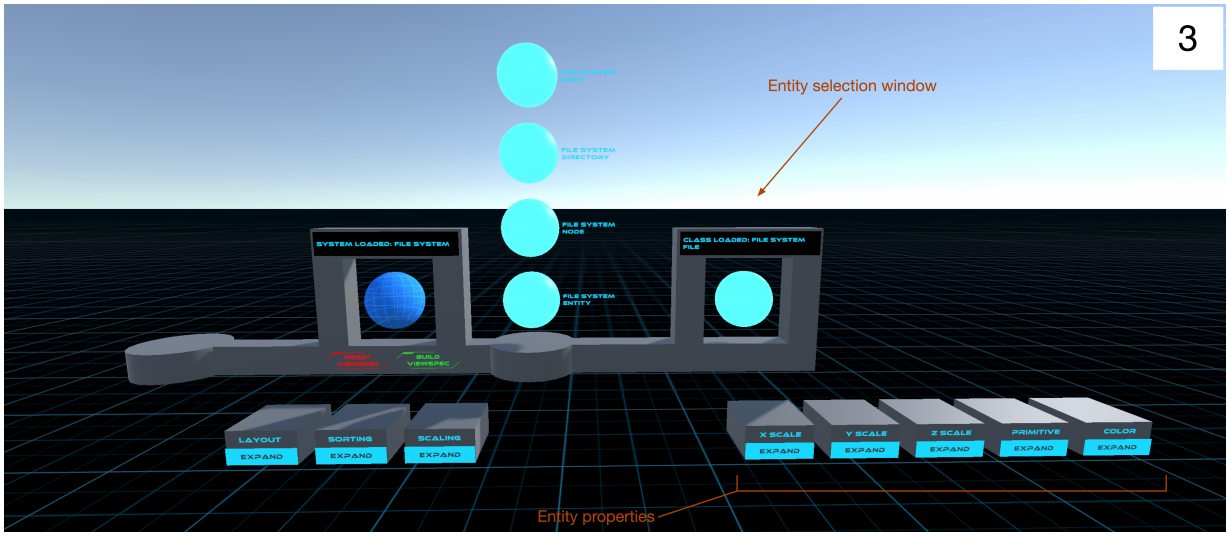

Figure 4.20: The user selects a domain entity by placing it in the entity selection window. Entity properties are shown accordingly.

4.6 Reflections

We started by comparing five of the most popular VR headsets to select the optimal one for testing our tool. Following this, we detailed the materials and technologies employed in the implementation of both the back-end and front-end components, as well as the tools used for profiling and debugging.

Next, we introduced the integrated tool resulting from the combination of IVAR-NI and VIRTEX. This tool enables users to customize the visualization of a domain and interact with the glyphs representing its entities, showcasing the potential of our approach in real-world applications.

- IVAR-NI leverages the 3D selection metaphor to provide a novel method for manipulating view specifications;
- VIRTEX employs the designed metaphors to facilitate exploration within a VR visualization.

The implementation prompted important reflection. We developed some components, like the HUD, early on, before we conceived the glyph mapper and the 3D selection metaphor. As a result, HUD interaction currently relies on traditional 2D interfaces displayed on VR panels. To align with our goal of creating a fully VR-native interface, we will need to redesign the HUD functionalities to fully harness the immersive and intuitive interaction capabilities that VR offers.

In the next chapter, we will demonstrate the tool's application through two case studies, showcasing its ability to visualize and interact with complex software systems. These case studies not only

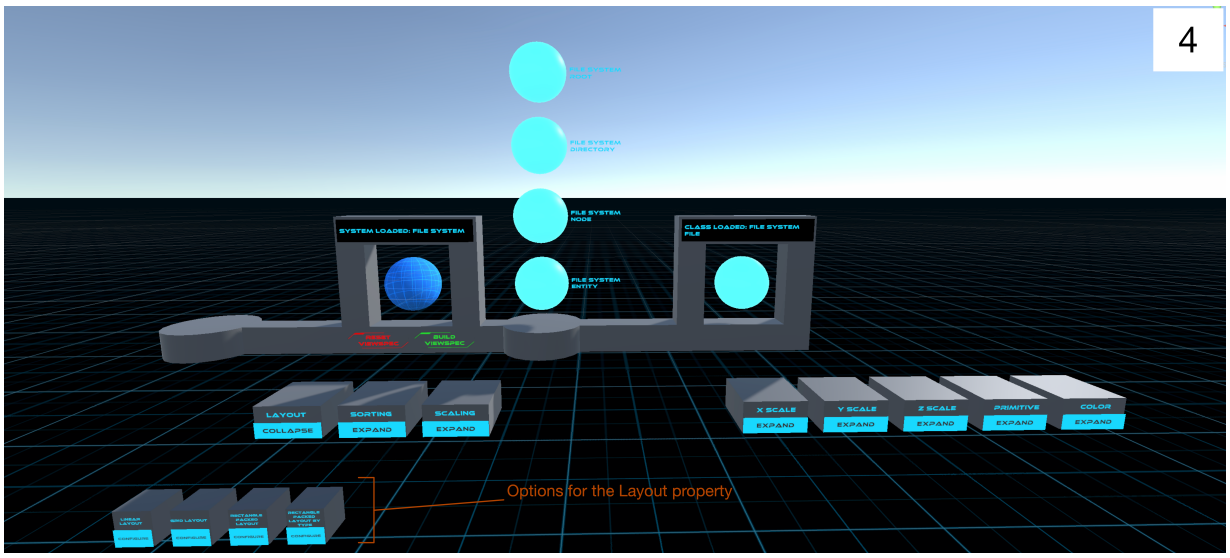

Figure 4.21: The expand/collapse button of a property toggles available options for that property.

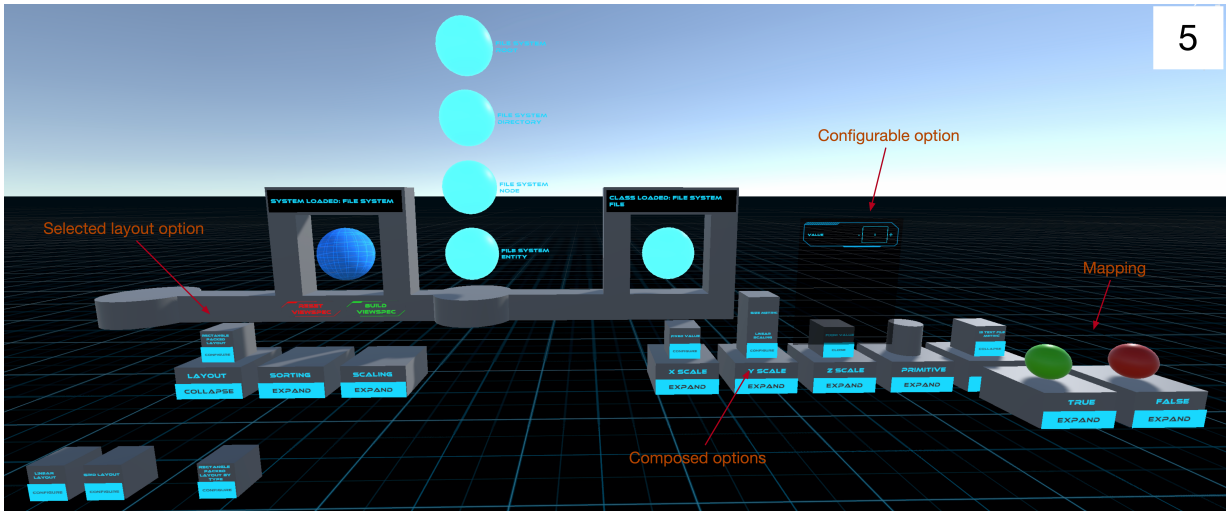

Figure 4.22: The user selects an option for a property by placing the glyph depicting the option on top of the platform representing the property.

internally validate the tools but also emphasize the practical advantages of using immersive VR for program comprehension, offering insights for future improvements.

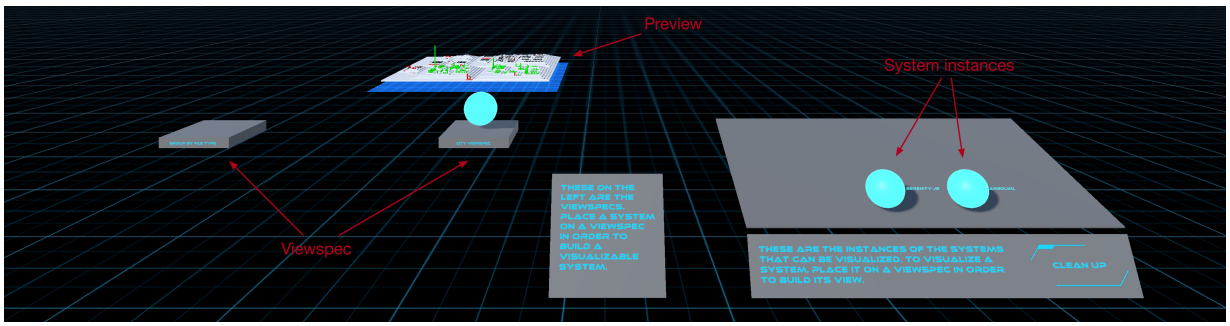

Figure 4.23: The system instances area, the viewspecs area and a preview of the visualization.

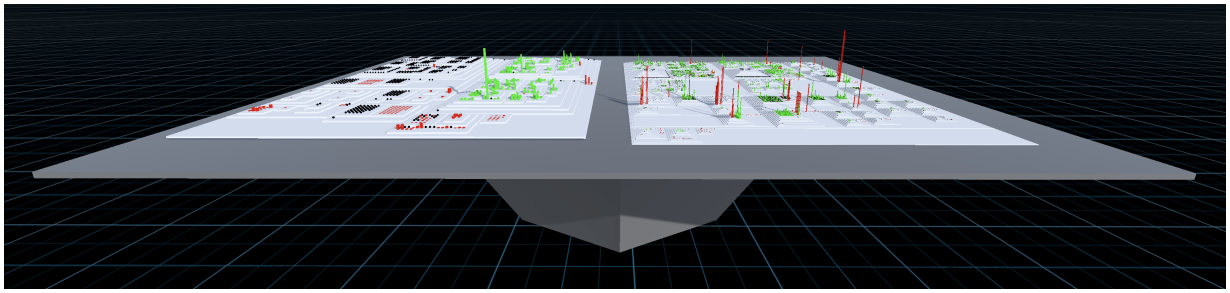

Figure 4.24: The planning table used to plan the visualization scene.

Chapter 5

Case Studies

In this chapter we present two case studies to show the capabilities of the tool. These case studies also act as an internal validation of the tool and help us figure out the direction to take by future work.

- In Section 5.1, we present a case study related to the file system model. In particular we show how a custom visualization can be used to spot large binary files in the file system;
- In Section 5.2, we present a case study related to the java project model. In particular we show how a custom visualization can be used to guide a developer in re-arranging java components.

5.1 File System Case Study

A binary file is any file that is not a human-readable text file. Unlike text files that consist of plain characters (ASCII or Unicode) that can be interpreted directly by humans, binary files store data in a format that is only interpretable by specific software or hardware. This could include images, videos, compiled programs, audio files, or any other non-text-based data. Binary files are encoded in a format of 0s and 1s that makes it difficult to track differences in a line-by-line comparison, unlike text files.

Git is a software used to tracks versions of files. When a text file is changed, Git stores the line-by-line difference (diff) between the current version and the previous version. This way, Git efficiently manages changes in text files by keeping only the modifications.

For binary files, Git cannot generate meaningful diffs in the same way it does with text files because binary files don't have lines of content that can be compared. Therefore, when a binary file is modified, Git treats it as a completely new version and stores the entire file again, not just the changes. This can lead to larger repositories and higher storage costs over time as binary files are modified.

We hence present a case study in which the custom visualization allows distinguishing between text and binary files, as well as spotting large binary files.

5.1.1 Spotting Large Binary Files

We now build a viewspec based on the city layout to allow a user spot large binary files. To make them stand out over text files, we differentiate them by color. We then map onto the height of the building representing binary files the Size metric that calculates the size of the files in MB. In this way, large binary files will be represented by taller buildings and can be immediately recognized.

Hence, we build the following mapping that results in the visualization shown in Figure 5.1:

- A directory is mapped onto white districts;
- A text file is mapped onto a black cube of side 1;
- A binary file is mapped onto a yellow 1-by-1 base parallelepiped. Its height is mapped to the Size metric.

The user immediately spots not only the binary files with the largest sizes, but also clusters of related files. In Figure 5.1 for example, the majority of the binary files are grouped into the same directory.

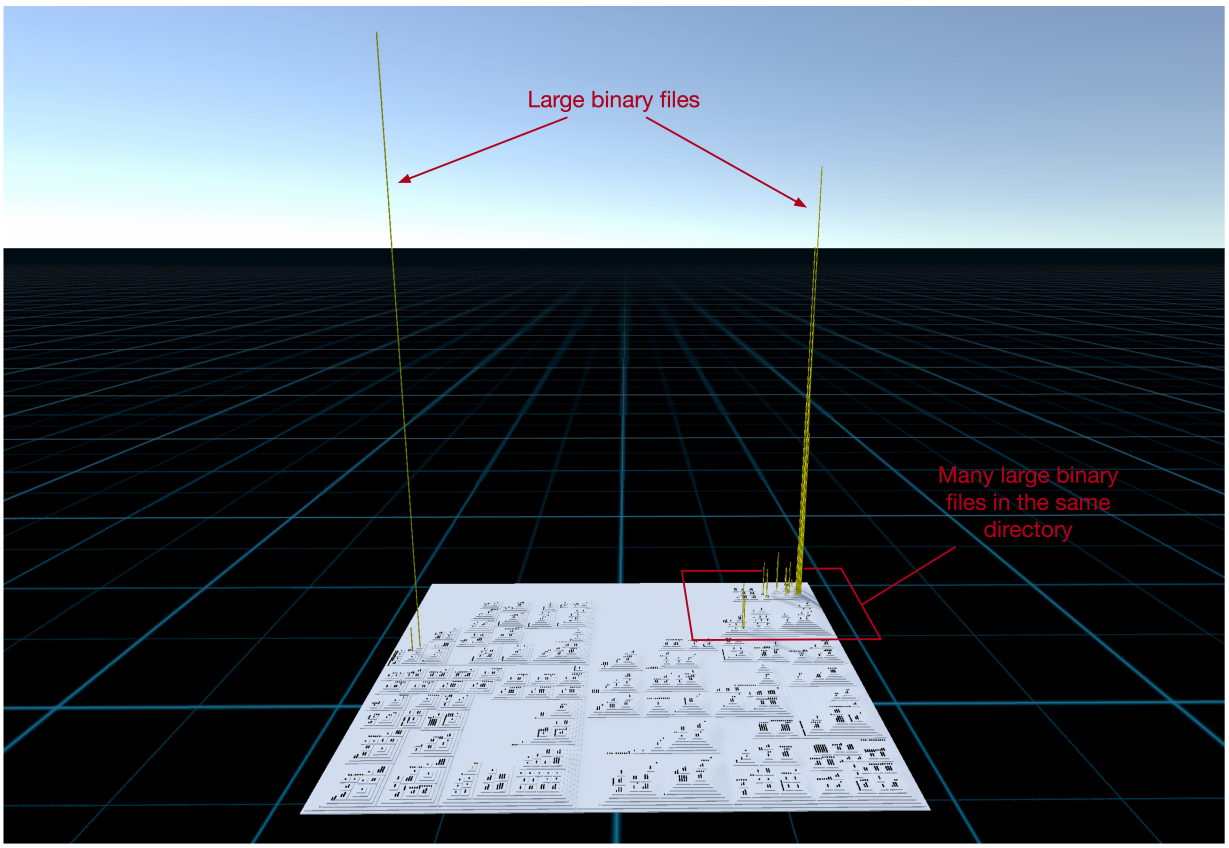

Figure 5.1: The JetUML repository visualized for spotting large binary files.

5.1.2 Thoughts on the File System Case Study

This case study shows how it is possible to use the tool to spot not only large binary files immediately, but also their context, visualizing if and how they are related to each other. From Figure 5.1 in fact, you notice that many tall buildings are located in the same district of the city, and thus that many large binary files are all located "close together".

In the case of a repository, this might indicate, for example, that the folders represented by that district contain the GUI icons (*i.e.,* images, usually binary files). In the next case study we present how the inspection tool can be used to collect insights about the objects of the visualized domain, allowing the user to further explore the insights that they grasp when visualizing a domain.

5.2 Java Project Case Study

Maintaining a high-quality codebase is essential for the maintainability and longevity of any software project. Over time, as features are added and requirements evolve, the initial code structure can become cluttered and challenging to manage. This is where the practices of *refactoring* comes into play, offering solutions to keep the code clean, efficient, and adaptable.

> *Refactoring is the process of restructuring existing code without changing its external behavior* [1].

This practice is crucial because it enhances the readability and understandability of the code. As developers works on a project, the code can accumulate complexity due to quick fixes, temporary solutions, and evolving coding styles. By taking the time to refactor, developers can simplify the code, making it easier for themselves and others to navigate. This clarity significantly reduces the cognitive load on developers, allowing them to focus on problem-solving rather than deciphering tangled code.

Furthermore, refactoring improves maintainability. A well-organized codebase allows for smoother updates and bug fixes, reducing the risk of introducing new issues while addressing existing ones.

When code is clean and modular, it becomes straightforward to implement new features or modify existing ones without the fear of disrupting the entire system. This aspect of refactoring is particularly important in today's development environments, where frequent iterations and updates are the norm.

In addition to enhancing readability and maintainability, refactoring plays a crucial role in managing technical debt. Over time, shortcuts taken to meet deadlines can accumulate, leading to a burden that hinders future development. By regularly refactoring, teams can proactively address these issues, keeping the codebase healthy and more manageable. This not only fosters a culture of quality within the team but also prevents potential pitfalls in future development cycles.

As projects evolve, the organization of components within the codebase may no longer be optimal. By rearranging code components (*e.g.,* packages, classes, methods), developers can enhance the separation of concerns within the application. Each component can take on a specific role, which simplifies understanding and reduces complexity. This modularity allows teams to work more efficiently, as they can focus on distinct parts of the system without stepping on each other's toes.

Moving components also supports scalability. As an application grows, its architecture needs to adapt to handle increased user demands and features. By reorganizing components, developers can optimize the system for scalability, ensuring that it can grow alongside its user base. This is particularly relevant in micro-services architectures, where decoupled components can be developed, deployed, and scaled independently. Moreover, the practice of moving components encourages reusability. When similar functionalities are consolidated into shared components, code duplication is minimized, which simplifies maintenance and streamlines future development efforts. This not only saves time but also reduces the risk of introducing inconsistencies across the codebase.

In this case study we present three refactoring actions and an additional tool to guide a developer execute them.

5.2.1 Move Class Refactoring

This refactoring task involves relocating a class from one package to another. This refactoring is typically done when the class has become more relevant to a different context or when it is no longer fitting well within its original package. Another reason could also be that the size of the package in which it is contained is too large. The primary motivations for moving a class include:

- **Improved Organization**: As codebases grow, some classes may need to be reorganized to reflect their usage better. For instance, if a class originally placed in a generic package is only used by a specific module, moving it to that module's package can enhance clarity;
- **Reducing Dependencies**: Moving a class to a more appropriate package can help minimize dependencies, making the system more modular and easier to manage;
- **Encouraging Cohesion**: Classes that have similar responsibilities or functionalities can be grouped together, promoting better cohesion within the system. This grouping makes it easier for developers to understand related components;

5.2.2 Move Field Refactoring

This refactoring task involves moving a field from one class to another. This refactoring is beneficial when the field is more closely related to another class, indicating a need for better encapsulation or improved object relationships. Key reasons for moving a field include:

- **Improving Encapsulation**: Moving a field to a more relevant class enhances encapsulation, ensuring data is managed in the appropriate context;
- **Reducing Coupling**: By relocating a field, you can decrease the dependencies between classes, promoting a more modular architecture. This separation helps prevent changes in one class from impacting others unnecessarily;
- **Enhancing Clarity**: When a field's purpose aligns more closely with a different class, moving it can clarify the roles of the classes involved, making the code easier to understand and maintain.

5.2.3 Move Method Refactoring

This refactoring task involves moving a method from one class to another. This refactoring is particularly useful when a method's functionality is more aligned with another class, indicating that it may not belong to its current owner. Reasons to perform a move method refactoring include:

- **Improving Responsibility Assignment**: If a method operates primarily on data from another class, moving it can better align responsibilities and enhance cohesion;
- **Reducing Complexity**: By transferring a method to a more appropriate class, you can simplify the original class and improve its readability. This also helps in reducing the size of the class, making it easier to understand;
- **Enhancing Reusability**: When a method is moved to a class where it is more applicable, it may become easier to reuse in different contexts, facilitating better design practices

5.2.4 Applying a Move Class Refactoring

In this specific case study, we show how it is possible to build a custom visualization that aids in the refactoring task of moving a class in the context of a java project (*i.e.,* the JetUML repository [2]). For this purpose, we note that the information that help in this task are:

- **The size of the packages**: If the package contains a large number of classes, it might be difficult for developers to find what they need, leading to confusion and errors. Moreover, it might be that the package contains classes that serve different purposes or functionalities, thus violating the Single Responsibility Principle;
- **The references to a class**: Many external references could imply that a class doesn't logically fit within its current package, thus reducing cohesion and increasing coupling.

Once we identify the necessary information, we then build a viewspec that displays the entire project in a layout that minimizes wasted space (*i.e.,* the city layout described in Sub-Section 3.1.1), so that all the components of the whole project can be visualized in a compact manner. We choose the containment relationship between components (*e.g.,* packages, classes, interfaces, members) for the nesting hierarchy. In this way, the lower-most platform represents the java project and the platforms above represent the packages and the classes or interfaces. The classes and interfaces are shown in a darker color to distinguish them from the packages while the members of the classes and interfaces are shown as buildings in the city layout. Their height is mapped to the number of external references, to aid spotting elements with many external references. Figures 5.2 and 5.3 show JetUML repository visualized via the viewspec just described.

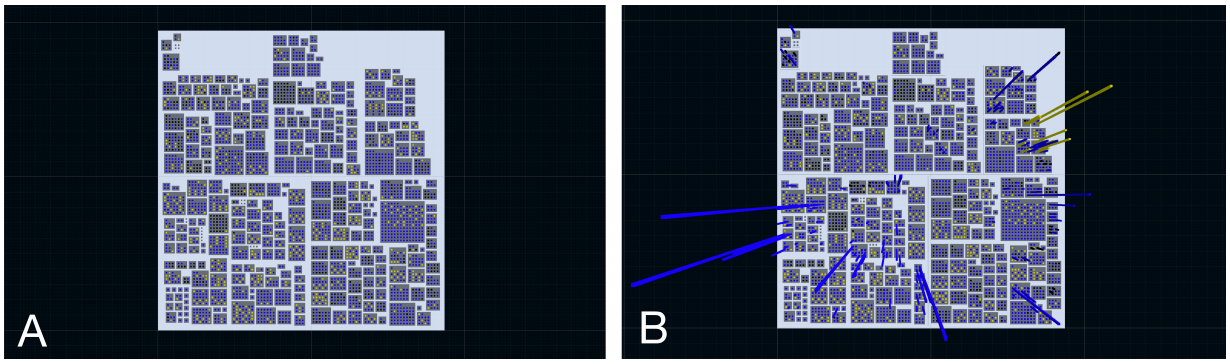

Figure 5.2: The JetUML repository visualized for aiding in the refactoring task of moving a class. Subfigure (A) shows a 2D representation of the repository from above. Subfigure (B) shows a 3D representation of the repository from above.

[2] See `https://github.com/prmr/JetUML`

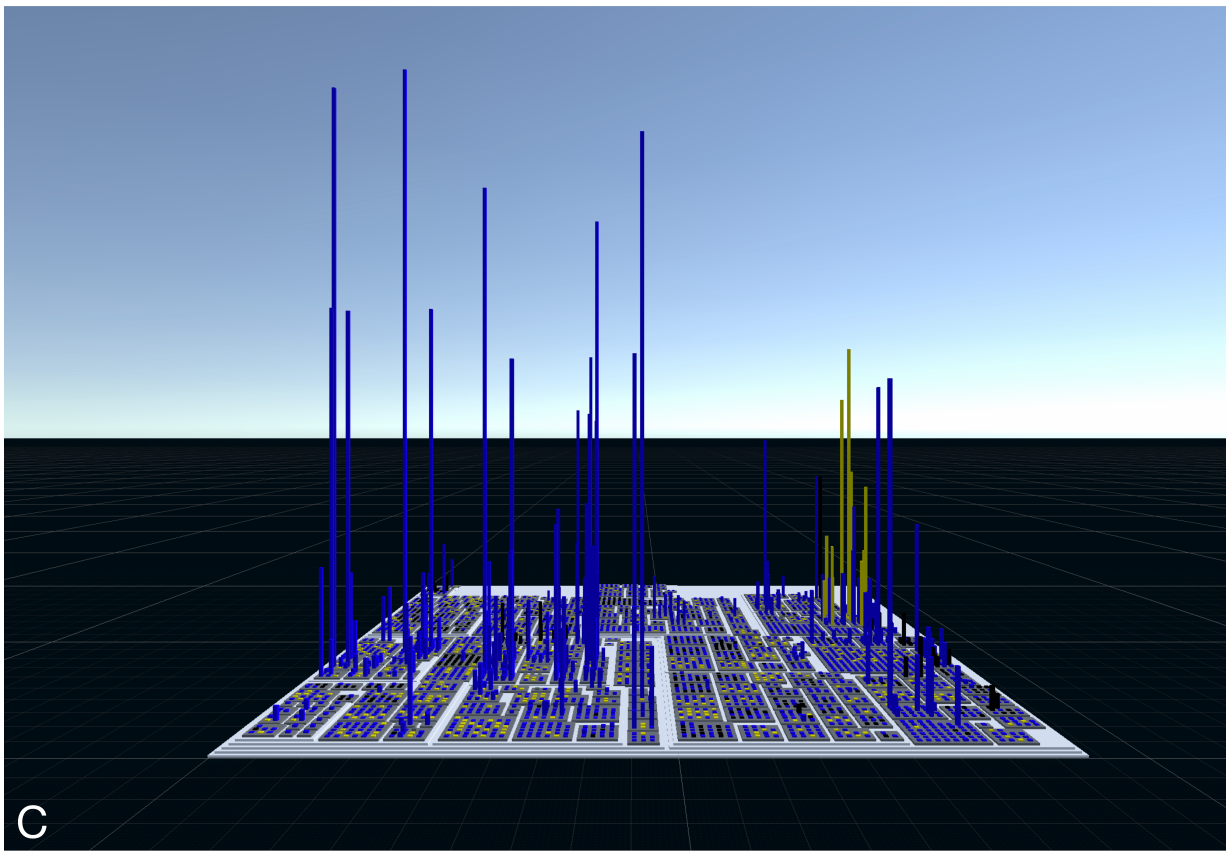

Figure 5.3: The JetUML repository visualized in 3D for aiding in the refactoring task of moving a class.

Once the user identifies the class to be moved, with the help of the buildings height and the inspection tool, as shown in Figure 5.4, they can grab the glyph representing that class and activate a specific command to dynamically change the visualization.

Using the trigger button on the controller, in fact, the user can hide all the components that have no relations with the grabbed one, and visualize the connections with the packages that reference the chosen class, as shown in Figure 5.5. This information can guide the user in choosing a new package for the selected class in order to minimize the number of external references.

5.2.5 A Model for Complex Interactions

The interaction presented in the java project case study is relevant because it consists of several simple actions chained in a single complex interaction. The user can in fact interact with an object either by placing the ray on it to highlight and possibly inspect it, or by grabbing it to move it in space. The object can also be activated with the trigger button to visualize additional information.

Moving an object also has a semantics: If the object is released in a position that does not fit, it will automatically return to its place, if it is released in a meaningful position, then it will be moved accordingly. In the case study just described, a class can be moved to another package and the move class refactoring is performed accordingly.

The analysis of such interaction led us to build a model for the interactions. In fact, it can be then represented with the Finite-State Automata (FSA) shown in Figure 5.6, as a composition of the following elementary interactions:

- **Hover**: The ray casted from the controller is placed on the object. The object is highlighted to give a feedback to the user. Also an haptic feedback is reproduced;
- **Grab**: The object is grabbed and can be moved in the virtual space;

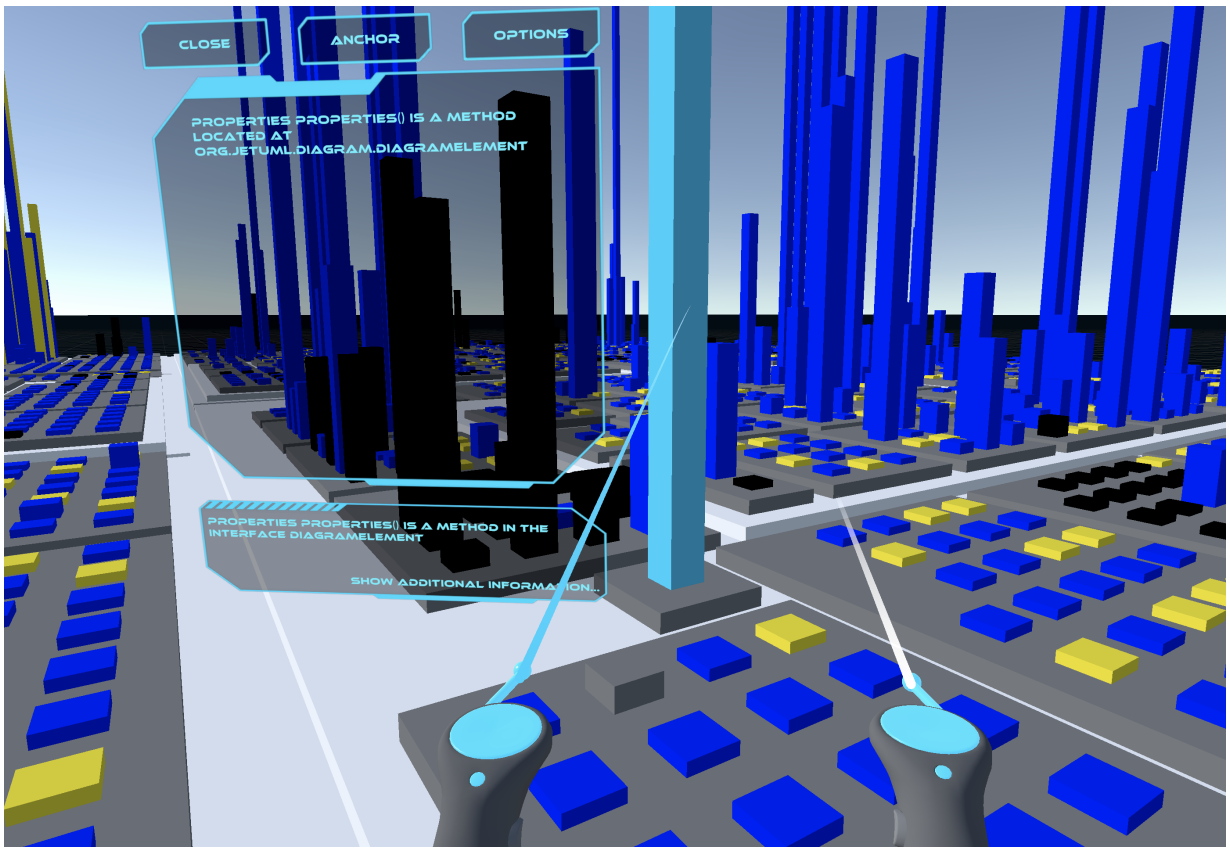

Figure 5.4: The user choose an object by looking at its height and inspecting it with the inspection tool.

- **Activate**: A domain action must be mapped on this interaction in order for this to take effect. In this case study, the action of showing the references is bound to this interaction;
- **Deactivate**: As for activate, an action must be mapped to this interaction. In this case study, the action of hiding the references is bound to this interaction;
- **Release**: The object is released in the current position. The position can also be related to a semantic. In this case study, a position is only valid if the class is placed on a package to which it can be moved, otherwise it is placed back to the original position. If an object is activated is also automatically deactivated;
- **Leave**: The ray casted from the controller is moved away from the object;
- **Inspect**: Some buttons on the controller can be used to inspect the hovered object. In particular, the A/X buttons show the tool-tip while the B/Y buttons show the advanced tool-tip.

Please note that in general the FSA changes depending on how a complex interaction is implemented, but there are some elementary actions that cannot be left out, as well as some limitations.

For instance, it is possible to choose to deactivate an object when it is released, as well as not to deactivate it. However, it is not possible to avoid hovering the object before starting any interaction when it comes to ray-cast interactions.

When it comes to direct interactions, on the other hand, it is not possible to hover an object because this interaction is only made through the ray casted by the controller. In the case of direct interactions, therefore, the initial state of the FSA would be the grab interaction and there would be no possibility of inspecting the object, as the inspection tool depends on the ray casted from the controller.

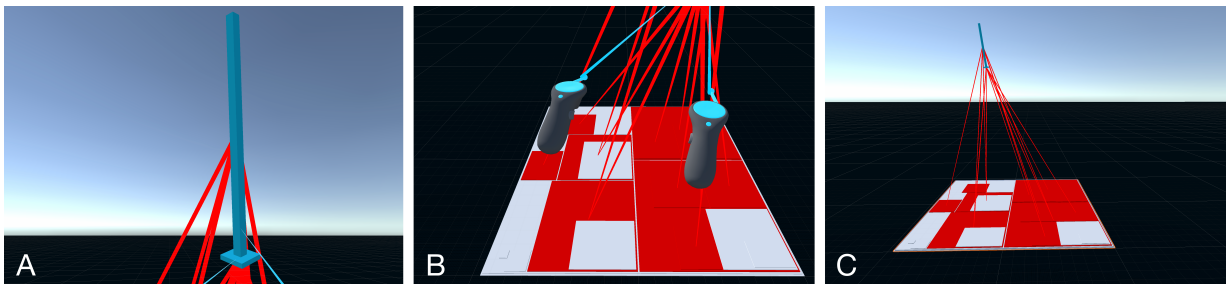

Figure 5.5: Subfigures (A) and (B) show the object and the references from the point of view of the user. Subfigure (C) shows the same from a distance.

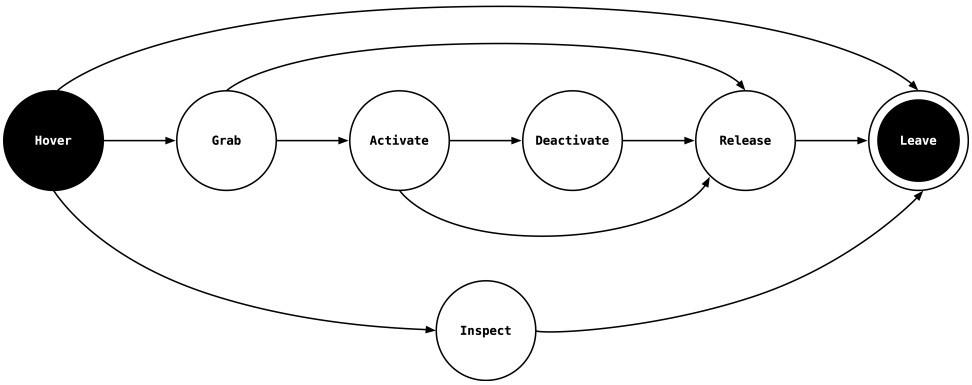

Figure 5.6: A finite-state automata representing the complex interaction implemented for the java project case study.

5.3 Reflections

In this chapter, we presented two case studies that demonstrate the practical applications of the developed tool in real-world scenarios. The first case study focused on exploring a file system, showing how the tool can facilitate efficient navigation and visualization of file structures. The second case study involved guiding a refactoring task within the context of a Java project, emphasizing the tool's capability to assist developers in managing a code-base.

Through the process of conducting these case studies, we identified a significant insight: complex interactions can be modeled as chains of elementary actions. By further refining our model of elementary actions and the overall interaction framework, we could accommodate even more complex interactions, ultimately leading to a more robust tool.

The internal validation of our tool through these case studies highlighted its effectiveness in improving both the visualization and interaction aspects of software development. The immersive capabilities of VR allow users to engage with software systems in a way that feels more intuitive and natural. This not only streamlines complex tasks but also reduces cognitive load, enabling developers to focus on the critical aspects of their work.

These findings pave the way for future developments. The successful application of IVAR-NI and VIRTEX in these case studies demonstrates the potential of VR to contribute to software visualization and interaction.

Chapter 6

Conclusions and Future Work

The rapid advancements in VR technologies, combined with the evolving needs of software developers, have opened new avenues for immersive software visualization. Traditional 2D methods, while effective in certain contexts, often struggle to represent the complexity and abstraction inherent in software systems. The integration of VR into software visualization offers a promising solution, enabling developers to interact with 3D representations of software in a more natural, intuitive manner.

In this thesis, we explored the history and theoretical foundations of both software visualization and VR. This groundwork has informed the development of IVaR-NI and VirtEx, two complementary tools designed to enhance visualization and interaction within VEs. By leveraging VR's immersive and interactive capabilities, these tools offer developers a richer, more intuitive way to explore and manipulate software systems.

The following sections provide a discussion of our findings and experiences during the development of IVaR-NI and VirtEx, followed by a detailed outline of future work aimed at refining and expanding the capabilities of these tools.

6.1 Discussion

We began by presenting an overview of the history of software visualization, tracing its development from early on-paper representations to the cutting-edge tools of today. This history underscores a fundamental need for visual aids to help developers understand the abstract and complex nature of software. The advent of 3D visualizations, in particular, has been transformative, allowing developers to create information-rich views that use visual properties such as size, shape, and color to represent domain-specific data.

In particular, the city layout visualization stands out among the others. It is a well-established metaphor that provides a compact and insightful way of visualizing a software system. Given its relevance, we extended it to a generic domain, allowing any entity of a given model to be represented as a building in the city visualization.

VR also has a rich history, particularly in simulation-based fields, due to its ability to foster immersion and interaction. Immersion is achieved when a VR system convincingly presents a VE with high visual quality and fast rendering. Poor visual fidelity or lag can disrupt this immersion and create discomfort. We reviewed the hardware properties critical for achieving quality and performance, providing a basis for evaluating VR systems. On the interaction front, cognitive science and human-computer interaction principles provide foundational guidelines. By leveraging affordance, feedback, and ergonomic design, developers can create intuitive, natural interactions in VR environments that mimic real-world behaviors.

Interaction in a VE can take two forms: interaction with 3D objects and interaction with GUIs. The former is intuitive, as users naturally apply their real-world knowledge to manipulate virtual objects. For example, picking up or throwing a virtual object follows real-world physics, making it easy for users to understand.

On the other hand, interacting with 2D GUIs within VR draws from users' familiarity with traditional 2D interfaces designed for screens. This fall short of the interaction capabilities that VR

provides, especially in terms of DoF. In addition, many existing visualization tools rely on on-screen GUIs for configuration, requiring users to frequently switch between the immersive 3D environment and the 2D interface. This context-switching interrupts the flow and reduces immersion.

We proposed a novel approach to address this issue: designing GUIs specifically for VR. Our novel 3D selection menu metaphor offers a more natural and immersive way of building user interfaces, leveraging the interaction capabilities of VR systems. Building on this metaphor, we developed IVaR-NI, a proof-of-concept that enables users to manipulate view specifications through VR-native interfaces directly within the VE. This eliminates the need to switch to external configuration interfaces, thereby preserving the immersive experience. The positive reception from the scientific community, as evidenced by feedback from VISSOFT, confirms the value of this approach.

We then developed VirtEx to complement IVaR-NI by providing a way to explore and interact with visualized software systems in VR. Together, these tools enable a rich, intuitive experience for developers working with complex software systems.

Our work also draws heavily on insights from cognitive science, particularly regarding the use of metaphors to reduce cognitive load in VR. The city layout metaphor, for example, helps users comprehend complex systems by associating abstract software elements with familiar real-world structures, such as buildings and streets. This not only aids in understanding but also enhances navigation and interaction in the VE.

We evaluated various VR headsets to determine the most suitable hardware for our tool and profiled the back-end and front-end components to ensure performance. Finally, we internally validated our tool through two case studies, which demonstrated the practical applications of IVaR-NI and VirtEx. The tools facilitated the exploration of a file system and the refactoring of a Java project. This validation showed how VR can improve both the visualization and interaction aspects of software development, making complex tasks more intuitive and less cumbersome.

6.2 Future work

While the integration of IVaR-NI and VirtEx has demonstrated the potential of immersive software visualization in VR, the tool remains a proof-of-concept with significant room for development. The following areas outline our planned future work.

6.2.1 Technical Improvements

There are several technical enhancements we plan to implement:

- **Integration as a Library**: We aim to refactor IVaR-NI as a reusable library, which could be easily integrated into other visualization projects. This would broaden its applicability and streamline its adoption across different domains;
- **Performance Optimization**: One of the major limitations of the current system is the slow scene construction during the layouting phase, which hinders real-time interaction. Optimizing this process is crucial to support smoother and more dynamic interactions. Additionally, enriching the model with more detailed data (*e.g.,* capturing method and constructor parameters, and other fine-grained elements) will enhance the depth of the visualized domain, providing richer interaction possibilities.

6.2.2 New Features

We also planned several new features to extend the capabilities of IVaR-NI and VirtEx:

- **Deeper Integration**: More seamless integration of IVaR-NI into VirtEx will enable users to configure viewspecs directly from within the visualization, making real-time adjustments without leaving the exploration environment. This would allow for a continuous configuration of visualizations, enhancing the user experience;

- **Advanced Interaction Techniques**: We plan to experiment with more natural interaction techniques, such as hand-tracking to allow direct manipulation of virtual objects, and voice recognition for text input. These techniques would make the interaction more intuitive and reduce the reliance on traditional controllers;
- **HUD Redesign**: We plan to re-design the HUD to leverage the interaction capabilities of VR technology, further moving away from traditional 2D interfaces. The mini-map, for example, could be re-designed as a miniature of the scene that user can enlarge and shrink to zoom in and out. This interaction could be implemented as a two-handed interaction, further extending the set of elementary actions that the user can perform;
- **Spatial Audio Integration**: Implementing spatial audio could be an effective way to guide users' attention to off-view elements, improving the overall immersion and user awareness within the VE;
- **Enhanced Refactoring Capabilities**: Expanding the system's ability to support more refactorings, particularly in the context of software engineering tasks, will enhance the tool's utility in practical use cases.

6.2.3 User Study Evaluation

We planned a comprehensive user study to evaluate the usability and effectiveness of the tool:

- **Pilot Study**: Initially, a pilot study will help refine the interface and interaction techniques. This will provide preliminary insights and allow us to address any technical or design issues before larger-scale testing;
- **Larger User Study**: A more extensive user study will follow, including surveys and interviews to gather qualitative feedback from participants. This will help us assess user satisfaction, ease of use, and the potential for real-world adoption. Insights gained from this study will inform the iterative improvement of both IVaR-NI and VirtEx.

6.3 Closing Words

This thesis focused on improving interaction within a VE for program comprehension. By moving beyond traditional 2D interfaces, IVaR-NI and VirtEx show how VR can offer more natural, intuitive ways to explore and manipulate complex software systems. The seamless, immersive interaction made possible by VR reduces the cognitive load and enhances user engagement.

As we look to the future, refining interaction metaphors will further improve the user experience, keeping interaction at the core of creating more intuitive and powerful visualization tools.

Bibliography

[1] B. L. Stuart, "Programming the ENIAC [Scanning our Past]," *Proceedings of the IEEE*, vol. 106, no. 9, pp. 1760–1770, 2018.

[2] H. H. Goldstine and J. Von Neumann, "Planning and Coding of Problems for an Electronic Computing Instrument," 1947.

[3] F. E. Allen, "Control Flow Analysis," *ACM Sigplan Notices*, vol. 5, no. 7, pp. 1–19, 1970.

[4] P. P.-S. Chen, "The Entity-Relationship Model—Toward a Unified View of Data," *ACM Transactions on Database Systems (TODS)*, vol. 1, no. 1, pp. 9–36, 1976.

[5] D. Stephan, *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer Science & Business Media, 2007.

[6] R. Wettel and M. Lanza, "CodeCity : 3D Visualization of Large-Scale Software," in *Companion of the 30th International Conference on Software Engineering*, 2008, pp. 921–922.

[7] G. C. Burdea and P. Coiffet, *Virtual Reality Technology*. John Wiley & Sons, 2024.

[8] S. Feist, "A-BIM: Algorithmic-Based Building Information Modelling," 2016.

[9] M. Mori, K. F. MacDorman, and N. Kageki, "The Uncanny Valley [From the Field]," *IEEE Robotics & Automation Magazine*, vol. 19, no. 2, pp. 98–100, 2012.

[10] F. Steinicke, T. Ropinski, and K. Hinrichs, "Object Selection in Virtual Environments Using an Improved Virtual Pointer Metaphor," in *Computer Vision and Graphics: International Conference, ICCVG 2004*. Springer International Publishing, 2006, pp. 320–326.

[11] A. Vakunov, C.-L. Chang, F. Zhang, G. Sung, M. Grundmann, and V. Bazarevsky, "MediaPipe Hands: On-device Real-time Hand Tracking," in *Workshop on Computer Vision for AR/VR*, 2020, p. 5.

[12] S. Rothe, K. Tran, and H. Hußmann, "Dynamic Subtitles in Cinematic Virtual Reality," in *Proceedings of the 2018 ACM International Conference on Interactive Experiences for TV and Online Video*, vol. 20, no. 4. ACM, 2018, pp. 209–214.

[13] B. P. Frederik, "No Silver Bullet: Essence and Accidents of Software Engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987.

[14] A. Hoff, C. Seidl, and M. Lanza, "Uniquifying Architecture Visualization through Variable 3D Model Generation," in *Proceedings of VaMoS 2023 (International Working Conference on Variability Modelling of Software-Intensive Systems)*. ACM, 2023, pp. 77–81.

[15] D. F. Jerding, J. T. Stasko, and T. Ball, "Visualizing Interactions in Program Executions," in *Proceedings of International Conference on Software Engineering*, 1997, pp. 360–370.

[16] T. D. LaToza and B. A. Myers, "Visualizing Call Graphs," in *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 117–124.

[17] C. Armenti and M. Lanza, "Using Interactive Animations to Analyze Fine-grained Software Evolution," in *Proceedings of VISSOFT 2024 (Working Conference on Software Visualization)*. IEEE, 2024, pp. 36–47.

[18] G. Occhipinti, C. Nagy, R. Minelli, and M. Lanza, "SYN: Ultra-Scale Software Evolution Comprehension," in *Proceedings of ICPC 2023 (International Conference on Program Comprehension)*. IEEE, 2023, pp. 69–73.

[19] D. Moreno-Lumbreras, J. M. González-Barahona, and M. Lanza, "Understanding the NPM Dependencies Ecosystem of a Project Using Virtual Reality," in *Proceedings of VISSOFT 2023 (Working Conference on Software Visualization)*. IEEE, 2023, pp. 84–94.

[20] 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 *Proceedings of VISSOFT 2018 (Working Conference on Software Visualization)*. IEEE, 2018, pp. 112–116.

[21] M. Raglianti, C. Nagy, R. Minelli, and M. Lanza, "DiscOrDance: Visualizing Software Developers Communities on Discord," in *Proceedings of ICSME 2022 (International Conference on Software Maintenance and Evolution)*. IEEE, 2022, pp. 474–478.

[22] M. Raglianti, R. Minelli, C. Nagy, and M. Lanza, "Visualizing Discord Servers," in *Proceedings of VISSOFT 2021 (Working Conference on Software Visualization)*. IEEE, 2021, pp. 150–154.

[23] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides, "Visualizing the Behavior of Object-Oriented Systems," in *Proceedings of the Conference on Object-oriented Programming Systems, Languages and Applications*. ACM, 1993, pp. 326–337. [Online]. Available: https://doi.org/10.1145/165854.165919

[24] R. T. Alfredo and C. R. Marcelo, "An Overview of 3D Software Visualization," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 1, pp. 87–105, 2009.

[25] R. Wettel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," in *Proceedings of ICSE 2011 (International Conference on Software Engineering)*., 2011, pp. 551–560.

[26] R. Koschke, "Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey," *Journal of Software Maintenance*, vol. 15, no. 2, pp. 87–109, 2003.

[27] A. Von Mayrhauser and A. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[28] M. H. Weik, "The ENIAC Story," *Ordnance*, vol. 45, no. 244, pp. 571–575, 1961.

[29] G. Bonfante, M. Kaczmarek, and J.-Y. Marion, "Control Flow Graphs as Malware Signatures," in *International Workshop on the Theory of Computer Viruses*, 2007. [Online]. Available: https://inria.hal.science/inria-00176235

[30] A. V. Phan, M. Le Nguyen, and L. T. Bui, "Convolutional Neural Networks Over Control Flow Graphs for Software Defect Prediction," in *Proceeding of ICTAI 2017 (International Conference on Tools with Artificial Intelligence)*. IEEE, 2017, pp. 45–52.

[31] B. J. Jansen, "The Graphical User Interface," *ACM SIGCHI Bulletin*, 1998.

[32] J. T. Stasko, "Tango: A Framework and System for Algorithm Animation," *ACM SIGCHI Bulletin*, 1990.

[33] S. G. Eick, J. L. Steffen, and E. E. Sumner, "SeeSoft-A Tool for Visualizing Line Oriented Software Statistics," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 957–968, 1992.

[34] A. Watson, "Visual Modelling: Past, Present and Future," *White Paper UML Resource Page*, vol. 28, 2008. [Online]. Available: https://api.semanticscholar.org/CorpusID:7908366

[35] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language," *Unix Review*, vol. 14, no. 13, p. 5, 1996.

[36] M.-A. Storey, C. Best, and J. Michand, "SHriMP Views: An Interactive Environment for Exploring Java Programs," in *Proceedings of IWPC 2001 (International Workshop on Program Comprehension)*. IEEE, 2001, pp. 111–112.

[37] M. Lanza, "CodeCrawler—Lessons Learned in Building a Software Visualization Tool," in *Proceedings of CSMR 2003 (European Conference on Software Maintenance and Reengineering)*. IEEE, 2003, pp. 409–418.

[38] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. Van Deursen, and J. J. Van Wijk, "Execution Trace Analysis Through Massive Sequence and Circular Bundle Views," *Journal of Systems and Software*, vol. 81, no. 12, pp. 2252–2268, 2008.

[39] A. H. Caudwell, "Gource: Visualizing Software Version Control History," in *Proceedings of International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*. ACM, 2010, pp. 73–74.

[40] P. Chatterjee, K. Damevski, L. Pollock, V. Augustine, and N. A. Kraft, "Exploratory Study of Slack q&a Chats as a Mining Source for Software Engineering Tools," in *Proceedings of MSR 2019 (International Conference on Mining Software Repositories)*. IEEE / ACM, 2019, pp. 490–501.

[41] D. Moreno-Lumbreras, G. Robles, D. Izquierdo-Cortázar, and J. M. González-Barahona, "Software Development Metrics: To VR or Not to VR," *Empirical Software Engineering*, vol. 29, no. 2, p. 42, 2024.

[42] R. Oberhauser and C. Lecon, "Virtual Reality Flythrough of Program Code Structures," in *Proceedings of VRIC 2017 (Virtual Reality International Conference)*. ACM, 2017, pp. 1–4.

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

[44] J. Vincur, P. Navrat, and I. Polasek, "VR City: Software Analysis in Virtual Reality Environment," in *Proceedings of QRS-C 2017 (International Conference on Software Quality, Reliability and Security Companion)*. IEEE, 2017, pp. 509–516.

[45] A. Hori, M. Kawakami, and M. Ichii, "CodeHouse: VR Code Visualization Tool," in *Proceedings of VISSOFT 2019 (Working Conference on Software Visualization)*. IEEE, 2019, pp. 83–87.

[46] S. Romano, N. Capece, U. Erra, G. Scanniello, and M. Lanza, "On the Use of Virtual Reality in Software Visualization: The Case of the City Metaphor," *Information and Software Technology*, vol. 114, pp. 92–106, 2019.

[47] M. Inkarbekov, R. Monahan, and B. A. Pearlmutter, "Visualization of AI Systems in Virtual Reality: A Comprehensive Review," *arXiv preprint*, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2306.15545

[48] A. Hoff, C. Seidl, and M. Lanza, "Immersive Software Archaeology: Exploring Software Architecture and Design in Virtual Reality," in *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2024, pp. 47–51.

[49] H. Morton, "Stereoscopic-Television Apparatus for Individual Use," 1960, Patent US2955156A. [Online]. Available: https://patents.google.com/patent/US2955156A

[50] ——, "Sensorama Simulator," 1962, Patent US3050870A. [Online]. Available: https://patents.google.com/patent/US3050870A

[51] Y. A. Boas, "Overview of Virtual Reality Technologies," in *Interactive Multimedia Conference*, vol. 2013, 2013, pp. 1–6.

[52] P.-A. Heng, C. Chun-Yiu, W. Tien-Tsin, X. Yangsheng, n. C. Yim-Pa, C. Kai-Ming, and T. Shiu-Kit, "A Virtual-Reality Training System for Knee Arthroscopic Surgery," *IEEE Transactions on Information Technology in Biomedicine*, vol. 8, no. 2, pp. 217–227, 2004.

[53] Y. M. Khalifa, D. Bogorad, V. Gibson, J. Peifer, and J. Nussbaum, "Virtual Reality in Ophthalmology Training," *Survey of Ophthalmology*, vol. 51, no. 3, pp. 259–273, 2006.

[54] J. Whyte, *Virtual Reality and the Built Environment.* Routledge, 2007.

[55] F. Mantovani and G. Castelnuovo, "Sense of Presence in Virtual Training: Enhancing Skills Acquisition and Transfer of Knowledge through Learning Experience in Virtual Environments," *Being There: Concepts, Effects and Measurement of User Presence in Synthetic Environments*, pp. 167–182, 2003.

[56] B. O. Rothbaum and A. C. Schwartz, "Exposure Therapy for Post-Traumatic Stress Disorder," *American Journal of Psychotherapy*, vol. 56, no. 1, pp. 59–75, 2002.

[57] E. B. Foa, "Prolonged Exposure Therapy: Past, Present, and Future," *Depression and Anxiety*, vol. 28, no. 12, pp. 1043–1047, 2011.

[58] A. Hoff, M. Lungu, C. Seidl, and M. Lanza, "Collaborative Software Exploration with Multimedia Note Taking in Virtual Reality," in *Proceedings of ICPC 2024 (International Conference on Program Comprehension).* ACM, 2024, pp. 346–357.

[59] A. Hoff, L. Gerling, and C. Seidl, "Utilizing Software Architecture Recovery to Explore Large-Scale Software Systems in Virtual Reality," in *Proceedings of VISSOFT 2022 (Working Conference on Software Visualization).* IEEE, 2022, pp. 119–130.

[60] D. Todorovic, "Gestalt Principles," *Scholarpedia*, vol. 3, no. 12, p. 5345, 2008.

[61] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction – Third Edition.* Prentice Hall, 2004.

[62] J. H. Steffen, J. E. Gaskin, T. O. Meservy, J. L. Jenkins, and I. Wolman, "Framework of Affordances for Virtual Reality and Augmented Reality," *Journal of Management Information Systems*, vol. 36, no. 3, pp. 683–729, 2019.

[63] D.-H. Shin, "The Role of Affordance in the Experience of Virtual Reality Learning: Technological and Affective Affordances in Virtual Reality," *Telematics and Informatics*, vol. 34, no. 8, pp. 1826–1836, 2017.

[64] Y. Rogers, "New Theoretical Approaches for HCI," *Annual Review of Information Science and Technology*, vol. 38, no. 1, pp. 87–143, 2004.

[65] J. Kreimeier, S. Hammer, D. Friedmann, P. Karg, C. Bühner, L. Bankel, and T. Götzelmann, "Evaluation of Different Types of Haptic Feedback Influencing the Task-Based Presence and Performance in Virtual Reality," in *Proceedings of International Conference on Pervasive Technologies Related to Assistive Environments.* ACM, 2019, pp. 289–298.

[66] C. Boletsis, J. E. Cedergren, and S. Kongsvik, "HCI Research in Virtual Reality: A Discussion of Problem-Solving," in *Proceeding of IHCI 2017 (International Conference on Interfaces and Human Computer Interaction).* SINTEF, 2017, pp. 263–267.

[67] J. Dudley, L. Yin, V. Garaj, and P. O. Kristensson, "Inclusive Immersion: A Review of Efforts to Improve Accessibility in Virtual Reality, Augmented Reality and the Metaverse," *Virtual Reality*, vol. 27, no. 4, pp. 2989–3020, 2023.

[68] D. M. Cook, D. Dissanayake, and K. Kaur, "Virtual Reality and Older Hands: Dexterity and Accessibility in Hand-Held VR Control," in *International Conference in Cooperation, HCI and UX*. ACM, 2019, pp. 147–151.

[69] I. E. Sutherland, "Sketch Pad a Man-Machine Graphical Communication System," in *Proceedings of the SHARE Design Automation Workshop*. ACM, 1964, p. 6.329–6.346.

[70] M. Slater, "A Note on Presence Terminology," *Presence Connect*, vol. 3, no. 3, pp. 1–5, 2003.

[71] D. A. Bowman and R. P. McMahan, "Virtual Reality: How Much Immersion Is Enough?" *Computer*, vol. 40, no. 7, pp. 36–43, 2007.

[72] M. I. Berkman and E. Akan, "Presence and Immersion in Virtual Reality," in *Encyclopedia of Computer Graphics and Games*. Springer, 2024, pp. 1461–1470.

[73] R. P. McMahan, C. Lai, and S. K. Pal, "Interaction Fidelity: The Uncanny Valley of Virtual Reality Interactions," in *Proceeding of the International Conference on Virtual, Augmented and Mixed Reality*. Springer, 2016, pp. 59–70.

[74] T. B. Sheridan, "Interaction, Imagination and Immersion Some Research Needs," in *Proceedings of the Symposium on Virtual Reality Software and Technology*. ACM, 2000, pp. 1–7.

[75] E. D. Mekler and K. Hornbæk, "A Framework for the Experience of Meaning in Human-Computer Interaction," in *Proceedings of CHI 2019 (Conference on Human Factors in Computing Systems)*. ACM, 2019, pp. 1–15.

[76] M. A. O'Brien, W. A. Rogers, and A. D. Fisk, "Developing a Framework for Intuitive Human-Computer Interaction," in *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 52, no. 20. SAGE Publications, 2008, pp. 1645–1649.

[77] D. Vyas, C. M. Chisalita, and G. C. Van Der Veer, "Affordance in Interaction," in *Proceedings of the Eurpoean Conference on Cognitive Ergonomics: Trust and Control in Complex Socio-Technical Systems*. ACM, 2006, pp. 92–99.

[78] S. Ghosh, C. S. Shruthi, H. Bansal, and A. Sethia, "What Is User's Perception of Naturalness? An Exploration of Natural User Experience," in *Proceeding of INTERACT 2017 (Human-Computer Interaction)*. Springer, 2017, pp. 224–242.

[79] J. J. LaViola Jr., E. Kruijff, R. P. McMahan, D. A. Bowman, and I. Poupyrev, *3D User Interfaces: Theory and Practice*. Addison-Wesley Professional, 2017.

[80] T. Luong, Y. F. Cheng, M. Möbus, A. Fender, and C. Holz, "Controllers or Bare Hands? A Controlled Evaluation of Input Techniques on Interaction Performance and Exertion in Virtual Reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 29, no. 11, pp. 4633–4643, 2023.

[81] I. B. Adhanom, P. MacNeilage, and E. Folmer, "Eye Tracking in Virtual Reality: A Broad Review of Applications and Challenges," *Virtual Reality*, vol. 27, no. 2, pp. 1481–1505, 2023.

[82] Y. Wang, G. Zhai, S. Chen, X. Min, Z. Gao, and X. Song, "Assessment of Eye Fatigue Caused by Head-Mounted Displays Using Eye-Tracking," *Biomedical Engineering Online*, vol. 18, pp. 111–130, 2019.

[83] J. Dudley, H. Benko, D. Wigdor, and P. O. Kristensson, "Performance Envelopes of Virtual Keyboard Text Input Strategies in Virtual Reality," in *Proceeding of ISMAR 2019 (International Symposium on Mixed and Augmented Reality)*. ACM, 2019, pp. 289–300.

[84] K. Zibrek, S. Martin, and R. McDonnell, "Is Photorealism Important for Perception of Expressive Virtual Humans in Virtual Reality?" *ACM Transactions on Applied Perception (TAP)*, vol. 16, no. 3, pp. 1–19, 2019.

[85] T. Van Gemert, N. C. Nilsson, T. Hirzle, and J. Bergström, "Sicknificant Steps: A Systematic Review and Meta-analysis of VR Sickness in Walking-based Locomotion for Virtual Reality," in *Proceedings of CHI 2024 (Conference on Human Factors in Computing Systems)*. Association for Computing Machinery, 2024, pp. 1–36. [Online]. Available: https://doi.org/10.1145/3613904.3641974

[86] J. Wang, R. Shi, W. Zheng, W. Xie, D. Kao, and H.-N. Liang, "Effect of Frame Rate on User Experience, Performance, and Simulator Sickness in Virtual Reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 29, no. 5, pp. 2478–2488, 2023.

[87] H. K. Kim, J. Park, Y. Choi, and M. Choe, "Virtual Reality Sickness Questionnaire (VRSQ): Motion Sickness Measurement Index in a Virtual Reality Environment," *Applied Ergonomics*, vol. 69, pp. 66–73, 2018.

[88] E. Bozgeyikli, A. Raij, S. Katkoori, and R. Dubey, "Point & Teleport Locomotion Technique for Virtual Reality," in *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*, 2016, pp. 205–216.

[89] M. Giannaccari, M. Raglianti, and M. Lanza, "Manipulating VR-native User Interfaces for Software Visualization Customization," in *Proceedings of VISSOFT 2024 (Working Conference on Software Visualization)*. IEEE, 2024, p. in press.

[90] A. Yeo, B. W. Kwok, A. Joshna, K. Chen, and J. S. Lee, "Entering the Next Dimension: A Review of 3D User Interfaces for Virtual Reality," *Electronics*, vol. 13, no. 3, 2024.

[91] F. Aslam and R. Zhao, "Voice-Augmented Virtual Reality Interface for Serious Games," in *2024 IEEE Conference on Games (CoG)*. IEEE, 2024, pp. 1–8.

[92] A. C. P. de França and M. M. Soares, "Metaphors and Embodiment in Virtual Reality Systems," in *Proceedings of the International Conference on Design, User Experience, and Usability: Technological Contexts*. Springer, 2016, pp. 278–286.

[93] K. Kilteni, R. Groten, and M. Slater, "The Sense of Embodiment in Virtual Reality," *Presence: Teleoperators and Virtual Environments*, vol. 21, no. 4, pp. 373–387, 2012.

[94] I. Herman, G. Melançon, and M. S. Marshall, "Graph Visualization and Navigation in Information Visualization: A Survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 24–43, 2000.

[95] B. Shneiderman, "The Eyes Have it: A Task by Data Type Taxonomy for Information Visualizations," in *The Craft of Information Visualization*. Elsevier, 2003, pp. 364–371.

[96] D. Bennett, O. Metatla, A. Roudaut, and E. Mekler, "How Does HCI Understand Human Autonomy and Agency?" ACM, 2023. [Online]. Available: https://arxiv.org/abs/2301.12490

[97] G. Wilson, M. McGill, M. Jamieson, J. R. Williamson, and S. A. Brewster, "Object Manipulation in Virtual Reality Under Increasing Levels of Translational Gain," in *Proceedings of CHI 2018 (Conference on Human Factors in Computing Systems)*. ACM, 2018, pp. 1–13.

[98] D. Marr, "Visual Information Processing: The Structure and Creation of Visual Representations," *Philosophical Transactions of the Royal Society of London. B, Biological Sciences*, vol. 290, no. 1038, pp. 199–218, 1980.

[99] J. J. Van Wijk, "The Value of Visualization," in *Proceedings of VIS 2005 (Visualization)*. IEEE, 2005, pp. 79–86.

[100] K. G. Seeber, "Cognitive Load," *Routledge Encyclopedia of Interpreting Studies*, vol. 60, pp. 82–84, 2015.

[101] F. H. Post, G. Nielson, and G.-P. Bonneau, "Data Visualization: The State of the Art," 2002.

[102] M.-A. Storey, F. D. Fracchia, and H. A. Müller, "Cognitive Design Elements to Support the Construction of a Mental Model During Software Exploration," *Journal of Systems and Software*, vol. 44, no. 3, pp. 171–185, 1999.

[103] D. Gentner and A. L. Stevens, *Mental Models*. Psychology Press, 2014.

[104] N. Fenton and J. Bieman, *Software Metrics: A Rigorous and Practical Approach*. CRC press, 2014.

[105] T. Klemola, "Software Comprehension: Theory and Metrics," Ph.D. dissertation, Concordia University, 1998.