A Multi-Projector Software Visualization for Collaborative Program Comprehension

Heiko Tobias Helmut Bielfeldt

Bachelor's Thesis
September 29, 2023

Software Engineering Group
Department of Computer Science
Kiel University

Advised by
Prof. Dr. Wilhelm Hasselbring
Additional Advisor, M.Sc. Malte Hansen

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

29.09.2023 H. Bielfeldt

Abstract

Software development is complex, requiring tools that facilitate both program comprehension and collaboration. As software architectures grow in size, developers require environments that not only aid in understanding these large architectures but also support collaborative possibilities to use collective knowledge. ExplorViz identifies as a solution, structuring information through software landscape models and providing collaborative features. While ExplorViz offers solutions, integrating it into an immersive environment like ARENA2 can further enhance its capabilities. By extending the display of one single software landscape to the size of five, it allows for the original software landscape to be presented in an expansive, planetary view.

This thesis explores the integration of ExplorViz into ARENA2 through the development of the Synchronization Feature. To achieve this, we integrate our feature into the existing microservice architecture, reusing pre-existing components while introducing new ones. The Frontend of ExplorViz serves as the visualization's base, whereas the Collaboration Service acts as a communication channel between clients. By reusing the infrastructure of the Spectate Feature, we can manipulate the spectating process for the Synchronization Feature without affecting the original feature. Therefore, a control instance, named Main, in ARENA2 establishes the visualization's origin from which we copy. The five projections that illuminate the inside of ARENA2's dome are represented by virtual cameras. These cameras store the graphical information from Main. Through graphical manipulation of these five identical images, we produce five distinct projections, each presenting only a fifth of the original software landscape. In conclusion, the Synchronization Feature we aim to implement creates a unified, immersive visualization on the surface of the ARENA2 dome by assembling these five projections.

Our implementation concludes in an evaluation involving pairs of participants to provide insights into the ExplorViz implementation within ARENA2. The findings underline the thesis's collaborative focus, while showing the cognitive and cooperative strengths of ExplorViz. Furthermore, ARENA2's environment provides lots of personal and projection space, which is a factor for effective collaboration. The participants' opinions obtained, showing possibilities for future improvement and limitations of the current Synchronization Feature.

Contents

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Document Structure . . . 2

2 Goals **3**
2.1 G1: Development-Design of the Basic Concept . . . 3
2.2 G2: Implementation according to Development-Design . . . 3
2.2.1 G2.1: Proof of Concept . . . 3
2.2.2 G2.2: Expand Prototype . . . 4
2.2.3 G2.3: Implementation in ARENA2 . . . 4
2.3 G3: Evaluation . . . 4
2.4 G4: Optional Development . . . 4
2.4.1 G4.1: Automatically Starting the Synchronization . . . 4
2.4.2 G4.2: Support for an Unspecific Number of Devices . . . 4
2.4.3 G4.3: Use of Xbox-Controller . . . 5
2.4.4 G4.4: Uploading or Adapting Configuration in Frontend . . . 5

3 Core Concepts and Technologies **7**
3.1 ARENA2: Visualization Laboratory . . . 7
3.2 Software . . . 7
3.2.1 ExplorViz . . . 7
3.2.2 Digital Earth Viewer . . . 11
3.3 Frameworks and Libraries . . . 12
3.3.1 Three.js . . . 12
3.3.2 Ember.js . . . 13
3.4 Multiple Projection Common Data Interchange . . . 14
3.5 Graphical Concepts . . . 14
3.5.1 Camera's Frustum . . . 14
3.5.2 Euler's Angles: Yaw, Pitch, and Roll . . . 16
3.5.3 Position . . . 17
3.5.4 Quaternion . . . 17
3.5.5 Projection Matrix . . . 18
3.5.6 View Matrix . . . 19

4 Technical Approach **21**
4.1 Concept 21
4.2 Initial Prototype 22
4.3 Synchronizing Four Instances 23
4.4 Automatic Initiation and Configuration Format 24
4.5 Implementation in the Intended Environment: ARENA2 26

5 Synchronization Feature **29**
5.1 Infrastructure: Synchronization Process Overview 29
5.1.1 SynchronizationSession: The Information Storage 29
5.1.2 Projector Configuration 31
5.1.3 Initiating and Updating Synchronization: Frontend Perspective 32
5.1.4 Initiating Synchronization: Collaboration Service Perspective 36
5.2 Graphical Prework: Monitor-Solution 37
5.2.1 Camera Attempt 37
5.2.2 Matrix Approach 40
5.3 Graphical Processing: ARENA2 43
5.3.1 Testing Monitor-Solution 44
5.3.2 Frustum Manipulation 45
5.4 ARENA2 Specific 50
5.4.1 CORS Workaround 50
5.4.2 Script Based Start 50
5.4.3 Hide Timeline 50

6 Evaluation **51**
6.1 Motivation and Research Questions 51
6.2 Participants 52
6.3 Material 52
6.3.1 Digital Survey 52
6.3.2 Tasks 53
6.3.3 Evaluation Questions 53
6.4 Procedure 54
6.5 Results 54
6.5.1 Usability: User Experience 55
6.5.2 Cognitive Support 55
6.5.3 Enhance Collaboration 56
6.6 Discussion 56
6.6.1 Result Interpretation 56
6.6.2 Limitations 58

7 Related Work **61**

8 Conclusion and Future Work **63**
8.1 Conclusion 63
8.2 Future Work 63

A Evaluation Tasks **65**

B Evaluation Statements **67**

C Evaluation Open-Text Questions **69**

Bibliography **71**

Chapter 1

Introduction

1.1 Motivation

In software engineering, understanding the structure and behavior of distributed software systems is a challenging task. This underscores the need for software engineers to have effective and efficient tools to facilitate program comprehension. These tools should provide insights into their runtime behavior —including method invocations, communication paths, and component interactions— as well as the dependencies between different parts of the program and the system's architecture [Krause-Glau et al. 2022b]. Meeting these requirements, ExplorViz [Hasselbring et al. 2020], an open-source software visualization tool, provides a comprehensive and interactive platform. Not only does it aid in understanding a point of interest in the program's analysis, such as runtime behavior and dependencies, but it also promotes collaborative comprehension of the software, by providing features like visually highlighting the point of interest.

This thesis aims to leverage the capabilities of ExplorViz in the context of collaborative program comprehension in a specific environment, ARENA2 [*ARENA - GEOMAR - Helmholtz Centre for Ocean Research Kiel*]. Building on ExplorViz's existing support for technically enriched reality [Krause-Glau et al. 2022b], the evolution of ExplorViz into a spatially immersive collaboration space is a progression to explore further use cases of ExplorViz. This immersive environment, combined with the immediacy and multifaceted of real-life communication, is set within a comfortable meeting room. This not only enhances the user experience but also opens up a new possibility to use ExplorViz, demanding further exploration and innovation. The goal is to explore the potential of this tool to enhance the understanding of the interaction between ExplorViz and ARENA2.

To ensure the successful realization of the implementation of ExplorViz in ARENA2, our strategy involves conducting a comprehensive exploration of ExplorViz's features, capabilities, and customization options. Insights figured out from this investigation are integrated into the creation of our conceptual developmental plan (see Chapter 4), highlighting techniques of ExplorViz's visualization capabilities. This enables us to formulate a set of questions for evaluating how well the environment supports software visualization and collaborative program comprehension. The responses to these questions explore the cognitive and collaborative capabilities of our implementation (see Chapter 6).

This thesis is supported by a thorough review of relevant literature. Important work by

Hasselbring et al. [2020] demonstrate the efficacy of ExplorViz in various software engineering domains, highlighting its potential for comprehensive visualization and analysis of distributed software systems. Furthermore, Krause-Glau et al. [2022a] empathizes practical scenarios for ExplorViz like *Knowledge Transfer* or *Team Meetings*, providing the need for visualizations of complexity rising software systems to foster a similar mental model of a software system distributed over all developers or users.

In conclusion, this thesis seeks to leverage the capabilities of ExplorViz to find further valuable insights into collaborative program comprehension using software visualizations in a specified immersive environment.

1.2 Document Structure

This thesis starts by pinpointing its core objectives, intending to lay a path for both the conceptual and practical plans to follow. Before we discuss the approaches and implementations of these goals, we provide insights of relevant technologies and concepts. Especially graphical understanding is important, because we provide deeper information about the graphical processing using virtual cameras itself and its underlying transformation matrices for the rendered scene. Building on this foundation, we then discuss a conceptual approach, detailing our conceptual plan to find knowledge by experimenting with these graphical concepts. These explorations do not stay theoretical; they show the way for a hands-on practical implementation. By using insights from these experiments, we integrate ExplorViz into ARENA2's setup. The following evaluation of the Synchronization Feature, summarizes its capabilities and disadvantages, especially when used in combination with ARENA2. This thesis concludes with future work based on evaluation outcomes and usability tests.

We encourage readers who wish to launch ExplorViz in ARENA2 to specifically refer to ARENA2's setup instructions in Section 5.4. For a more theoretical or code-based explanation, see Section 5.1.3.

Chapter 2

Goals

In the following chapter, we briefly describe the objectives of the bachelor's thesis, we discuss in more conceptually detail in the approach (see Chapter 4) and more practically or experimentally in the Synchronization Feature (see Chapter 5).

2.1 G1: Development-Design of the Basic Concept

To extend an existing program, it is necessary to have a fundamental understanding of how the program works in order to identify relevant parts of the software. Thus, we create a draft based on the identified communication of these relevant parts and their characteristics. Consideration is also given to which existing components (e.g. microservices in Figure 3.3) of ExplorViz (see Section 3.2.1) can be used for the implementation. Furthermore, necessary configurations for synchronizing multiple visualizations are identified and defined. In this case, considerations are given to the necessity of the MPCDI (see Section 3.4) file format. It is plausible that it may not be required if the solution can be stored through an alternative method.

2.2 G2: Implementation according to Development-Design

To reduce complexity, the design is first applied to two instances and then to five instances of ExplorViz. The synchronization of five instances aims to represent a comprehensive simulation of the scenario in ARENA2, without taking into account the surface of the illumination space. After creating the infrastructure for the synchronization process of five instances, we aim to implement this solution in the initially intended environment. The projection surface in ARENA2 is a hemispherical shape, which necessitates some adjustments to the manipulated camera attributes. The above statement translates to the following sub-objectives.

2.2.1 G2.1: Proof of Concept

Our aim is to create the basic functionality for synchronizing at least two instances, constructing the infrastructure for this feature while considering the existing parts of the architecture of ExplorViz and ARENA2. This results in a prototype with hard-coded

attributes as a provisional solution to manage complexity, thereby creating a kind of testing environment for more automated configuration processes.

2.2.2 G2.2: Expand Prototype

The prototype created in G2.1 (see Section 2.2.1) evolves from relying on hard-coded values to a more automated configuration process. We modify the existing concept to enable the synchronization of at least five instances and create a file format that encapsulates the necessary attributes for synchronization. In the later stages of development, there may be more time to extend the solution to an unspecified number of instances (see Section 2.4.2).

2.2.3 G2.3: Implementation in ARENA2

Once we developed a solution for synchronizing at least five instances, we are ready to implement it in ARENA2. However, it is important to note that ARENA2's projection space is a dome, a factor that the existing concept must adapt to. Therefore, we adapt and test the enhanced prototype from G2.2 (see Section 2.2.2) within ARENA2 and analyze any problems that arise.

2.3 G3: Evaluation

The Evaluation aims to ensure the practicality and effectiveness of the solution. Thus, once ExplorViz is fully functional in ARENA2, several participants are invited to assess it, which allow us to qualitative explore the benefits of implementing such a tool in a specific environment.

2.4 G4: Optional Development

Upon successful implementation of implementation's fundamental features and achieving correct execution in ARENA2, we consider the following optional implementations, listed in order of priority:

2.4.1 G4.1: Automatically Starting the Synchronization

Initiate the synchronization feature through a script, allowing for automated start-up of every ExplorViz instance on computers connected to ARENA2 projectors.

2.4.2 G4.2: Support for an Unspecific Number of Devices

As highlighted in G2.2 (see Section 2.2.2), we aim to create a configuration process adaptable to an unspecified number of ExplorViz instances.

2.4.3 G4.3: Use of Xbox-Controller

Introduce the use of an Xbox-controller to interact with the software landscape provided by ExplorViz.

2.4.4 G4.4: Uploading or Adapting Configuration in Frontend

Expand the Frontend to include a form for either uploading an external configuration file or manually adjusting configuration variables.

Chapter 3

Core Concepts and Technologies

In this chapter are discussed all relevant concepts and technologies. Especially for the graphical understanding, the section Three.js and Graphical Concepts provide more insights in saving graphical information in data structures.

3.1 ARENA2: Visualization Laboratory

GEOMAR, the Helmholtz Centre for Ocean Research in Kiel [*GEOMAR Helmholtz Centre for Ocean Research Kiel*], has developed an immersive projection space named *Artificial Research Environment for Networked Analysis (ARENA2)* [*ARENA - GEOMAR - Helmholtz Centre for Ocean Research Kiel*] [Kwasnitschka et al. 2023]. ARENA2 consists of a dome tilted at an angle, illuminated from the inside by five projectors (see 1-5 in Figure 3.1). Each projector is connected to a high-performance cluster computer, and the five images from these computers are synchronized to form a whole image. The *Main* (refer to the bottom right in Figure 3.1) serves as the control instance for synchronizing the illumination of the projectors. This control provides possibilities to adjust the visualization within the dome.

This synchronization process requires configurations for the frustum—the part of the virtual camera that represents what is seen in real life. Often, this configuration is done using the MPCDI (Section 3.4) format, which supports features like *Warping* and *Blending*. Warping involves the adjustment of angles to create a seamless, immersive visual experience, while Blending deals with the brightness at overlapping points to ensure a uniform visual output.

The spatially immersive visualization laboratory, even prior to this thesis, has been leveraged for collaborative learning and understanding of information (see Section 3.2.2).

3.2 Software

3.2.1 ExplorViz

ExplorViz is an innovative web-based tool specifically designed to enhance program comprehension by enabling collaborative software visualization. It serves as a platform that allows developers to dive into the structure, behavior, and evolution of software systems, thereby improving the development and maintenance processes [Hasselbring et al. 2020].

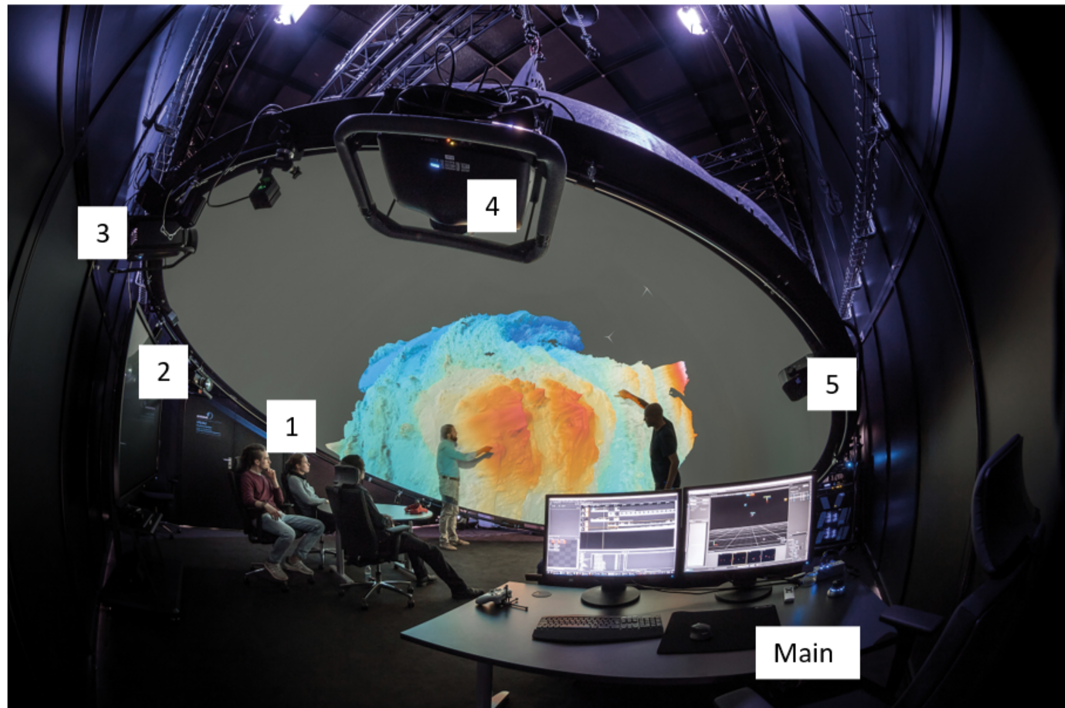

Figure 3.1. ARENA2 as meeting room using five projectors to visualize communication topics. This picture is adapted and retrieved from Kwasnitschka et al. 2023.

ExplorViz consists of a range of supported visualization techniques. In Krause-Glau et al. [2022b], they explore the utilization of static code analysis, runtime behavior, and software evolution as methods of visualization to enhance program comprehension and reduce cognitive load. They draw attention to the prevalent issue in software engineering: The dependence on text-based tools to comprehend software and collaborate with project members.

Understanding software systems is an essential task in software engineering that requires developers to comprehend complex structures and behaviors. Therefore, it must be supported by other than text-based tools to build an effective mental model for a software system [Krause-Glau et al. 2022a]. This is where ExplorViz comes into play, offering an interactive, web-based solution designed to boost program comprehension via collaborative software visualization.

Figure 3.2 presents a view of a software landscape. A deeper insight into the architecture of different software components can be accessed by clicking on them. This reveals software packages or components (depicted in blue and green), classes (shaped as gray pillars), and communications between various system components (highlighted in yellow). The height

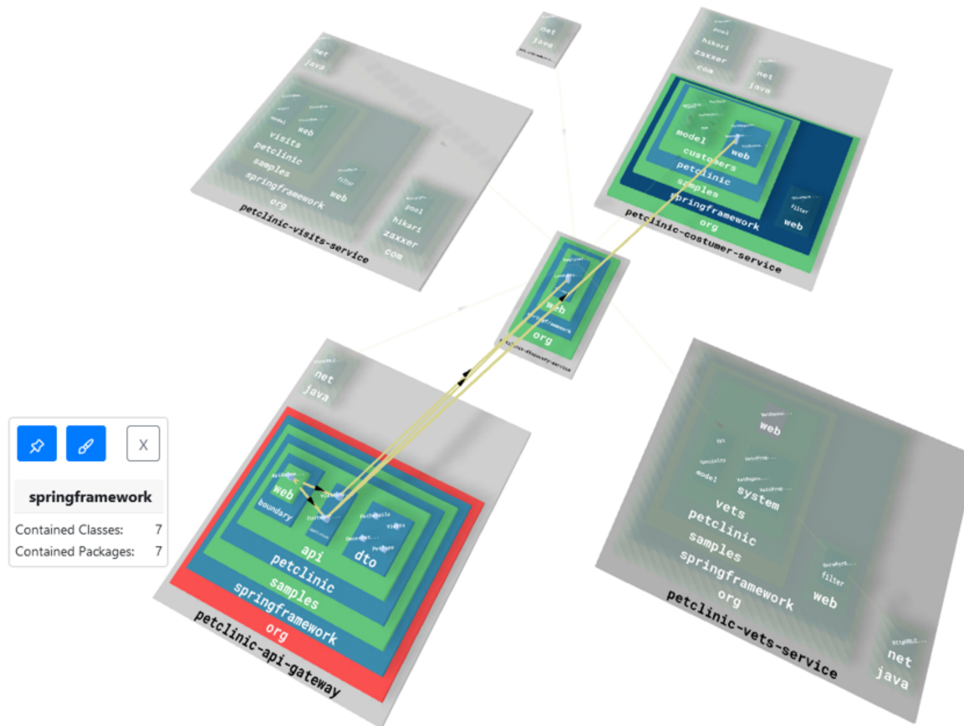

Figure 3.2. A visual representation of the software landscape, highlighting a component and showing more information of a corresponding component via pop up windows.

of the pillars on top of this detailed structure represent the quantity of objects, referring to the classes the pillars itself stand for. This representation is inspired by *City-Metaphor* [Wettel and Lanza 2007], a software system displayed as a city with numerous multi-story buildings (e.g. components, classes or packages) and streets that connect parts of the city (e.g. communication).

To highlight the collaborative capabilities, ExplorViz supports pop-up information windows (we refer to bottom left of Figure 3.2), containing information about the quantity of classes or packages a software part has. It is triggered by hovering the mouse cursor on the part of interest. By using one left-click, we highlight a part of the software landscape, while fading all irrelevant parts to this highlighted. Thus, the visualization of a software landscape facilitates an exploration of the system's architecture, while providing possibilities to analyze it in groups.

The following sections outline ExplorViz's support for the Synchronization Feature. We

start with an overview of the general architecture, highlighting the relevant frontend and backend software parts. Next, we discuss the concept of rendering in a web browser and how it is kept up-to-date. We conclude by examining the role and function of the Spectate Feature.

Architecture

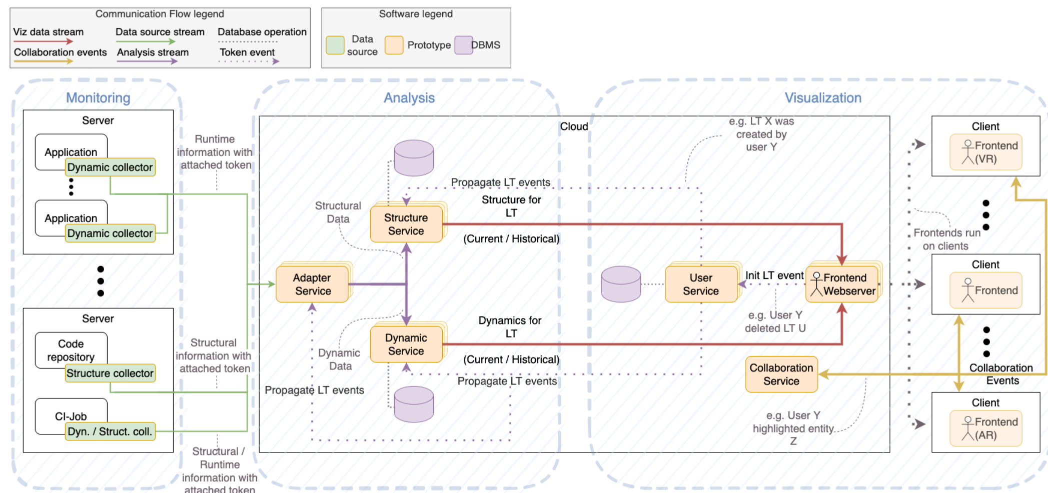

Figure 3.3. Microservice architecture of the backend of ExplorViz

ExplorViz is primarily divided into two parts: The frontend and backend. The backend architecture of ExplorViz consists of multiple microservices [Zirkelbach et al. 2018]. Figure 3.3 offers an overview of the backend architecture, with the Collaboration Service being a key feature for collaboration. We note that the Frontend of ExplorViz runs on client machines, allowing for multiple instances of a collaboration session managed by the Collaboration Service. This session facilitates communication between clients by managing messages that enable features, such as spectating.

On the other hand, the Frontend utilizes the JavaScript library Three.js (Section 3.3.1) to render software landscapes on a canvas using WebGL [*WebGL model view projection - Web APIs | MDN*]. Here, the position and rotation of a virtual camera are set to present the software landscape in a scene using a renderer (see for more details in the following section Section 3.3.1). The Frontend does more than just serve the visual aspects of the web application; it also fulfills backend functions. As a result, in following chapters, Frontend is capitalized to signify its importance as a service, similar to the Collaboration Service.

Browser Rendering

It is crucial to utilize the browser-rendering capabilities of ExplorViz's Frontend for this thesis. The browser-rendering is setting up all the essential elements required for the visual experience of a software landscape. This includes, for example, (a) configuring camera controls for moving, zooming or rotating the software landscape. (b) Establishing the starting position of the software landscape, ensuring that users get the best initial view. (c) Integrating event handlers to manage and react to various inputs or interactions.

Rendering Loop

Apart from laying down the foundation for the rendered scene, ExplorViz also establishes a rendering loop. This loop tracks every modification or interaction within the scene. Whether it is pop-ups that need to be displayed, movements of the software landscape, or other scene-related events, the loop detects these changes. As soon as an event is recognized, the system initializes corresponding actions, ensuring that the visual display remains updated and interactive.

Spectate Feature

The *Spectate Feature* in ExplorViz allows one instance to observe another. It does this by copying the positional and rotational data from the spectated instance and sending it to one or multiple other ExplorViz instances.

For this to work, all instances must be in a shared space, a *room* provided by the Collaboration Service. When an instance joins this room, a *WebSocket* is established. This WebSocket is essential for sharing collaboration events, as shown in the bottom right part of Figure 3.3. These events are triggered when users interact with the software landscape or use the Spectate Feature in a shared room.

When an instance observes another, its position and rotation match the spectated instance's. If the target instance moves, an event is triggered. This event travels from the Frontend to the Collaboration Service. After being processed, the event returns to the Frontend, updating the observing instance's camera's position and rotation.

To ensure that observers only see the target instance's movements, the observers' camera controls are turned off during spectating. This makes sure the observers solely track the target without any manual adjustments.

3.2.2 Digital Earth Viewer

The *Digital Earth Viewer (DEV)* [*Digital Earth Viewer*] serves as a platform for interactive visualization of geospatial data (see Figure 3.4) and has seen implementation in ARENA2, providing therefor an ideal example for the implementation of ExplorViz. It was implemented in ARENA2 using the MPCDI file format (Section 3.4). This usage will be interesting if, during the goal realization of G2.3 (see Section 2.2.3), it becomes apparent

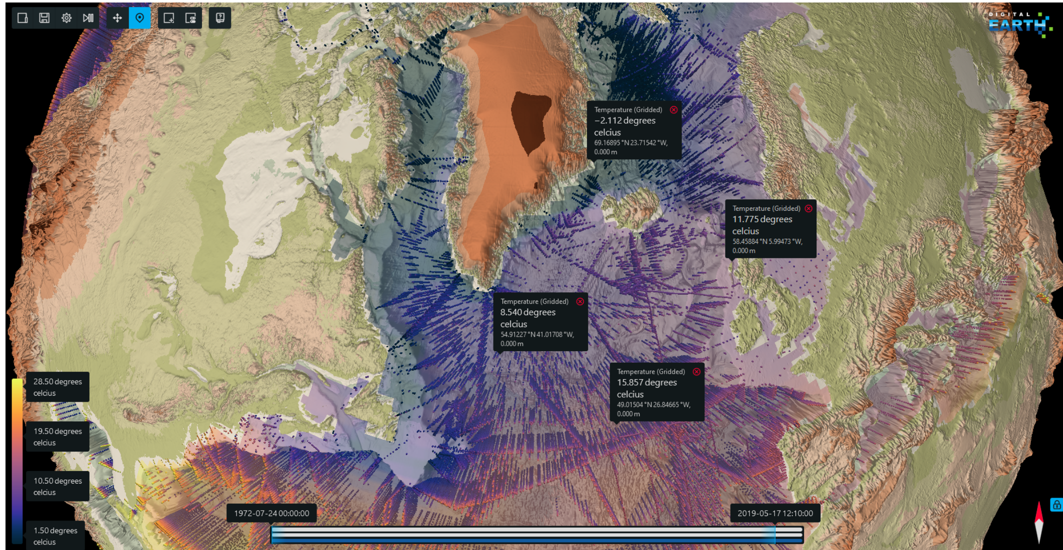

Figure 3.4. DEV presenting the parts of the world in a multi-layer view. This picture is retrieved from *Digital Earth Viewer*.

that the resources provided by ExplorViz and Three.js (Section 3.3.1) are not sufficient to achieve a flawless image through the synchronization of the projectors.

This tool integrates earth data sources, e.g. climate models, or geospatial datasets, to create a comprehensive and detailed picture of our planet. Through this unified presentation of geospatial information, users are empowered to navigate, explore, and comprehend earth's dynamics.

3.3 Frameworks and Libraries

3.3.1 Three.js

Three.js [*Three.js – JavaScript 3D Library*] is a JavaScript [*JavaScript*] library that provides a comprehensive set of tools and functionalities for creating and rendering three-dimensional objects, scenes, and animations in a web browser environment. It is flexible and extensible, which allows users to create diverse three-dimensional scenes with customization possibilities. The most important components are *Scene*, *Camera* and *Renderer*. These parts are required to place and position objects in a three-dimensional scene. The camera controls the viewpoint of the user. The definition of visualization space is provided by the scene, and the renderer places the actual visible objects in a scene.

Object3D

Object3D is a fundamental class in Three.js. It serves as the base for many objects in the library and acts in this thesis as a container for graphical information in three-dimensional space. Most objects in Three.js that can be added to a scene are derived from Object3D. The class simplifies the management of objects in three-dimensions by providing key properties and methods, including position, quaternion, view matrix, and projection matrix.

Given its comprehensive representation of graphical data, Object3D is also referred to as camera in this thesis. Every time when graphical information from the Main are copied or cloned, it is an object of type Object3D.

PerspectiveCamera

The PerspectiveCamera is the relevant type of virtual camera of ExplorViz's graphical setup and inherits from Object3D. It simulates the way human eyes perceive the world around them. It mimics how objects get smaller the further away they are from the viewer, creating a sense of depth.

When we create a PerspectiveCamera, we need to provide:

- *Field Of View (fov)*: This is the extent of the scene that is seen on the display at once.
- *Aspect Ratio*: This is generally the width of the element divided by its height.
- *Near* and *Far*: These values define the nearest and farthest points, using the concept of clipping planes, that will be defined in a three-dimensional setup.

3.3.2 Ember.js

Ember.js [*Ember.js - A framework for ambitious web developers*] is a JavaScript [*JavaScript*] framework designed for building web applications. The core features of this framework include Ember CLI – a toolkit for creating and managing applications, a component system for reusable UI elements, a routing mechanism, and tools for data management within applications [*The Ember CLI - Introduction - Ember CLI Guides*].

The essential building elements of Ember's UI are components, which are reusable parts of an Ember based software. Controllers are storage places for application state and actions, complementing models with display logic. The router, translates URLs into a series of templates represented by routes, linking specific URLs to templates and models. This ensures shareable application states. Additionally, Ember offers a query parameter feature to include application state information in the URL without additional routing [*How To Use The Guides - Getting Started - Ember Guides*].

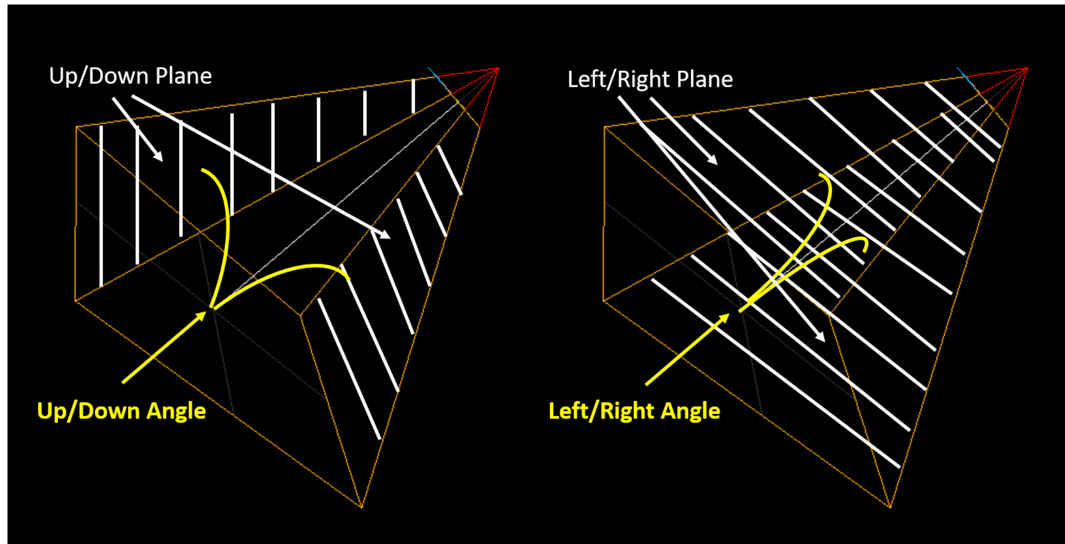

Figure 3.5. Up, down, left and right clipping plane and corresponding angles. This figure is retrieved from Three.js's editor [*three.js editor*].

3.4 Multiple Projection Common Data Interchange

The Video Electronics Standards Association [*VESA - Interface Standards for The Display Industry*] released the *Multiple Projection Common Data Interchange (MPCDI)* for projection calibration systems in multi-display configuration. This standard allows these systems to generate data necessary to combine individual display components (in our case, projectors) into a seamless single image. Furthermore, it includes specific information (e.g. geometric data of the surface they project on) about each of the displays. Two specifics, which are relevant for the implementation of the DEV in ARENA2, are Warping and Blending (Section 3.1). Furthermore, MPCDI provides information about the manipulation of the virtual camera's frustum, to serve the place they are defined to illuminate on (see Section 3.5.1 and Section 3.5.2).

3.5 Graphical Concepts

3.5.1 Camera's Frustum

A frustum, in the context of a virtual camera, refers to the portion of space in the modeled world or three-dimensional space that can be seen or captured by the camera. The frustum represents the visualized space of the camera and is typically visualized as a truncated pyramid [*Three.js – JavaScript 3D Library*]. As depicted in Figure 3.5, the top of the pyramid

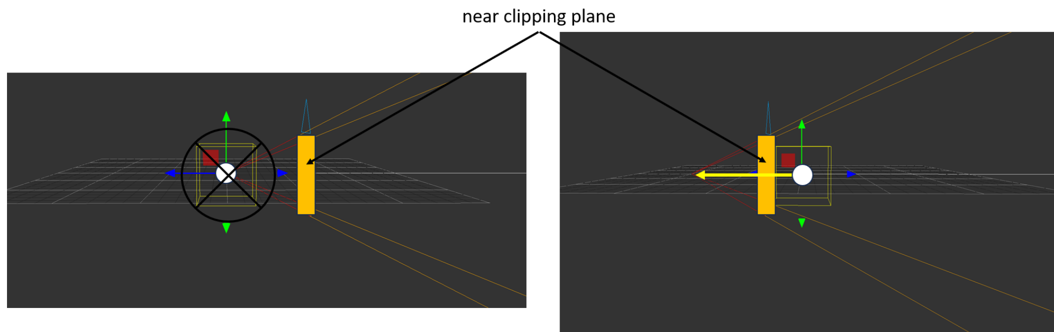

Figure 3.6. Manipulating the camera's position to move towards an object, making it visible. This figure is retrieved from Three.js's editor [*three.js editor*].

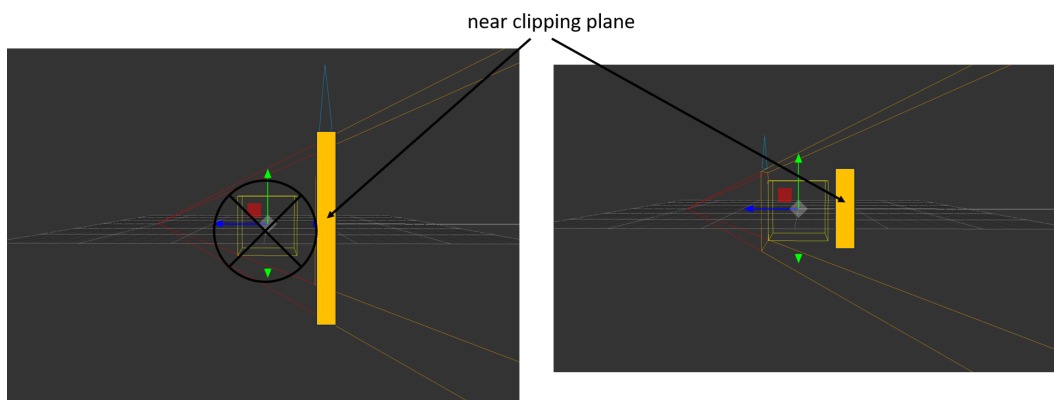

Figure 3.7. Adjusting the camera's frustum's near clipping plane to shift the rendering space towards the object. This figure is retrieved from Three.js's editor [*three.js editor*].

is removed, leaving two bases: A smaller one closer to the camera's lens (known as the *near clipping plane*) and a larger one further from the camera (referred to as the *far clipping plane*).

Understanding the frustum is vital for those using computer graphics, as it underpins rendering and optimization processes. For instance, attributes of a virtual camera, such as its position, can influence other specifications. In Figure 3.6, the manipulation of a virtual camera's position makes an object visible by moving the camera forward. Also, achieving the visibility can be done by adjusting the near clipping plane, which shortens the size of the removed top of the truncated pyramid (as shown in Figure 3.7).

Any objects too close to the camera (before the near clipping plane) or too distant (beyond the far clipping plane) will not be rendered. The frustum also has four sides: The up, the down, the left, and the right clipping planes that connect the near and the far clipping planes. These sides determine the *horizontal field of view (HFOV)* and *vertical field*

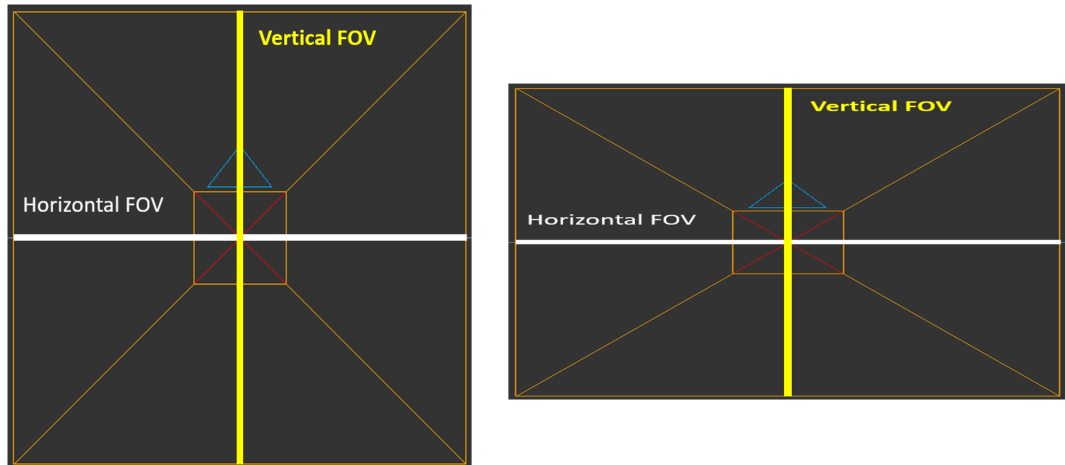

Figure 3.8. Adjusting HFOV and VFOV to transition from a 1:1 to a 16:9 aspect. This figure is retrieved from Three.js's editor [*three.js editor*].

of view (VFOV) of the camera and can be adjusted using corresponding angles. Anything outside these planes, also, will not be rendered.

Using these clipping planes, we can characterize the visual space and the volume of the rendering space. By setting angles for the clipping planes, we can also modify the aspect of the visualized image, like a 16:9 resolution, which denotes a visualization that is 16 units wide and 9 units tall (see right Figure 3.8). For instance, a wider angle can be set for the left and right clipping planes than for the up and down planes to establish a 16:9 aspect ratio. Keeping all angles the same, we achieve a 1:1 aspect ratio, as visualized on the left side of Figure 3.8.

3.5.2 Euler's Angles: Yaw, Pitch, and Roll

Euler's angles [Ang and Tourassis 1987] represent rotation in three-dimensional space using three angles, as depicted in Figure 3.9: *Yaw*, *Pitch*, and *Roll*. In Three.js, rotations using Euler's angles can be set with `THREE.Euler`. The order in which rotations are applied is crucial, as it influences the final orientation. While in Three.js one can specify this order (e.g., 'XYZ', 'YZX', etc.), this thesis requires interpreting single rotations as either positive or negative.

Together, yaw, pitch, and roll provide a comprehensive description of an object's orientation in space.

▷ **Yaw** refers to the rotation around the vertical axis. It provides the side-to-side movement of the nose of an object, essentially indicating its deviation from north.

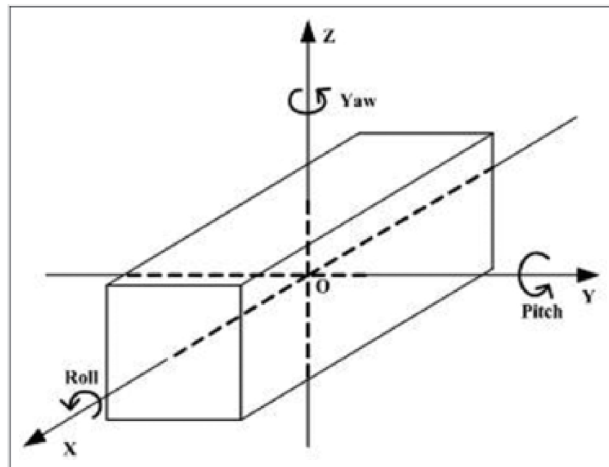

Figure 3.9. Concept of yaw, pitch, and roll [Liu and Zhao 2013].

- ▻ **Pitch** describes the rotation around the lateral axis, which runs from the left end of an object to another. This rotation corresponds to the up-and-down movement.
- ▻ **Roll** refers to the rotation around the longitudinal axis. This rotation indicates the tilting motion.

The axes in Figure 3.9 differ from those in Figure 3.11. This difference highlights the challenge in accurately interpreting the rotation axes. When applying multiple quaternions, position changes, and frustum manipulations, it is to consider the configuration order and understand the impact of each step. To determine the axes' assignment of yaw, pitch and roll, we encourage the developer to test simple isolated rotations (see Figure 3.11).

3.5.3 Position

The position attribute in Three.js designates the camera's location within three-dimensional space. Considering x, y, and z components, it defines the camera's coordinates in the space where objects are rendered. As illustrated in Figure 3.10, the camera's initial point moves across the three dimensions based on the camera controls used, such as *OrbitControls* from Three.js. This movement not only alters the visual space but also shifts the perspective of a three-dimensional model.

3.5.4 Quaternion

A common method for implementing a complex rotation of an object is through the use of quaternions. Figure 3.11 displays basic quaternions, each relying on a single rotation

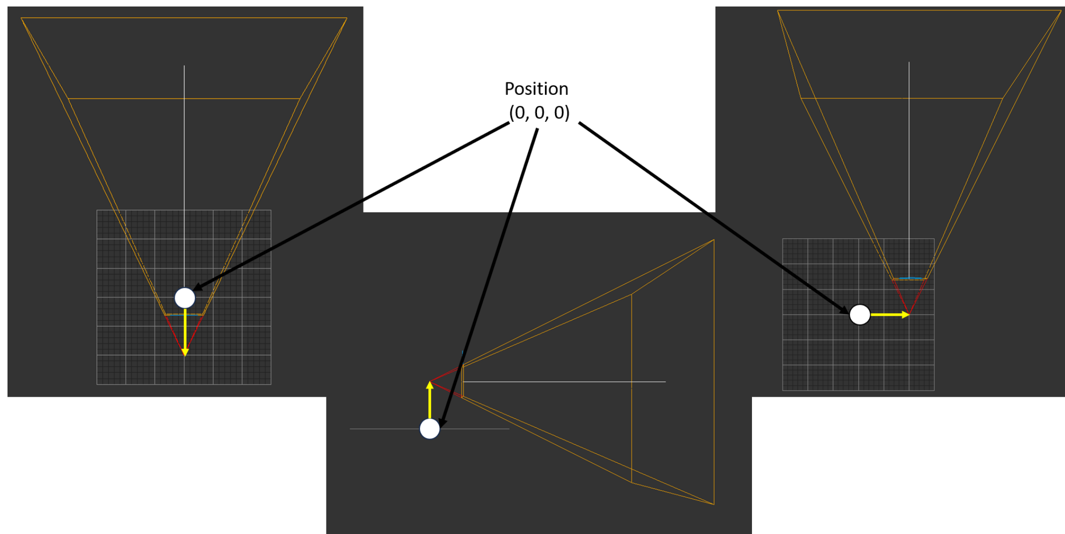

Figure 3.10. Camera position changes: At x (see top right: x = 10, y = 0, z = 0), at y (see middle: x = 0, y = 10, z = 0), and at z (see top left: x = 0, y = 0, z = 10). This figure is retrieved from Three.js's editor [*three.js editor*].

axis. Three.js offers approaches for manipulating these quaternions or defining them as a whole. Internally, Three.js utilizes quaternions when objects are rotated. In this thesis, we do not dive deeply into quaternion's theoretical aspects. Instead, quaternions should be understood as rotation tools with unique problem-solving capabilities. For instance, they can be more efficient and bypass the *Gimbal Lock Problem* [Hemingway and O'Reilly 2018] that might arise with Euler's angles. When rotations are depicted using Euler's angles, a state can be reached in which one degree of freedom is lost, making the representation of certain rotations not reproducible.

For the purposes of this thesis, there is no necessity to directly modify quaternions. Instead, we rely on Euler's angles, allowing Three.js to handle the conversion to quaternions on our behalf.

3.5.5 Projection Matrix

The 4x4 projection matrix is transforming three-dimensional coordinates in camera space into the two-dimensional coordinates displayed on screens. Within the camera, this matrix is configured to transform these coordinates into a frustum, making objects appear smaller the further they are from the camera [*WebGL model view projection - Web APIs | MDN*].

In Three.js, the projection matrix is automatically generated once the camera is set up. However, it is crucial to be mindful of potential overwriting issues when using both, the projection matrix and the camera, to achieve a desired result. Alterations to the

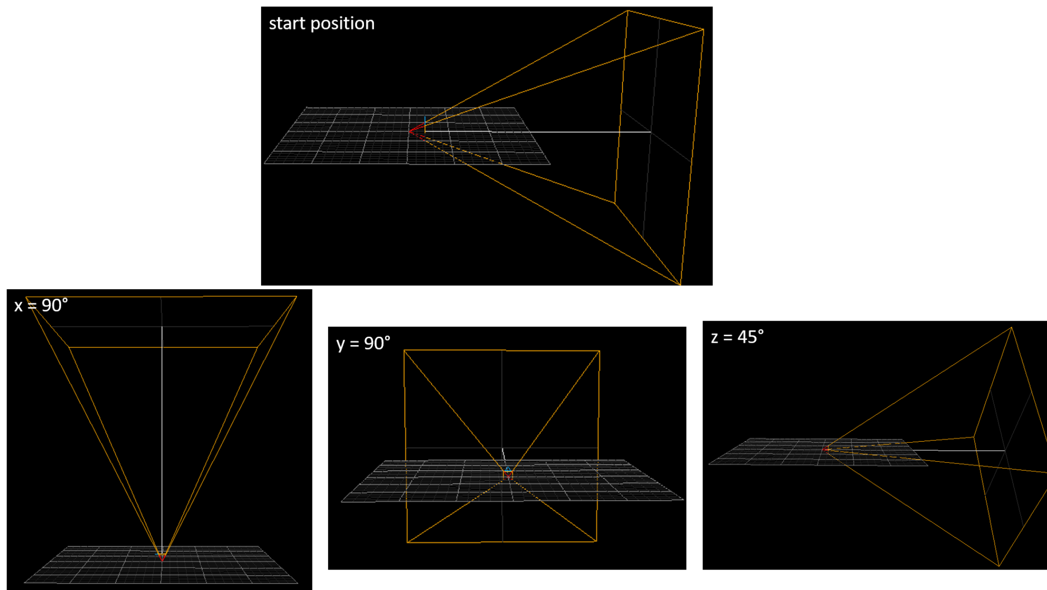

Figure 3.11. Change camera quaternion considering the starting position of the left picture to x = 90°, the middle picture to y = 90°, and the right picture to z = 45°. This figure is retrieved from Three.js's editor [*three.js editor*].

virtual camera's frustum will only take effect after triggering a from Three.js provided function, that updates the projection matrix. This action effectively overrides any manual modifications to the projection matrix by generating a new matrix based on the camera attributes such as `fov, aspect ratio, near, and far` (refer to Section 3.3.1).

3.5.6 View Matrix

While the projection matrix defines the visual space, the view matrix emulates the position and rotation of the camera within that space. Adjusting the view matrix allows for modifications to the viewpoint, altering the perspective from which we observe the three-dimensional environment [*WebGL model view projection - Web APIs | MDN*]. In Three.js, this matrix is not directly accessible but is constructed using the camera's position, up-vector, and quaternion values. Contrary to the projection matrix, the view matrix updates automatically whenever related attributes are altered.

Chapter 4

Technical Approach

The chapter first outlines the general concept of this project, followed by a discussion of the implementation process.

4.1 Concept

The primary aim of this thesis is to implement the pre-existing software, ExplorViz, within the immersive projection environment ARENA2. To realize this objective, it is necessary to synchronize five separate ExplorViz instances, all under the governance of an additional Main (control) instance (see Figure 4.1).

The process of synchronizing multiple ExplorViz instances leverages the infrastructure of the existing Collaboration Service (see Section 3.2.1). This service enables the initiation of a room, which, in turn, establishes a WebSocket connection between the Frontend and the Collaboration Service.

Once ExplorViz instances join the same room, they gain the capability to spectate each other. The Spectate Feature is available after the WebSocket connection setup and stays in interaction with the Collaboration Service. When an instance starts spectating, it is labelled as a spectator, and it copies the positional data of the instance it is spectating. The process involves managing information, with messages being sent and responses being received via the WebSocket connection.

For synchronizing multiple ExplorViz instances, the plan is to designate the Main instance as the spectated entity (see bottom Figure 4.1). This synchronization procedure involves manipulating the copied position and quaternion data of the Main's camera to determine the visual area of the other ExplorViz instances meant to be synchronized. Each time the Main moves, the spectating cameras are readjusted similarly as they would be during spectating (using the browser rendering capabilities of ExplorViz). However, they do not portray the exact visual area of the Main, but rather, they do only display their allocated area (as illustrated conceptually on a monitor in Figure 4.2).

At the point, we have set up the infrastructure to access multiple instances. It is imperative to understand that when the landscape is moved, it is rotating. This quaternion would yield unsatisfactory visualizations if the camera update calculations are not adjusted accordingly (refer to Figure 4.3). The quaternion within a three-dimensional scene signifies that the synchronized instance must mirror the same but sectional visual area displayed by

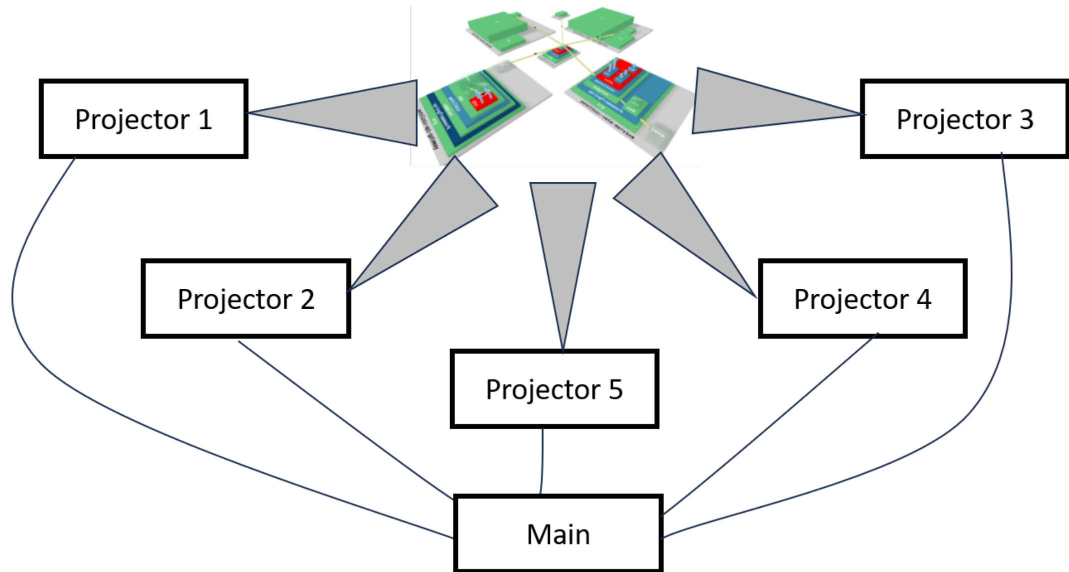

Figure 4.1. Concept of projector synchronization in ARENA2.

the Main. As such, the camera's perspective or the calculated perspective of the matrices of the camera should be adjusted in tandem with any positional and quaternial shifts that occur during the synchronization process.

Upon constructing the foundational infrastructure and achieving synchronization of multiple ExplorViz instances on a monitor, we apply the knowledge acquired during the implementation phase to test the system in ARENA2, keeping in mind the hemispherical projection surface. Our objective shifts towards adapting the solution from a flat two-dimensional monitor to a curved two-dimensional surface, with four side-projecting and one top-projecting source (referring to Figure 4.7). If adapting the solution to ARENA2 proves unfeasible, we use the insights gained from graphical experiments during the implementation phase to find a new strategy. The development of this alternative solution benefits from the operational setup of the DEV (Section 3.2.2), granting us more practical perspectives.

4.2 Initial Prototype

According to Goal G2.1 (Section 2.2.1), the implementation of synchronizing two instances on a monitor serves as a test environment to build the infrastructure for the Synchronization Feature. The frontend is extended to include an option in the right-click menu for collaboration, which involves starting the synchronization manually.

The implementation takes into account both horizontal and vertical synchronization of

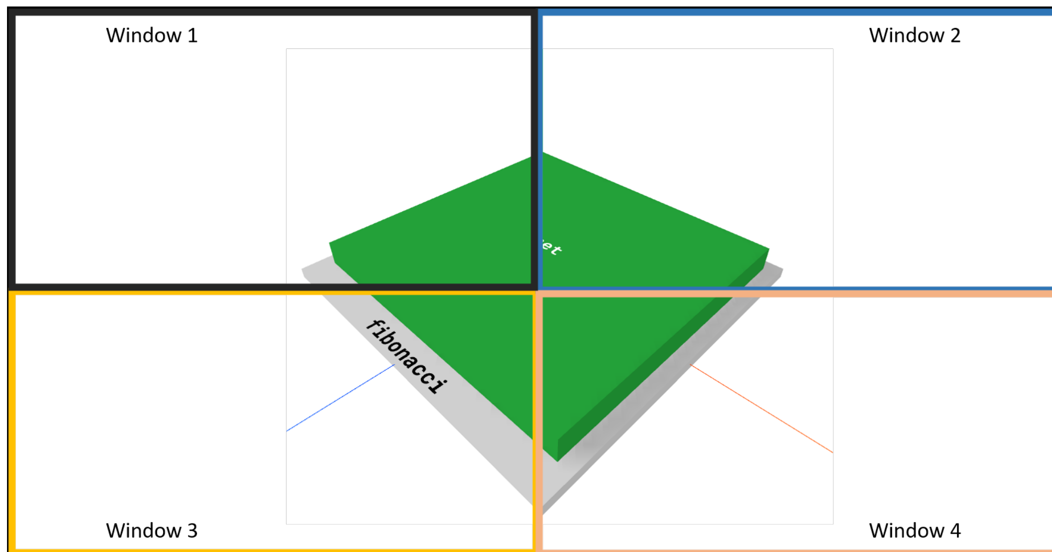

Figure 4.2. Conceptual visualization of synchronizing four ExplorViz instances.

the initial starting position of the software landscape. By enabling the testing of perspectives depending on how the visual areas of the cameras are divided. First, we work with a two-dimensional starting position of the software landscape, which makes considerations for perspective unnecessary at first, as there is no rotation occurring in the third dimension that could lead to misalignment in synchronization (see Figure 4.4). This approach provides an opportunity to examine the infrastructure and general concept's functionality. Once the infrastructure is set up to allow changes to camera objects, we can progress to more complex synchronization tasks (see Figure 4.6), highlighting the unsatisfactory shift in a software landscape's viewpoint when disregarding the three-dimensional scene, as visualized in Figure 4.5. This whole step of synchronizing two instances positional, and quaterinal, results in hard-coded values, which change is addressed in the following implementation process.

4.3 Synchronizing Four Instances

While our initial objective is to synchronize five instances, as outlined in Section 2.2.2, the intention was to replicate ARENA2's setup without factoring in its hemispherical attributes. However, during the implementation phase, we find that the fifth instance is redundant. Dividing a monitor's display into four equally spaced instances is sufficient. Introducing a fifth instance would mean placing it over the other quadrants, providing no additional insights. The challenge we aim to address does not come from the number

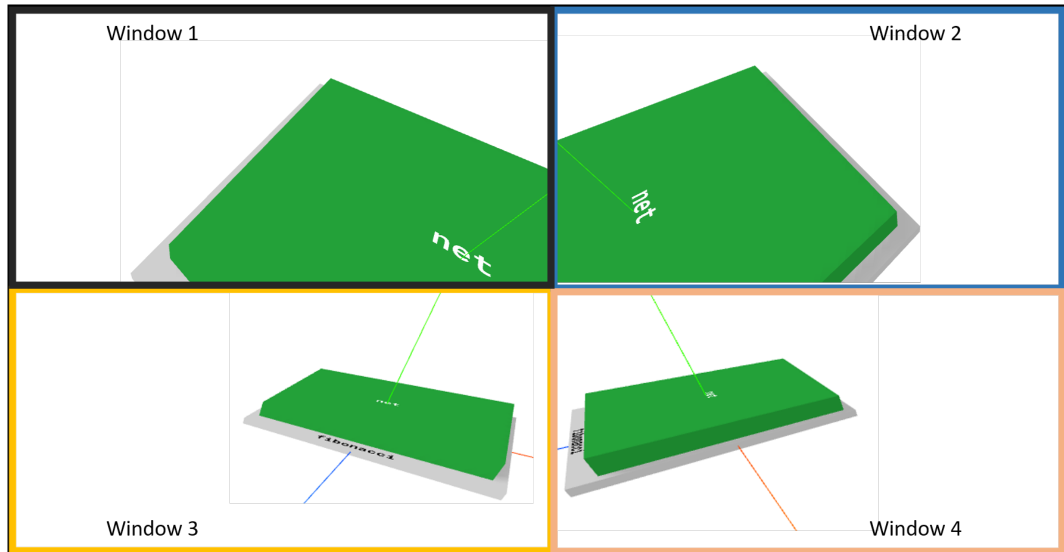

Figure 4.3. Conceptual view of the software landscapes displayed across four independent ExplorViz instances, assuming movement occurs without any modifications to the camera update functions.

of synchronized instances. Instead, the true task lies in the synchronization process itself: Combining multiple sets of manipulated graphical data to achieve a pre-defined visual output consistent with the Main instance.

Hence, we are extending the principle established for two instances to work for four, as depicted in Figure 4.2 and Figure 4.3.

4.4 Automatic Initiation and Configuration Format

The shift towards ARENA2 environment discovers the redundancy of manual starts. It emphasized our need to primarily concentrate on automating the initialization process. As we extended the prototype to handle the synchronization of four instances, the next logical progression is to work on the synchronization initiation. We target an automatic approach. To accomplish this, we adopt the use of query parameters. These parameters are designed to define the relevant variables crucial to the Synchronization Feature. Once set, these variables trigger the necessary processes, replicating the steps involved in manual synchronization initiation. Our goal is to eliminate hard-coded values from the code. To achieve this, we are defining a configuration format and outsourcing the relevant data to JSON files for this feature (goal G2.2; Section 2.2.2). The Collaboration Service is able to accommodate this format. Initially, this structure is filled with placeholder data suitable for monitor solutions. This foundation facilitates modifications, enabling the transition and

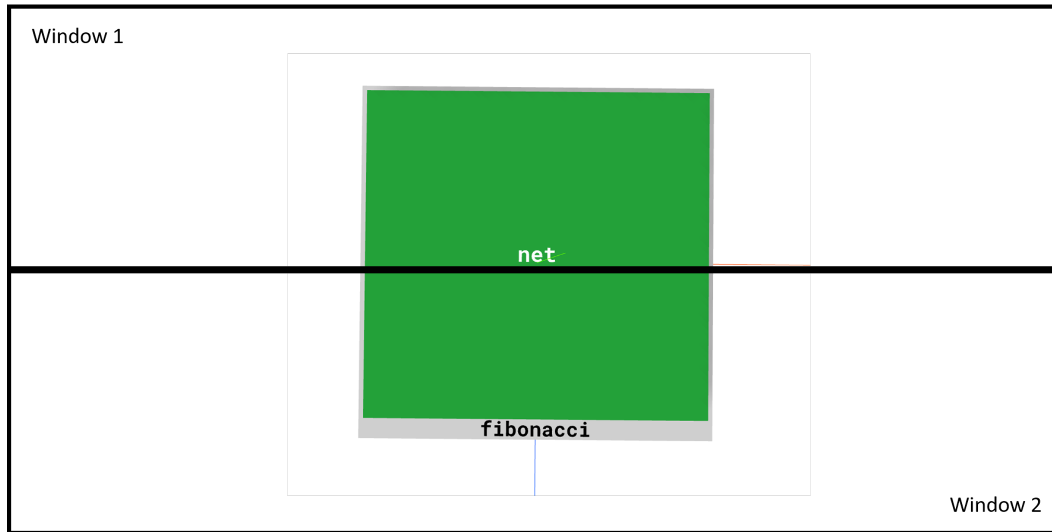

Figure 4.4. Conceptual horizontal synchronization of two instances in two-dimensionality.

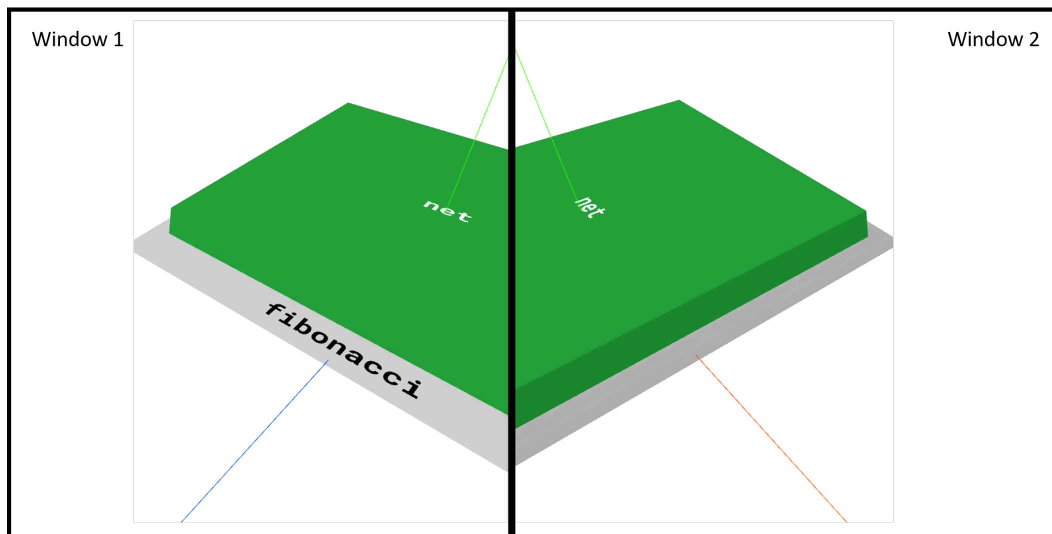

Figure 4.5. Conceptual vertical synchronization output when not considering change of rotation and position in a three-dimensional scene.

adaptation to ARENA2's requirements of receiving the MPCDI data.

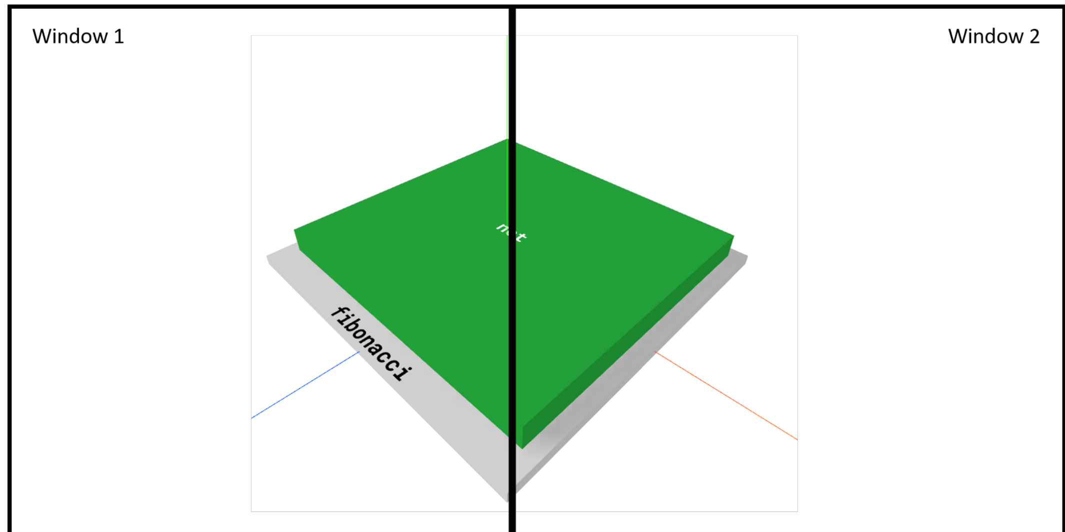

Figure 4.6. Correct camera rotation and position in three-dimensional using synchronizing two instances.

4.5 Implementation in the Intended Environment: ARENA2

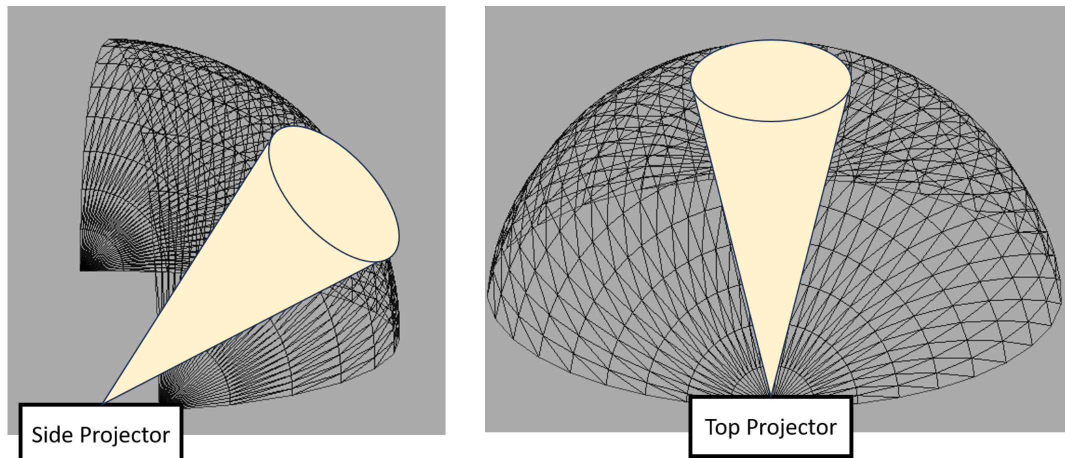

Figure 4.7. Conceptual view of one side-projecting and the top-projecting projector in ARENA2. This figure is retrieved from Three.js's editor [*three.js editor*].

The final implementation is in ARENA2 (see Goal G2.3; Section 2.2.3), taking the hemisphere into consideration. It involves adjusting the manipulation of position and

rotation, and considering the MPCDI file format (see Section 3.4) for the inconsistent image composition in ARENA2 (referring to Figure 5.11). MPCDI facilitates information exchange for multi-projections and is specialized in synchronizing a consistent image on unusual surfaces. However, Warping and Blending are already implemented at the system level of ARENA2. The information exchange is managed through ExplorViz. Thus, the use of MPCDI could be redundant, unless we encounter unsolvable problems that can be addressed through the multi-projector file format. Nevertheless, the information of the MPCDI file of the DEV is needed to implement ExplorViz in ARENA2.

At first, we test the extended prototype, which works on monitors. If we can not adapt the prototype to a state, which synchronizes the five projectors in ARENA2 while controlled by the Main, we translate working implementations of the DEV to the graphical setup of ExplorViz. Additionally, we possibly need to consider the placement of the projectors on the bottom border of the dome (see Figure 5.14), or the supposedly special role of having only one top-projecting source (as mentioned and visualized in Figure 4.7).

Chapter 5

Synchronization Feature

In this chapter, first, we explain infrastructure of the *Synchronization Feature*, which is followed by the research for the synchronization of four instances (in following mentioned as *Monitor-Solution*). This last point does lead to the implementation of ExplorViz in ARENA2. To structure the insights, we separate between the graphical and infrastructure part.

For better understanding, the code examples in this chapter are adapted, and do not reflect the same syntax but are semantically equal. Additionally, `this.localUser` defines the user of a ExplorViz instance as an Ember service, which contains the camera that controls the visual of the corresponding instance. When we copy the Main's attributes, we just copy from a `Object3D` object, that updates the necessary attributes based on events. It functions as a remote storage from the corresponding `this.localUser`, to open up the possibility to copy attributes, which are referenced, and not directly affecting the ExplorViz instance's visual itself we are copying from.

5.1 Infrastructure: Synchronization Process Overview

This section dives into how ExplorViz's infrastructure extension for the Synchronization Feature. For following implementations we reuse the infrastructure of the Spectate Feature (Section 3.2.1) which is already implemented in ExplorViz. This is not only saving code by reusing it, it is also already a good start for the implementation. We catch the copied position and quaternion data, to then manipulate it for synchronizations' purpose. With initiating the process, we mean to describe the automatic start of the synchronization, which is also further described in Section 5.4.2.

5.1.1 SynchronizationSession: The Information Storage

The *SynchronizationSession* stores all data for the Synchronization Feature. Not only does it provide a framework for graphic manipulation by containing all the required methods for integrating ExplorViz into ARENA2, but it also organizes the graphical information (see Section 3.5) to ensure type safety and accurately apply effects to ExplorViz's software landscape. We define four types:

Listing 5.1. Projector configuration types.

```
// (1) Three-dimensional rotation angles.
export type YawPitchRoll = {
  yaw: number;
  pitch: number;
  roll: number;
};
// (2) Aspect Ratio angles.
export type ProjectorAngles = {
  left: number;
  right: number;
  up: number;
  down: number;
};
// (3) Aggregated type with projector identifier.
export type ProjectorConfigurations = {
  id: string;
  yawPitchRoll: YawPitchRoll;
  projectorAngles: ProjectorAngles;
};
// (4) Aggregated transformed three-dimensional rotation angles for ypr and tilt.
export type ProjectorQuaternion = {
  synchronizationQuaternion: THREE.Quaternion;
  domeTiltQuaternion: THREE.Quaternion;
};
```

(1) `YawPitchRoll` and (2) `ProjectorAngles` represent the data from ARENA2's MPCDI file, (3) `ProjectorConfigurations` serves as an aggregated type to ensure type safety for the configuration data sent from the Collaboration Service to the Frontend, and (4) `ProjectorQuaternion` stores the quaternions for synchronization and projection adjustments related to the dome tilt (see for technical details: (II) in Section 5.3.2 and for a conceptual view: Figure 5.1).

For (4), we convert the yaw, pitch, and roll values from the `YawPitchRoll` type into a Three.js Euler's angles object. The `THREE.Quaternion.setFromEuler(yawPitchRoll: THREE.Euler)` function from Three.js is then utilized to adjust the newly created quaternion to represent these Euler's angles.

Besides the projection manipulation data, the SynchronizationSession also retains the query parameters `deviceId` for the projector identification and the `roomId` for the room's name. The `deviceId` is essential for requesting manipulation information from the Collaboration Service. The `roomId` provides the room's name supplied by the Collaboration Service, required to set up a communication channel for the projector's synchronization based on the Main camera's position and quaternion (for more details on room management,

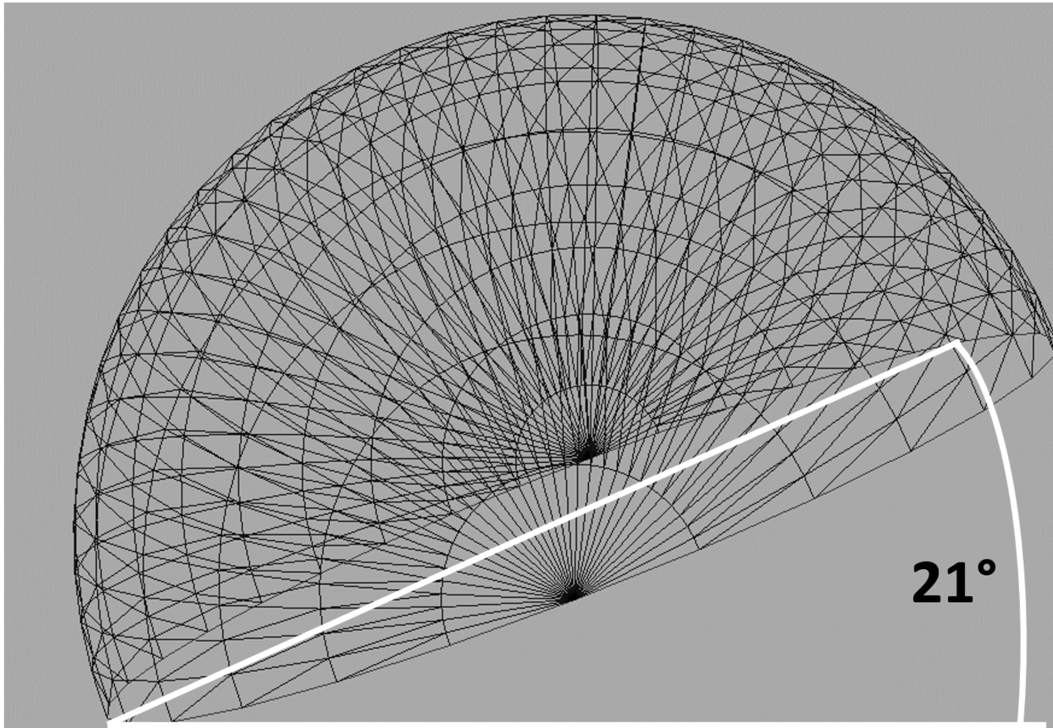

Figure 5.1. Conceptual view of the tilted dome in ARENA2. This figure is retrieved from Three.js's editor [*three.js editor*].

see Section 3.2.1). `deviceId` and `roomId` are set by processing the URL, which contains the query parameter (see Section 5.1.3).

5.1.2 Projector Configuration

Before exploring approaches to synchronize the five projections on a hemispherical surface, let us discuss the MPCDI file data.

For clarity, consider the following fictive snippet of an MPCDI file:

Listing 5.2. Fictive MPCDI snippet.

```
1 <region id="Projection1" xResolution ="1920" yResolution="1000">
2     <frustum>
3       <yaw>66.661227</yaw>
4       <pitch>-55.9123517</pitch>
5       <roll>17.71556767</roll>
6       <rightAngle>69.0000000</rightAngle>
```

```
7        <leftAngle>69.0000000</leftAngle>
8        <upAngle>39.0000000</upAngle>
9        <downAngle>39.000000</downAngle>
10     </frustum>
11 </region>
```

When using the axes' assignment of Figure 3.9, we interpret: The provided yaw, pitch, and roll angles adjust the projector's frustum by rotating it. For instance, a value of `66.661227` means rotating the projection about the vertical axis to the right by `66.661227` degrees (see the definitions of yaw, pitch, and roll in Section 3.5.2).

The `rightAngle`, the `leftAngle`, the `upAngle`, and the `downAngle` describe the angles from the projector's centerline to its edges, defining the frustum shape in a three-dimensional space.

Translating MPCDI to JSON

The MPCDI file for ARENA2 setting contains further data not required for synchronization. Thus, we formulate an individual format to store the relevant data (see Figure 5.2), adding an identifier for the specific projector. The following Java based translation process as well as the individual structured JSON file is located in the backend service: Collaboration Service.

To extract information from the JSON file, we employ the *Jackson Library* [*GitHub - FasterXML/jackson: Main Portal page for the Jackson project*]. This library offers tools for handling JSON data in Java, enabling us to convert JSON to Java objects without manual parsing. The *TypeReference* is an abstract class used to obtain full generics type information by subclassing, which aids in deserializing generic types. Meanwhile, *ObjectMapper* binds data, allowing for reading and writing JSON and mapping JSON data to Java classes like `ProjectorConfigurations`, `YawPitchRoll`, and `ProjectorAngles`. Once a `ProjectorConfigurations` object is created, it can be sent to the Frontend via a response message. The type definitions mentioned here are specific to the Java based Collaboration Service. Meanwhile, the types discussed in Section 5.1.1 share the same names to clarify their dependency, but they are defined in the TypeScript Frontend.

5.1.3 Initiating and Updating Synchronization: Frontend Perspective

This part provides an overview of which ExplorViz components are relevant to begin synchronization in ARENA2 or to keep it updated by receiving timely information. For all process descriptions, we assume that ExplorViz and all necessary services are already running.

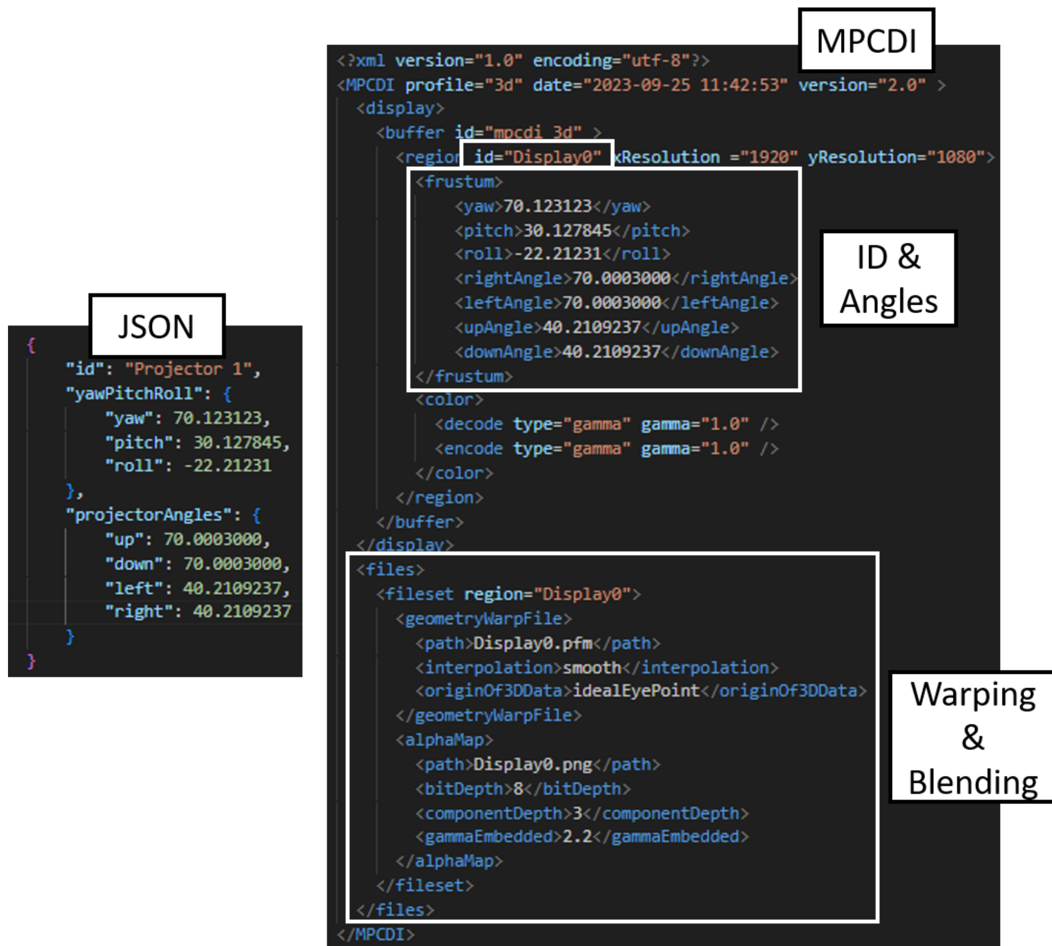

Figure 5.2. Fictive snippet of new configuration format, containing only relevant information of MPCDI file.

Starting the Synchronization

To automatically initiate synchronization, we access the URL `IP-address:4200/landscapes?deviceId=0&roomId=Synchronization&tokenId=landscapeIdentifier` (see *Trigger* processes of Figure 5.3). This URL launches ExplorViz's web interface and establishes synchronization parameters using query parameters (*Setup* processes). The `deviceId` parameter identifies the projector and requests its configurations from the Collaboration Service. These configurations are then structured and converted into usable data formats in the SynchronizationSession. On the other hand, the `roomId` defines the name of the synchronization-specific communication channel between the Frontend and Collaboration Service. In addition to

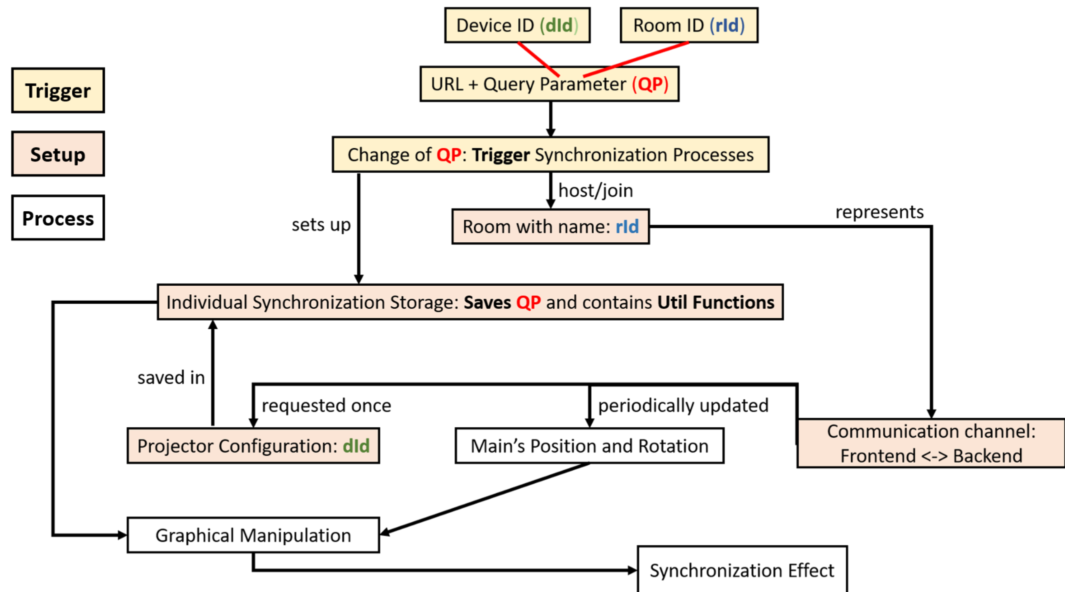

Figure 5.3. Conceptual process oversight in Frontend of the Synchronization Feature.

these two variables, there is the third query parameter `tokenId` that controls the selection of the rendered software landscape. The user can use specific identifier of software landscapes to access them.

If synchronization parameters are detected, ExplorViz executes the task within the Ember component: *SynchronizationStart*. This task is set up by `setUpSynchronizationTask()` and is initiated by the `checkQueryParams()` function. The task is triggered when the query parameters are specified in the URL. At ExplorViz's startup, the `deviceId` value is set to `-99`, and the `roomId` as well as `tokenId` is left blank. Setting up these parameters to different values triggers the following process.

Listing 5.3. Task setup for initiation of synchronization of one ExplorViz instance.

```
1  // Create task to handle async calls on room handling
2  setUpSynchronizationTask = task(async () => {
3      // (1) Set up service attributes
4      this.synchronizationSession.setUpIds(this.args.deviceId, this.args.roomId);
5      // (2) Set token and redirect to visualization space
6      await this.routeToVisualization(this.token);
7      // (3) handle room situation
8      await this.roomTask();
9      // (4) handle which instance is getting synchronized to which
10     await this.synchronizeTask();
```

```
11 });
```

When considering the `setUpSynchronizationTask()` as the primary function for synchronization initiation triggered by setting up query parameters, here is a brief overview of the Ember component `SynchronizationStart`:

- (1) At first, the SynchronizationSession attributes getting set up. Saving identity of the as numbered projector or the control instance (Main) and room name to have a permanent reference to the synchronization communication channel.
- (2) Navigation to the software landscape's rendering space after setting up a landscape token. The landscape token have to be set by using the query parameter `tokenId`, which defined the `value` attribute of the `LandscapeToken` object, which identifies as the definition of the software landscape in the rendering space of ExplorViz.
- (3) If the `deviceId` is set via query parameter, it triggers a Collaboration Service request. By building and sending a payload with data type `InitialSynchronizationPayload`, which contains the two payloads for hosting and joining a room, a synchronization-specified process within the Collaboration Service starts. The Collaboration Service then processes this payload information (for further details see Section 5.1.4), creates a room if needed, builds necessary objects to join this room, and extracts the identified projector's configuration (see Section 5.1.2). A `SynchronizationStartedResponse` containing all the necessary data for synchronization is then sent to the Frontend. After receiving the response, the Frontend sets up the SynchronizationSession to utilize it for the coming activation of synchronization.
- (4) Once the room is joined, ExplorViz's projector instances search for the Main instance. On finding the desired instance, synchronization begins by utilizing the Spectate Feature's activation function (see *Process* in Figure 5.3). Here, projectors act as spectators, with the Main instance being the observed entity. However, we adjust the Main's copied values before setting them by using the information storage, the Ember service `SynchronizationSession`, which contains all utilization functions and information to manipulate the copied Main's graphical information.

Updating Synchronization

The Synchronization Feature builds upon the Spectate Feature's methodology, triggering a function at regular intervals to update the projectors' current graphical information. This ensures all synchronized projections remain up-to-date with the Main's current state. When synchronization starts, the Ember service *SynchronizationService* periodically triggers copying processes. After copying position and quaternion data, the frustum's shape and quaternion of these virtual cameras are adjusted to synchronize the five projectors in ARENA2 (see Section 5.3).

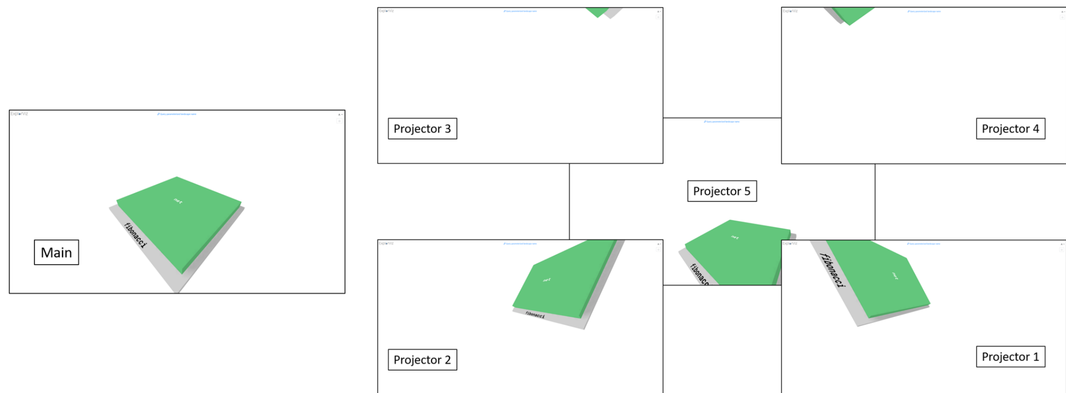

Figure 5.4. Skewed display of Main's monitor visual while Synchronization Feature is active.

To the user, the Main's software landscape appears like any other unmodified ExplorViz instance. However, the synchronized instances might appear skewed on standard monitors, but in ARENA2's dome, they display their unique portions of the Main's rendered space like we see the part on the Main's visual (see Figure 5.4).

5.1.4 Initiating Synchronization: Collaboration Service Perspective

This following process of the section refers to the *Communication Channel* in Figure 5.3. When the `InitialSynchronizationPayload` initially arrives, the synchronization-specific path of the Java based class `RoomResource` triggers. The function `startSynchronization(body: InitialSynchronizationPayload)` creates the mentioned `SynchronizationStartedResponse`, which is sent to the Frontend. By retrieving the `InitialRoomPayload` we can access the query parameter `roomId` (see Section 5.1.3) and check for the need of hosting the room. If it is not existent, the adapted `addRoom(body: InitialRoomPayload)` function looks for the room specified by the query parameter, and triggers its creation if needed. Before the Synchronization Feature, the Collaboration Service does not consider creating a room with a specific room name. Thus, we overload the `RoomService`'s function `createRoom()` by adding a parameter to specify the resulting room's name.

After a room is hosted, the existing infrastructure of the Collaboration Service utilizes the resulting `RoomCreatedResponse` to generate a `LobbyJoinedResponse`. Generally, the joining response includes a ticket that serves as an entry card to manage the room-joining process. This thesis does not modify the existing room-joining process.

Due to reusage of the Spectate Feature, the to-be-synchronized projectors need the Main instance to be connected before they activate the synchronization to the Main. Thus, we let the instances identified as projector wait until the Main is connected. Setting up a small Java based service `SynchronizationService` serves as indicator for the Main's connection.

The final step in constructing the `SynchronizationStartedResponse` involves extracting

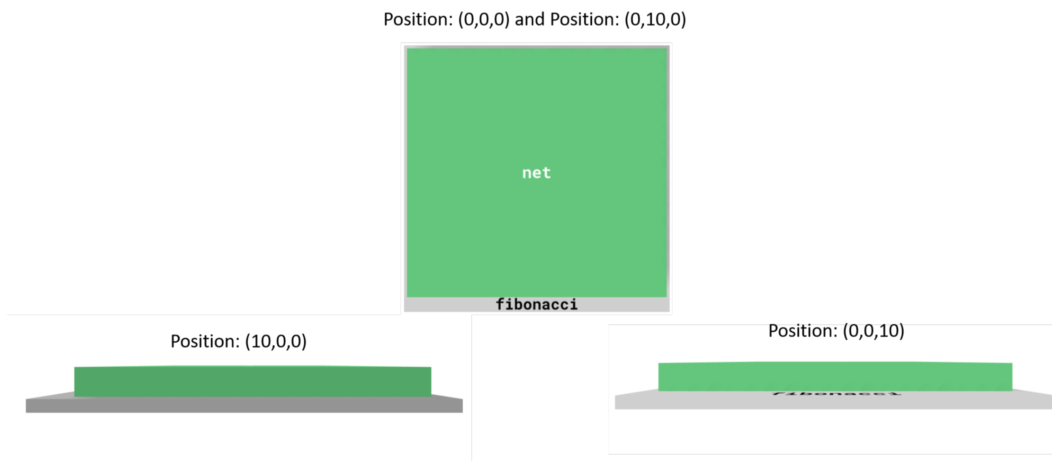

Figure 5.5. Changing the starting position to facilitate graphical processing research.

the identified projector's configuration, as referenced in Section 5.1.2. This extraction is carried out to manipulate the underlying virtual camera of the projection in the Frontend part, thereby achieving the synchronization effect we are aiming for.

5.2 Graphical Prework: Monitor-Solution

Like we mentioned in Chapter 4, we need to do some research for understanding the graphical processing itself using Three.js (referring to Section 3.3.1). In this testing phase, we implemented an initial solution for synchronizing two instances and after that extended the resulted prototype to synchronize four instances. The following solution is built to work on a monitor, separating the visual of the monitor in four distinct spaces. This explanation is not only important because of the definition of the goals, but also to show the difficulties using the Monitor-Solution to find a working procedure for the scenario of ARENA2.

The following section is divided in two approaches to find the Monitor-Solution. First experimenting with the attributes of the PerspectiveCamera of Three.js which is creating the projection matrix and second using the projection matrix itself to successfully synchronizing four instances, what we initially called as the Monitor-Solution. This satisfies goals G2.1 (Section 2.2.1) and G2.2 (Section 2.2.2).

5.2.1 Camera Attempt

Given the reliance on Three.js for rendering the software landscape and the Spectate Feature, it is the logical next step to utilize the provided Three.js PerspectiveCamera setup (Section 3.3.1) for the Synchronization Feature. Due to the nature of this thesis and our initial

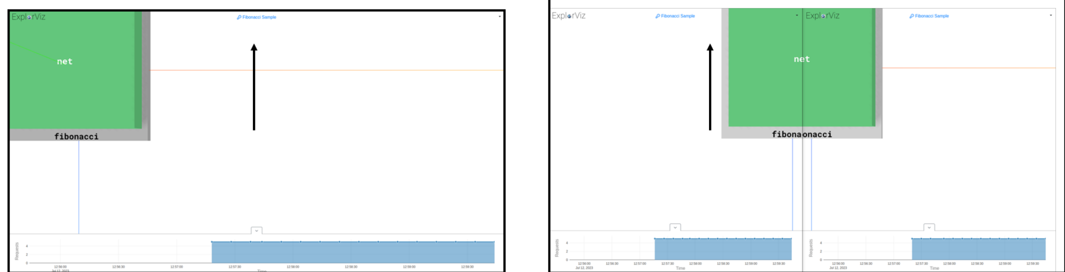

Figure 5.6. Testing one coordinate synchronization (left: Main; right: Two spectator instances).

understanding of three-dimensional modeling, we refrain from using PerspectiveCameras. However, we do share our insights from exploring the combination of Three.js's camera type and ExplorViz's setup.

To simplify the scenario, we adjusted the software landscape's initial position to provide a two-dimensional view, as shown at the top of Figure 5.5. This already indicates challenges in interpreting the camera's position when considering the effect within the editor provided by Three.js (compare Figure 3.10): At first the initial position of the software landscape at position (0,0,0) seems to be a two-dimensional top-view. Changing the position to (0,10,0) is not affecting the initial view, but the effect when first seeing the rendered software landscape. It starts at the same point on the monitor but visualizes an effect of being further away. Then rushing closer until the software landscape reaches the same size as if it would be at (0,0,0).

But changing the position to (10,0,0) shows us a rotation of the initial starting position of the software landscape, which we detect by interpreting the position of the name of the landscape: "fibonacci". Not seen on the bottom left in Figure 5.5, the name is presented on the left gray part. The landscape is rotated to the axis on which it initially got moved. Changing the position without deactivating the starting animation by moving the software landscape to the middle of the rendering space, needs to be considered when working with ExplorViz's camera setup.

With the idea to use the Spectate Feature to change the position permanently when activating the spectating, we start with basal vertical synchronization of two instances. This only tracks the vertical movement (see Figure 5.6) and holds the horizontal movement static. This gives us practical insights how to set up the position coordinates or quaternion data of the camera. The position change is like this:

Listing 5.4. Shifting the projection.

```
1 const leftVector = new THREE.Vector3(-1, 0, 0).applyQuaternion(Main.quaternion);
2 this.localUser.camera.position = Main.position
3         .clone()
4         .add(leftVector.multiplyScalar(1.6));
```

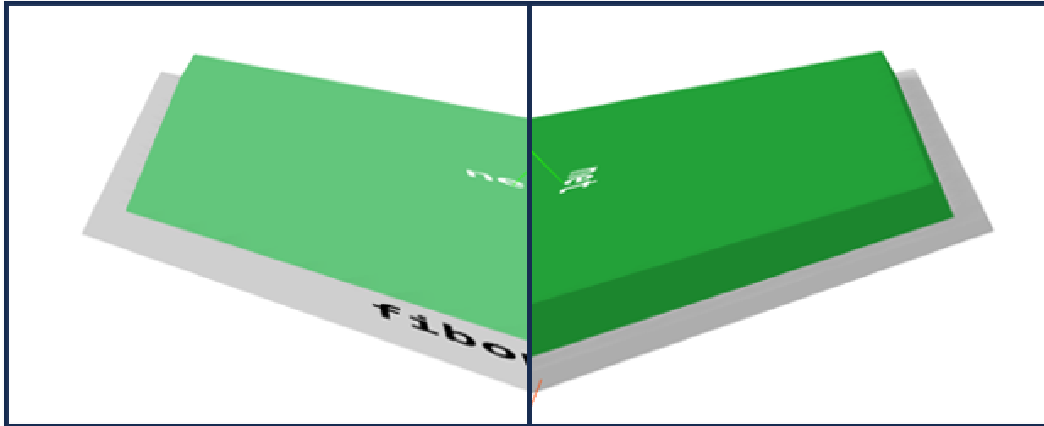

Figure 5.7. Checking quaternion with one coordinate synchronization (left and right: Spectator instances).

Here, `leftVector` is representing a three-dimensional vector created with the values (-1, 0, 0). This suggests that the initial direction of the vector is pointing leftwards along the x-axis. The `applyQuaternion(quaternion: THREE.Quaternion)` method then rotates this vector based on a quaternion. So, `leftVector` starts as a left-pointing vector and is then rotated given by the Main's quaternion. The overall effect is that the position is shifted to the left by a distance of 1.6 units (after taking into account the rotation applied by the quaternion). The same effect, but negative values, is done to the left part of Figure 5.7.

After that we need to work with the synchronization of the quaternion too (see Figure 5.7), not only because position and quaternion are different in their functionality (see Section 3.5) but also just copying the quaternion does not do the correct manipulation we seek for. After experimenting with the change of position and quaternion, we think that the camera is somewhere placed, and we can just sway the frustum to a point in space. Following in tests with the function `Object3D.lookAt(vector3: THREE.Vector3(x: number, y: number, z: number))` of the PerspectiveCamera, which directs the camera's frustum center to a point in space. But considering the bidirectional view of a human and the interpreted three-dimensional movement of the software landscape (looking at Figure 5.8), `Object3D.lookAt(vector3)` does not provide the solution. For example, we sway the camera to the middle of the left border and the middle of the camera's frustum. Then we are just looking at the position, and still not aligning the different rotations of the software landscape, when moving it to different sides of the frustum.

In the end, this experiment brings us to the idea of working with the frustum's visual space. Thinking about shrinking the frustum's space and changing its shape, while shifting the center of the frustum to the side.

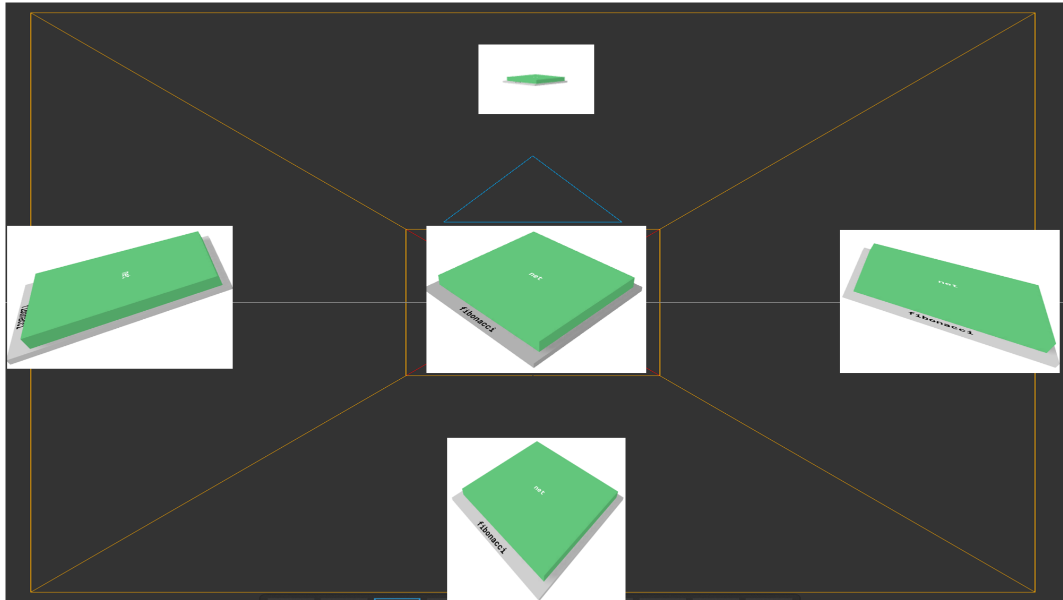

Figure 5.8. Rotating the initial (5,5,5) positioned software landscape to borders of the camera's frustum. This background of the figure is retrieved from Three.js's editor [*three.js editor*].

5.2.2 Matrix Approach

The Matrix approach starts from the idea of adjusting and reducing the canvas size to achieve equal portions of an ExplorViz instance's visual space. In exploring this concept, we identify several matrices instrumental in scene rendering. Among them, the camera's projection matrix —detailed further in Section 3.5.5— offers insights into the scene's projection via the camera's frustum definition.

In examining how Three.js implements the projection matrix, we discover the class `THREE.Matrix4`, representing a 4x4 matrix, which is also conceptually used in the DEV (see Section 3.2.2). `THREE.Matrix4` includes a built-in function, `makePerspective(left: number , right: number, top: number, bottom: number, near: number, far: number)`, which configures vertical and horizontal borders using the specified coordinate values. This function replaces the camera's current projection matrix with one that maps the specified frustum onto a coordinate cube, where all values range between -1 and 1. This approach often simulates the perspective effect of a three-dimensional object in space, aligning with our objectives for the Monitor-Solution [*WebGL model view projection - Web APIs | MDN*].

Since the clipping planes resulting from coordinates, we need to calculate the vertical and horizontal field of view to determine the coordinates for left, right, top, and bottom. For this, we define three variables:

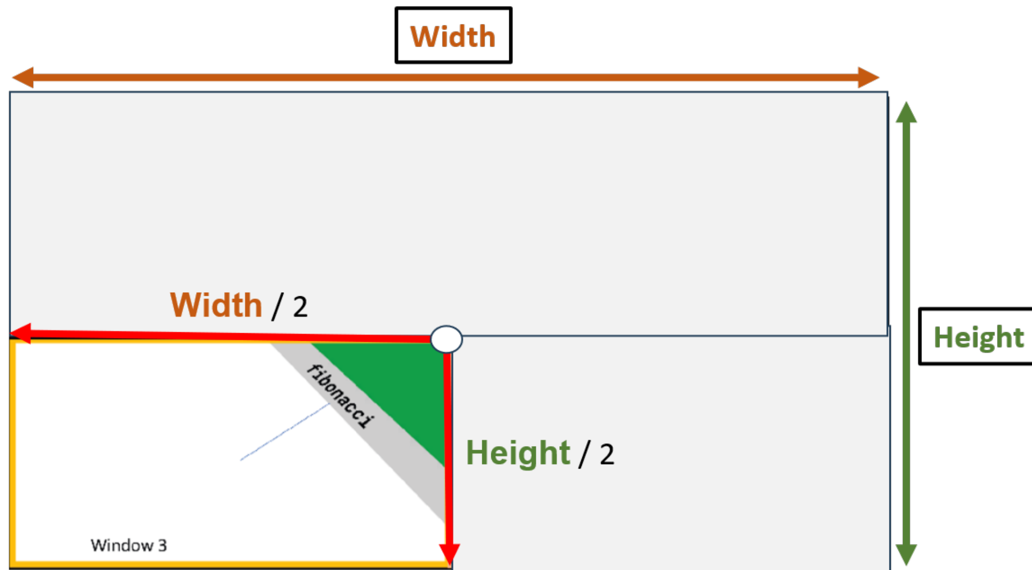

Figure 5.9. The left bottom instance of Monitor-Solution calculated using Main's visual space's width and height.

Listing 5.5. Setting up necessary variables for frustum manipulation.

```
1 const tanFOV = Math.tan(((Math.PI / 180) * this.localUser.camera.fov) / 2);
2 const height = tanFOV * this.localUser.camera.near;
3 const width = height * this.localUser.camera.aspect;
```

▷ `tanFov`: Converts the field of view of the camera from degree to radians and computes the tangent of this angle. It is divided by 2 because it should be the half of the vertical field of view.

▷ `height`: Gives the height of the near clipping plane from the center of the camera's view to the top (or bottom).

▷ `width`: Is the width of the near clipping plane, which is determined by multiplying the height by the camera aspect ratio, ensuring the correct proportions of the rendered scene.

To set up the four synchronized ExplorViz instances, all four instances need their corresponding configuration of the function `makePerspective(left, right, top, bottom, near, far)`. To clarify the procedure, we explain it with a code snippet for the bottom left instance (see Figure 5.9):

Listing 5.6. Manipulating frustum by accessing camera's projection matrix and setting coordinates for side clipping planes as well as near and far clipping plane.

```
this.localUser.camera.projectionMatrix.makePerspective(
    -width / 2, // left
    0, // right
    0, // top
    -height / 2, // bottom
    this.localUser.camera.near / 2,
    this.localUser.camera.far / 2);
```

By setting non-zero values for the left and bottom coordinates of the camera's frustum, this function call defines the bottom left quarter of the initial rendered space. The bottom clipping plane is set at half the defined height below the center, and the left clipping plane is set at half the defined width to the left of the center. Using this logic, we can create the three other partial visuals of the software landscape.

For the top left view, the top-coordinate is set to `height / 2` and the bottom-coordinate to `0`. If we set the bottom-coordinate to `height / 2` instead of the top-coordinate, the result is a view in the top left quarter, but with a flipped orientation. This flipping occurs because we are inverting the top and bottom values of the frustum, but the projection matrix still interprets the scene using the original coordinate system, leading to a mirrored interpretation of the scene.

Considering these specifications, we create the top right visual by setting the right coordinate to `width / 2` and the top coordinate to `height / 2`. To present the bottom right visual, we change the last definition and swap the values for the top and bottom coordinates and multiplying it by `-1` resulting in `-height / 2`.

After setting up the camera's frustum, we then need to consider the issue that we are copying the position and quaternion of the Main's camera object using the Spectate Feature. This last hurdle can be solved by considering the definition of the underlying view matrix. The copied values are changing the view matrix of the camera, which is different from the projection matrix (see sections for view matrix: Section 3.5.6 and projection matrix: Section 3.5.5) and updated without manually triggering an update. After that, we get a fully synchronized picture of four distinct instances presenting their own specific partial view of the whole Main's image: The Monitor-Solution (see Figure 5.10).

Attempt: Decompose Graphical Attributes

Achieving a solution with the projection matrix, we are now in a position to seek further refinements, possibly through extracting only the necessary camera attributes of the projection matrix. However, with time constraints to implement the Synchronization Feature in ARENA2, we only test the decomposition of relevant variables from the resultant projection matrix. The `THREE.Matrix4.decompose(position: THREE.Vector3, quaternion: THREE.Quaternion, scale: THREE.Vector3)` seem like a promising method for simplifying

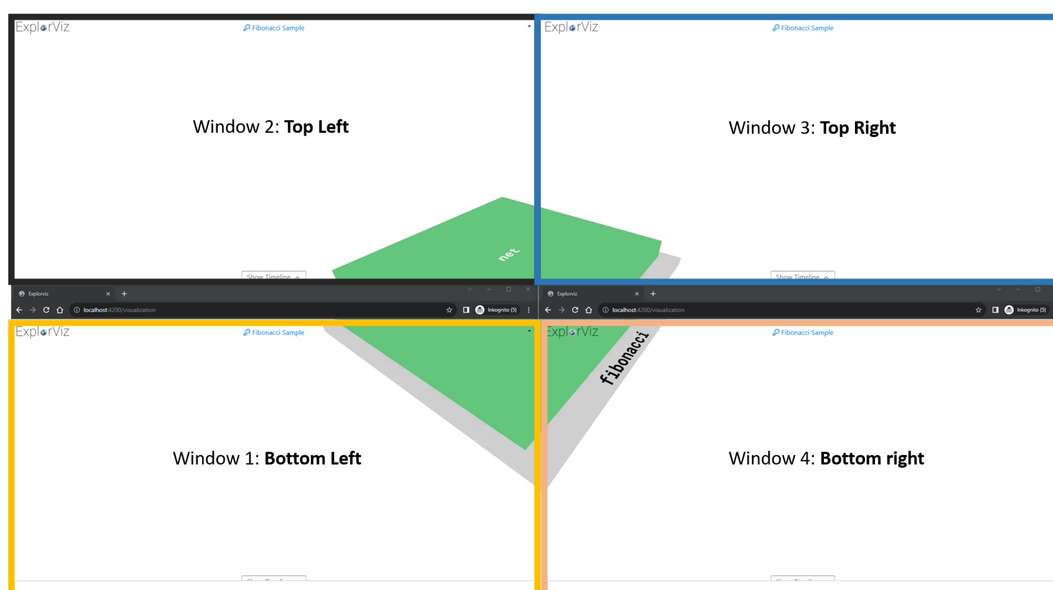

Figure 5.10. Synchronization of four browser windows displaying one visual of an ExplorViz instance on one monitor.

this task, providing a comparable effect using Three.js's camera and the given Matrix approach. However, our initial attempts revealed that the transformed projection matrix is not easily decomposable. The resulting position, quaternion, and scale produces a distorted and incorrectly scaled software landscape. Nevertheless, this concept of extracting information as future work in a camera approach remains a promising route for further refinement.

5.3 Graphical Processing: ARENA2

The dome of ARENA2 is illuminated by five projectors. This brings us to the second extension of the prototype, initially extended by the prework (refer to Section 5.2). For the Monitor-Solution, we can use four identically sized and symmetrically placed visual spaces to synchronize the ExplorViz instances. The challenge is not just adding an instance, but accommodating the hemispherical shape of the new projection surface and the distribution of the projectors (refer to Figure 5.11). Hereafter, we are referring to a projector's projection as "projection X", where X ranges from 1 to 5, denoting the position of the projection in the dome. This section satisfies the Goals G2.3 (see Section 2.2.3) and G4.1 (see Section 2.4.1).

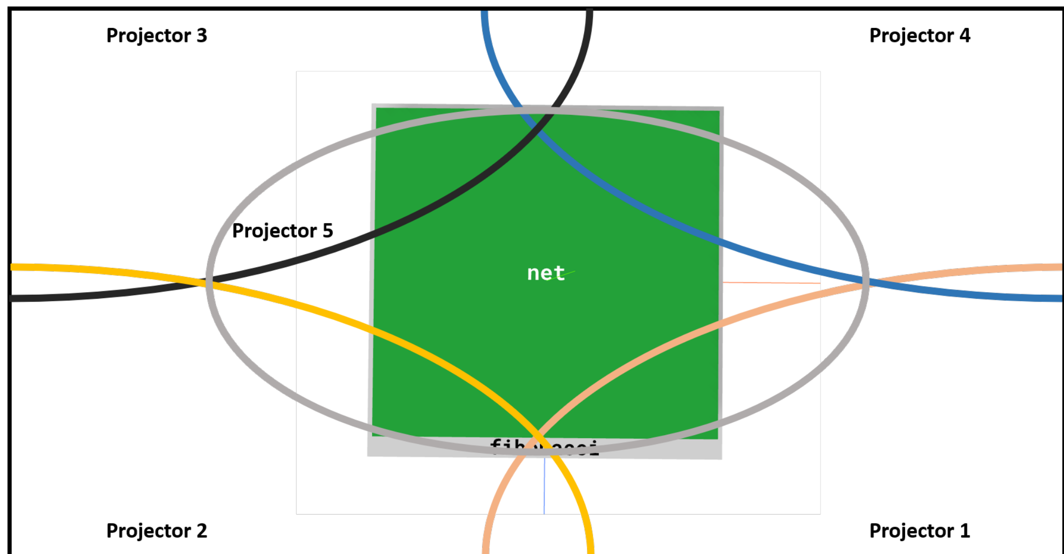

Figure 5.11. Conceptual view of ARENA2's projection distribution.

5.3.1 Testing Monitor-Solution

First, we try to adapt the Monitor-Solution using the projection matrix to determine the visual space of the projector's illumination. For this, we manually change the `makePerspective (left, right, top, bottom, near, far)` specifications for two projections, to shift and rotate the partial image that it does not stand out, we move the software landscape from one projection to another one (see Section 5.2 for better understanding of `makePerspective(left , right, top, bottom, near, far)`).

We manually adjust the parameters in `makePerspective(left, right, top, bottom, near, far)`, specifically for either the first and forth projections or the first and second projections. This is done by not only shifting the fragment of the image using `makePerspective (left, right, top, bottom, near, far)`, but also by rotating it, so it integrates seamlessly, when crossing from one projection to another. For this, we consider ARENA2's specifications provided by the MPCDI file (see Section 3.4), which is used for the Synchronization Feature of the DEV (see also Section 3.2.2). We define the yaw, pitch, and roll rotations of the MPCDI file as Euler's angles and apply them to the projection by multiplying the projection's quaternion. Using the camera's quaternion while manipulating the projection via changed projection matrix, resulting in vanishing or strongly shifted projections that are not close to be correctly rotated or placed in position. We test the perspective manipulation as well as the rotation by entering hard-coded values, but not achieving a flawless synchronization of at least two projections (see Figure 5.12).

After that, we decide to start at the very beginning, because we can not just change

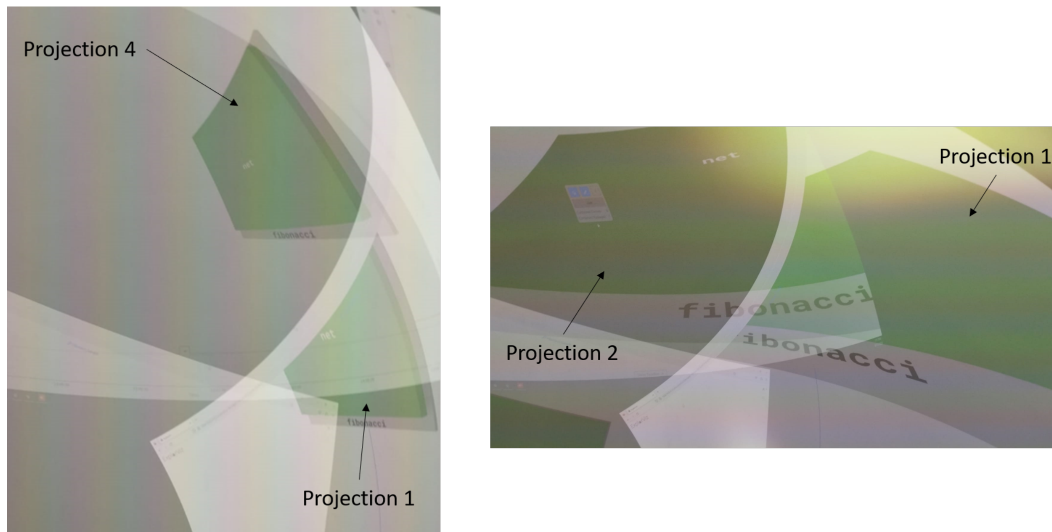

Figure 5.12. Best outcomes by manually adapting quaternion and perspective of projection.

the Monitor-Solution to work on a dome with not equally sized projections. We remove the perspective manipulation by `makePerspective(left, right, top, bottom, near, far)` and start working on the initial not-manipulated picture of ExplorViz's visualization, and incrementally add the specifications of the working MPCDI setting of ARENA2. To avoid redundancy, we note that applying the MPCDI setting to the camera's attributes once again results in effects that require significant time to fully understand the advanced three-dimensional manipulation and the way ExplorViz or Three.js's camera types and controls handles graphical manipulation. Which leads us to focus on the projection matrix.

5.3.2 Frustum Manipulation

Each ExplorViz instance's projection is represented by a camera object in Three.js. This camera contains both a projection matrix and a view matrix, which we adjust to synchronize the five distinct projections in ARENA2. We use the Spectate Feature to align the projector's view matrix with the Main's by copying its position and quaternion attributes and modifying the projection matrix each time.

The following function `setUpCamera()` is used for this section and subsections to discuss the different tasks, which need to be solved to achieve the synchronization:

Listing 5.7. Setting up projector configuration using camera's projectionmatrix.

```
1 setUpCamera() {
2     // Fov and Aspect Ratio
3     this.localUser.camera.projectionMatrix.copy(
```

```
4          new THREE.Matrix4().makePerspective(
5              -Math.tan(this.projectorAngles.left * DEG2RAD)
6              * this.localUser.camera.near,
7              Math.tan(this.projectorAngles.right * DEG2RAD)
8              * this.localUser.camera.near,
9              Math.tan(this.projectorAngles.down * DEG2RAD)
10             * this.localUser.camera.near,
11             -Math.tan(this.projectorAngles.up * DEG2RAD)
12             * this.localUser.camera.near,
13             this.localUser.camera.near,
14             this.localUser.camera.far
15         )
16     );
17     // Yaw, Pitch and Roll
18     this.localUser.camera.projectionMatrix.multiply(
19         new THREE.Matrix4().makeRotationFromQuaternion(
20             this.projectorQuaternion?.quaternion
21         ));
22     // Projectionshift: Dome Tilt
23     this.localUser.camera.projectionMatrix.multiply(
24         new THREE.Matrix4().makeRotationFromQuaternion(
25             this.getDomeTiltQuaternion()
26         ));
27 }
```

`setUpCamera()` is every time called when, the Main's position and quaternion data is copied.

Shape

Initially, we apply the angles for the frustum's aspect ratio using a function from the SynchronizationSession, which provides essential utilities and retains the MPCDI data.

First setting up left, right, up, and down angles to define the frustum's shape of the projector. In difference to before, we do not want to shift the projection anymore, but shaping it, resulting in no non-zero values for every clippling-plane-determined coordinate. We multiply by the value of the camera's near clipping plane to resize the projection. After setting up a new matrix using the function `makePerspective(left, right, top, bottom, near, far)` and `rightAngle`, `leftAngle`, `upAngle` and `downAngle` of the MPCDI configuration, we copy it and effectively overriding the camera's projection matrix of the projectors.

Quaternion

After that, we apply successively two quaternions to (I) first achieve the synchronization effect - creating one seamless picture by putting the five parts together - and (II) second lifting the synchronized projection up considering the tilt of the dome in ARENA2.

Our plan is to rotate the projection matrix, which is not affecting the view matrix. Although, the view matrix consists of positional and rotational data, the rotation of the projection matrix is a separate transformation and does not affect the view matrix. Applying a rotation to post-projected points is no common procedure in the graphical manipulation universe (according to discussion entries in *Mathematics Stack Exchange*) and can result in unpredictable results. But, in our scenario, it is providing the correct outcome.

The procedure for (I) is as following:

▻ **(I)** For the first part, we use Three.js's provided function for `THREE.Matrix4` objects `makeRotationFromQuaternion(quaternion: THREE.Quaternion)` to apply the yaw, pitch and roll angles (as `THREE.Euler(x: number, y: number, z: number)`) object described in Section 5.1), which are provided by the MPCDI file (explained in Section 5.1.2). To achieve the correct effect of the Euler's angles considering the synchronization of the five image parts (referring to: Projection placement in three-dimensional space Figure 4.7, and two-dimensional distribution Figure 5.11), we need to consider three topics:

  ▻ (I.a) Order of the applied rotations.

  ▻ (I.b) The assignment of yaw, pitch, and roll to the parameter of `THREE.Euler(x, y, z)`.

  ▻ (I.c) The specific interpretation of these Euler's angles for ARENA2's environment.

  These three topics can be different for every system, because of the interaction of the environment, used library, or framework. To overcome the following issues, we collaborate with the GEOMAR's scientists [*GEOMAR Helmholtz Centre for Ocean Research Kiel*]. The definition of the solution is as following:

  ▻ (I.a) Avoiding time intensive theoretical work finding the correct rotation order, we iterate over all possible permutations ('XYZ', 'XZY', 'YXZ', 'YZX', 'ZXY', 'ZYX').

  ▻ (I.b) Following quaternion tests on the software landscape (like we show via Three.js's editor Section 3.5.4), we assign the pitch rotation to the X axis, the yaw rotation to the Y axis, and the roll rotation to the Z axis.

  ▻ (I.c) For ARENA2, we need to take a positively or negatively interpretation of the axis rotation into count. By extending (I.a) we can also iterate over all permutations with this binary interpretation of each axis, which increases the initial 6 to 48 unique permutations to test. Considering this new iteration setting, we find the order 'ZXY' and negatively interpreting the angle of the rotation around the X axis, approving the synchronization.

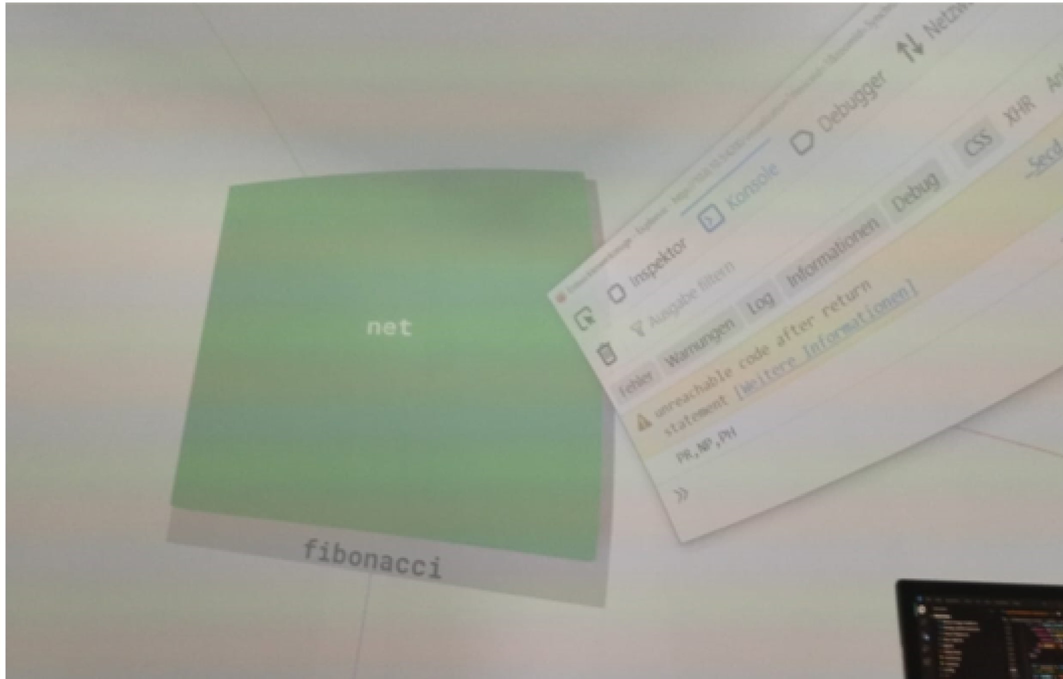

Figure 5.13. Camera picture of the correct iteration considering order of rotations applied, assignment to the rotational axes, and interpretation of rotation angles.

These three considerations lead us to the newly interpreted MPCDI angles. First apply positive roll rotation, then negative pitch rotation, and at the end, again positive, yaw rotation. Figure 5.13 displays the correct iteration and a console log with the value: PR stands for positive roll, NP stands for negative pitch and PH stands for positive heading or in our scenario positive yaw.

▷ **(II)** The second part of the quaternially manipulated frustum is using the quaternion function `makeRotationFromQuaternion(quaternion: THREE.Quaternion)`, but only applying a pitch rotation. Considering the tilt angle of the dome being 21° (see Figure 5.1), we end up positively interpreted rotating around the X axis by `90 - 21`°, to shift the starting point of the software landscape to the center of the dome. For better usability, we shift the starting point to `45`°, which does not afford to tilt the head completely back to see the software landscape. This shift is also needed in general, because, after the first quaternion to create the synchronization setting, the picture of the software landscape is displayed at the bottom border between projection 1 and 2 (see Figure 5.14 for projection shift and Figure 5.11 for border between projection 1 and 2), which would limit us to use the whole projection space of the dome because the software landscape vanishes at the bottom when moving it.

Keeping this order of applying these two quaternions is important, to achieve the synchronized image (see I) and after that moving the seamless picture up to a comfortable position (see II).

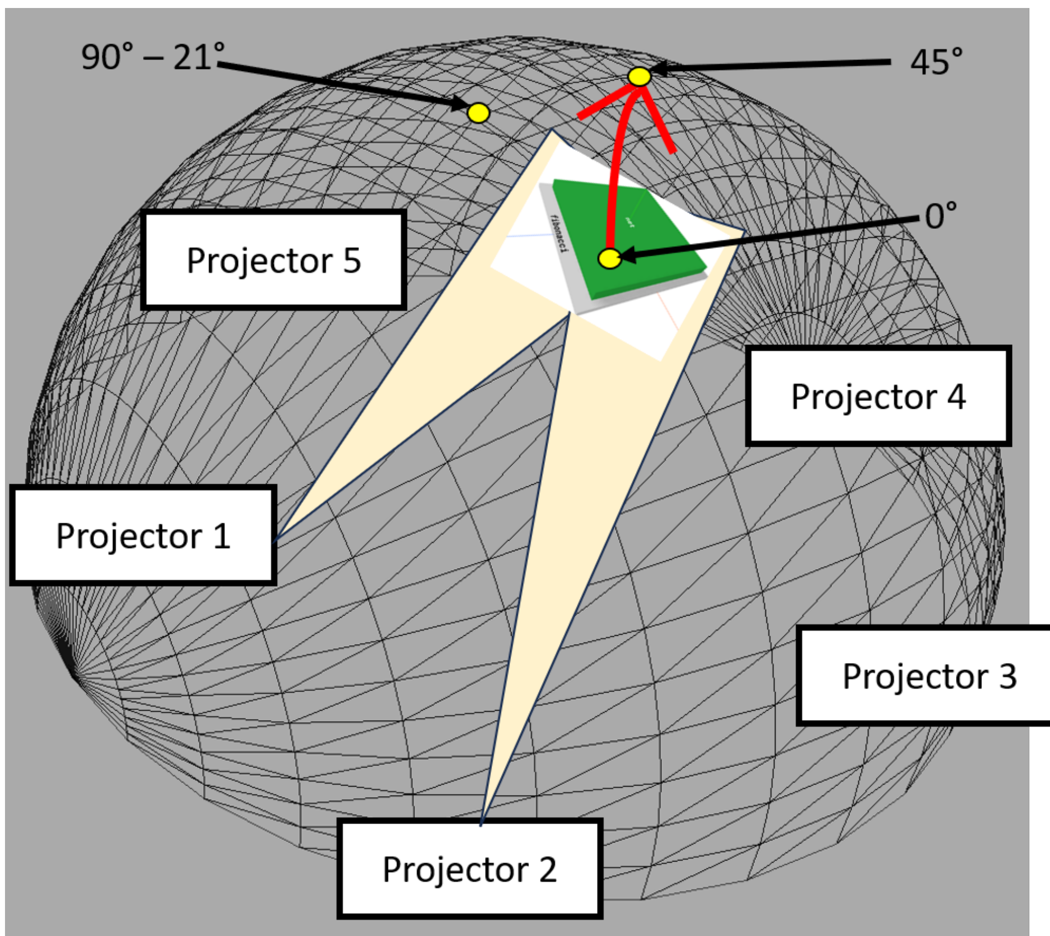

Figure 5.14. Moving synchronized picture from the edge of projection 1 and 2 to the 45° elevated position. This figure is retrieved from Three.js's editor [*three.js editor*].

5.4 ARENA2 Specific

5.4.1 CORS Workaround

The Synchronization Feature does not provide CORS specific management. We solve it by hard code the IP-adress of the Main directly in the file for environmental variables of the Frontend repository of ExplorViz. The user of the Synchronization Feature have to consider this issue when implementing the Synchronization Feature in other environments than the ARENA2.

5.4.2 Script Based Start

The computer setup for ARENA2 consists of devices running Microsoft Windows. To set up the necessary resources for synchronization, we execute a batch script. This action initiates a PowerShell script, which runs with administrative privileges. The script starts the Collaboration Service, launches Docker Desktop, and then initiates the corresponding Docker container [*Docker: Accelerated Container Application Development*]. Finally, the script builds the Ember Frontend and opens a web browser, navigating to the URL to set up all query parameters and initiate ExplorViz.

In order for ExplorViz and its Synchronization Feature to function within ARENA2, the initial setup of the Main requires setting a room name and a software landscape. The declaration of the software landscape is done using the identifier, referred to as `tokenId` in Section 5.1.3, as a query parameter. It is essential that all following projector instances access the same URL with identical values for the query parameters `roomId` and `tokenId`, but still consist of unique identifier for the device (query parameter: `deviceId`). If they not all access the same room and software landscape, the Main instance cannot fully control the composed image in the dome. If a projector does not use its unique identifier, it may be incorrectly configured or might not receive any configuration at all.

5.4.3 Hide Timeline

We consider the specific needs of the environment. The composed projection represents five ExplorViz instances. Thus, when entering the synchronization in ARENA2, we hide ExplorViz's timeline. Using five instances means also displaying five timelines, which are not equally complete projected considering the distribution of projection Figure 5.11.

Chapter 6

Evaluation

6.1 Motivation and Research Questions

The evaluation section outlines aspects and questions that need to be addressed when testing the implemented ExplorViz in ARENA2. The primary consideration is to understand the aims of this implementation. How will it contribute to existing knowledge or improve the usage of ExplorViz, and which advantages and disadvantages does the combination of ExplorViz and ARENA2 provide?

To check the advantages and disadvantages of using ExplorViz in ARENA2's environment, we invite participants to engage with ExplorViz in ARENA2. This approach allows participants to experience the environment firsthand, thus facilitating a better understanding of the advantages and disadvantages that ARENA2 offers when interacting with ExplorViz. According to the meeting room experience in ARENA2 and the collaborative factor of ExplorViz we invite probands to evaluate in groups of two.

When considering what distinguishes ARENA2 from other visualization spaces, four key areas of investigation are important: *User Experience*, *Affection*, *Cognition*, and *Collaboration*. By asking the participants for their opinions on ExplorViz and ARENA2 separately, we also invite them to share their views on the integration of these two concepts. These topics result in the following research questions:

- **1. Usability: User Experience**
  - 1.1 What advantages does the implementation of ExplorViz in ARENA2 offer in terms of user experience?
  - 1.2 Does the combination of ExplorViz and ARENA2 offer advantages over using ExplorViz on a conventional monitor?
- **2. Cognitive support:**
  - 2.1 Do participants feel that ExplorViz aids in understanding software landscapes?
  - 2.2 Do participants feel that ARENA2 aids in understanding of concepts?
  - 2.3 How does the combination of ExplorViz and ARENA2 enhance the process of understanding a software landscape, according to the participants?
- **3. Enhance Collaboration:**

- ▻ 3.1 Is ExplorViz supporting collaboration?
- ▻ 3.2 Is ARENA2 support collaboration?
- ▻ 3.3 Does combine ExplorViz and ARENA2 enhance collaboration compared to using them separately?
- ▻ 3.4 How many software developers can collaborate at the same time while maintaining an equal user experience?

6.2 Participants

It is important to think about what a proband needs to do in this evaluation. For analyzing software landscapes and solving software architecture tasks, we need probands with software development experience. It is essential they have an idea of foundational concepts like classes, packages, and function calls. Otherwise, they are not able to work with the software landscape and solve the tasks accordingly, hurting the validity and reliability of this evaluation study.

In summary, 10 participants completed the evaluation in groups of two. Most of them are either studying computer science or are employed in scientific roles in computer sciences. As such, their highest educational achievements range from Abitur to Bachelor's and Master's degrees. Given the focus on educational or scientific computing, 40 percent are already familiar with ExplorViz's capabilities. We also asked participants to estimate their software development experience, to assess their capability to analyze the software landscapes in this study. On a scale from 1 (no experience) to 10 (many years of experience as a software developer), the average score is approximately 5.1. We consider this to be sufficient experience for the purposes of this study.

6.3 Material

6.3.1 Digital Survey

We use a standardized way to run through the evaluation, by using a digital survey that runs locally on two computers. This survey uses a locally started python program, which's content is displayed in a web browser using the python library *Flask* [*Welcome to Flask — Flask Documentation (2.3.x)*]. There are two versions of the program, the first consists of 25 pages and contains all parts of the evaluation. The second version consists of 10 pages with slightly adapted content, because in the collaboration phase of the evaluation, the probands require only one survey. These pages provide instructions to the evaluation process and questions to demographics, the software landscape or their opinion to the project. The pages for instructions and the introduction to ExplorViz are pictures placed on HTML pages. The questions are developed with HTML and CSS.

At last, all responses are saved in separate text files.

6.3.2 Tasks

Inspired by Krause-Glau et al. 2022a, we create three types of tasks: *Metric*, *Oversee* and *Understanding*. For a detailed view, we encourage reading the appendix.

- **Metric Tasks:** These tasks ask the participants to quantify elements like classes or packages using only the software landscape visualized in ARENA2.
- **Oversee Tasks:** For these tasks, we encourage the participants to take in more macroscopic perspective of the software landscape. One of these tasks, for example, requires comparing the structures of components that are distant from one another.
- **Understanding Tasks:** These tasks require the participants to interpret the software landscape at the most detailed level. For instance, locating the sources of a bug in the application would require a deep dive into individual components while also considering their potential interactions with other parts of the software landscape.

Our aim is to engage the participants more deeply with the ExplorViz concept. During the collaboration phase, we plan to encourage them to work together to find a solution that satisfies both parties.

6.3.3 Evaluation Questions

We define four topics of evaluation questions: *General Usability*, *Collaboration*, *Cognition* and *Affect*. All open-text questions begin with a statement. Participants are asked to rate these statements on a Likert scale (see value coding Table 6.1) [Bortz and Döring 2006]. The questions to the general usability consists only of those statements, and we ask for no

Table 6.1. Likert scale values.

Description	Value
I disagree	1
I tend to disagree	2
I tend to agree	3
I agree	4

open-text responses.

- **General Usability Questions:** With these questions, we seek participants' opinions on the usability of ExplorViz within ARENA2's environment. We start by asking about their general thoughts regarding navigation and interaction with the software landscape. This is followed by questions concerning the advantages or the disadvantages of ARENA2's display compared to standard monitors. We conclude by asking for their impressions of interacting with ExplorViz in ARENA2 setting, as opposed to using a conventional monitor.

▷ **Collaboration Questions:** We begin with questions that address participants' opinions on collaboration within ExplorViz and within ARENA2 as standalone environments. We then ask how ARENA2 enhances the collaborative features of ExplorViz. Given the expansive visualization space offered by ARENA2, we also ask about the number of people who can effectively collaborate at the same time using ExplorViz in this setting.

▷ **Cognition Questions:** We start this section with questions aiming at gathering participants' opinions on understanding software within both ExplorViz and ARENA2 as separate platforms. Following that, we inquire about how ARENA2 complements ExplorViz in enhancing the program comprehension under use of software landscapes.

▷ **Affect Questions:** We wrap up with a simple question: Do the participants enjoy using ExplorViz in ARENA2? We also give them the opportunity to specify what aspects they find enjoyable while working on this evaluation.

All statements and questions are displayed in the appendix, and are matched by the identifier provided in the first column of the tables.

6.4 Procedure

The evaluation starts with meeting the group of two in front of the GEOMAR. After leading them to ARENA2, we start with evaluation by asking for initial questions to the evaluation. After that, both probands work individually on two computers and run through the greetings pages of the survey. After responding to demographic items, they are instructed to wait for each other, to then work together through the introduction into ExplorViz and answering the questions to the software landscape. For the introduction, we use a different but easy to understand and to oversee software landscape (Figure 5.5), which needs to be changed before the probands start working on the tasks. In the second part of the collaborative phase, the probands are encouraged to dive into the more complex software landscape (Figure 3.2). While solving the tasks, they can, at any time, update their understanding of ExplorViz's features by pressing an instructed key shortcut to access a version of the survey, which only contains the introduction part. After the collaborative phase, the participants are directed to separate and work independently. This approach ensures that each individual forms their own perspective and assessment of the experience with ExplorViz within the ARENA2's environment.

6.5 Results

In the following section, we discuss the results from the evaluation phase of our study. The tasks are used to provide a cognitive-engaging scenario to dive into the software landscape and the setup in ARENA2. Therefor, we are not reporting a review to answer correctness.

6.5.1 Usability: User Experience

▷ **1.1** What advantages does the implementation of ExplorViz in ARENA2 offer in terms of user experience?

The participants find the display and interaction of a software landscape in the dome enjoyable due to its high resolution, expansive visualization space, generous personal workspace, the immersive experience of mentally diving into the projection, and the unique aspect of looking up to extract information.

These quality findings are supported by the average ratings from the first question on general usability and the enjoyment report, participants experienced when interacting with and seeing the ExplorViz implementation in ARENA2. With an average rating of 3.5, participants tend to find their interaction with the software landscape to be smooth and intuitive. An average score of 3.6 indicates that participants tend to enjoy working with the combination of ExplorViz and ARENA2.

▷ **1.2** Does the combination of ExplorViz and ARENA2 offer advantages over using ExplorViz on a conventional monitor?

For the second and third questions in the general usability section, participants rate the advantages of projecting in the dome compared to a conventional monitor at 2.8. Meanwhile, their rating for interaction with the software landscape in the dome versus a conventional monitor are at 2.7.

6.5.2 Cognitive Support

▷ **2.1** Do participants feel that ExplorViz aids in understanding software landscapes?

Participants rate the cognitive support of ExplorViz alone with 3.5. They feel like the software landscape of ExplorViz provides an efficient oversight about the structure and processes of the software. Especially for the initial understanding of a code, the interactive visualization due ExplorViz, it provides a more tangible concept for unfamiliar software.

▷ **2.2** Do participants feel that ARENA2 aids in understanding of concepts?

In summary, participants do not perceive relevant cognitive support from ARENA2 when used on its own. While they recognize the benefits of its expansive projection space, especially for larger landscapes that would not fit in smaller projection areas, they generally find projection spaces to be of lesser importance for cognitive purposes. This is reflected in their rating of 2.2 for the second cognitive statement.

▷ **2.3** How does the combination of ExplorViz and ARENA2 enhance the process of understanding a software landscape, according to the participants?

Due to the nature of research questions 2.2 and 2.3, the responses are largely similar. However, the rating for this topic is slightly higher at 2.6.

6.5.3 Enhance Collaboration

▷ **3.1** Is ExplorViz supporting collaboration?

On average, participants appreciate the organized structure of the software landscape provided by ExplorViz. This clarity helps to focus conversations on specific areas of interest within the software. Comparatively, a unified visual representation proves more efficient than code-only discussions when teams are trying to pinpoint specific parts of the software to discuss. Furthermore, participants find it easier to understand explanations from other software developers, especially with the collaborative features offered by ExplorViz and the facilitated tracing, which means the execution tracking of a process within a system. This qualitative report is also underlined by the rating of the first collaborative statement of 3.7.

▷ **3.2** Is ARENA2 support collaboration?

The expansive projection space enhances collaboration by offering a consistent view, maintaining a unified perspective for all. Furthermore, the lots of personal space allows participants to see the advantages of additional tables, fostering a meeting-room experience where multiple collaborative discussions can occur simultaneously. The nature of co-located collaboration also enables teams to concentrate on their tasks. Consequently, the statement that ARENA2 enhances collaboration in general received an approximate rating of 3.67.

▷ **3.3** Does combine ExplorViz and ARENA2 enhance collaboration compared to using them separately?

Due to the collaborative nature of research questions 3.2 and 3.3, the responses are largely similar. However, the rating for this topic is marginally lower, standing at 3.2. Specifically, the participants mention the advantages of utilizing more expansive software landscapes than those employed in our study.

▷ **3.4** How many software developers can collaborate at the same time while maintaining an equal user experience?

On average, 9 participants assess about 7 software developers collaborating simultaneously while maintaining a consistent user experience, with estimates ranging from 4 to 10. However, one participant believes that ARENA2 can provide enough space for 33 software developers working in parallel.

6.6 Discussion

6.6.1 Result Interpretation

The nature of qualitative data acquisition does not let us have a clear differentiation between objective results and an interpretation. This results in a combination of summarizing the

qualitative results and partially interpreting it in another ways.

User Experience

Participants express a positive opinion towards working within ARENA2 environment. The unique projection system, characterized by its high resolution, allow them to mentally dive into the projections. While participants tend to favor the dome display over conventional monitors, they also like the workspace it provides. The expansive personal space and the dome's planetary visual appeal, let the participants enjoy operating within ARENA2.

Cognitive Role of the Implementation

The use test also leads us to discuss the cognitive role of ARENA2. Participants primarily mention the benefits of large personal and projection spaces for collaborative activities. However, they also highlight the less relevant cognitive support offered by replicating the visuals of ExplorViz, as seen on traditional monitors, within the dome. This points to a potential way for adapting the implementation to the dome's surface. The mention of the City-Metaphor [Wettel and Lanza 2007] [Krause-Glau et al. 2022a], of one participant, suggests an intuitive way to understand software architectures. This provides a foundation for adapting ExplorViz's three-dimensional scenes to ARENA2's hemispherical format. Importantly, our study replicates the cognitive support offered by ExplorViz.

Collaborative Role of the Implementation

The combination of ExplorViz and ARENA2 primarily pinpoints collaborative capabilities. The abstract visualization of software structures functions as a roadmap for team communication. This visualization aids in comprehending unfamiliar software, building discussions around it. The unified display within ARENA2, combined with lots of personal space, facilitates co-located meetings for up to seven software developers. Such team meetings optimize collaboration by allowing teams to analyze software landscapes together, using the benefits of synchronous interactions. One point from our findings is the potential of ExplorViz in interdisciplinary team discussions, which can be attributed to the software's detailed yet abstract landscapes that might help to give a brief overview of the software.

Interaction between Cognition and Collaboration

Considering the cognitive support, it is hard to differentiate between cognition and collaboration, because collaboration is inherently connected with an enhancing learning effect [Krause-Glau et al. 2022a]. While questions aimed to explore the standalone cognitive capabilities of ARENA2 and ExplorViz, the nature of ARENA2 needs a visual element for projection. Therefore, the requirement that make collaboration possible might also be hints for further cognitive aid.

6.6.2 Limitations

While working on the tasks, the process should pause after 50 percent of the tasks were completed. At that point, the pair should then switch to the person in control of the software landscape. However, we initially give participants the freedom to decide who would take control, this led to a situation where one individual often dominated, leaving often only one person to truly experience controlling the software landscape in ARENA2.

An understanding task (see Section 6.3.2) is misunderstood because the software landscape does not provide information to solve this type of task. Therefore, participants are instructed to solve an adapted version of the task that focuses on the payment process for the pet clinic's care, rather than on the payment process for buying a pet. During the evaluation process, the question concerning the enhanced understanding offered by ARENA2 as a standalone environment was duplicated with a similar question about the combination of ExplorViz and ARENA2. This led to duplicated responses from the participants, because ARENA2 itself supports cognition only by visualizing anything, in our situation software landscapes provided by ExplorViz. These issues, coupled with the challenges of interpreting complexly phrased task definitions, underscore the time-sensitive nature of this evaluation. The absence of a pilot study to identify problematic wording or to clarify technical limitations, should be considered when repeating this evaluation.

Technical Limitations

Due to our reliance on the Spectate Feature (see Section 3.2.1), we can only use ExplorViz's features that are considered in the spectating. Thus, we can not display a (a) mouse cursor, (b) the sidebar by right-clicking in the rendered space, (c) the pop-up information windows by hovering over components, and (d) the complete name of a part of the software landscape due to the length of the name.

(a) If we can not see the mouse cursor on the dome's visual, all users need to focus on the control instance, which hinders them from looking into the dome. To be stationary around the area of the Main, they also can not utilize the space ARENA2 is providing to wander around while interactively showing parts of the software landscape they want to talk about (see conceptual visualized in Figure 6.2).

(b) The right-click menu allows users to open the entire software landscape at once. This feature not only saves time but also reduces the potential for frustration. In software landscapes, each component requires an individual click, which some participants found to be tedious. Addressing this issue enables users to focus more on the control instance and less on the dome's visual display, which is a problem considering the user experience.

(c) The Spectate Feature does not support ExplorViz's functionality of displaying a pop-up information window when hovering over a component. Consequently, this window appears only on the Main display, diverting the user's attention away from the dome's visuals. Furthermore, even if this pop-up feature were supported by the Spectate Feature, it would appear five times due to the synchronization. Additionally, the overlapping

Figure 6.1. Shorten name of class and software.

projection from the dome's projector distribution (see Figure 5.11) would render the pop-up only partially visible.

(d) Due to the limitations in shortening the names of parts of the software landscape (see Figure 6.1), it becomes practically impossible to work with the dome's visual display in ARENA2 only. To view the complete name of a software component, one has to hover over that part, which triggers the mentioned pop-up window containing additional information.

In summary, the evaluation tasks are artificially made more challenging due to these technical limitations.

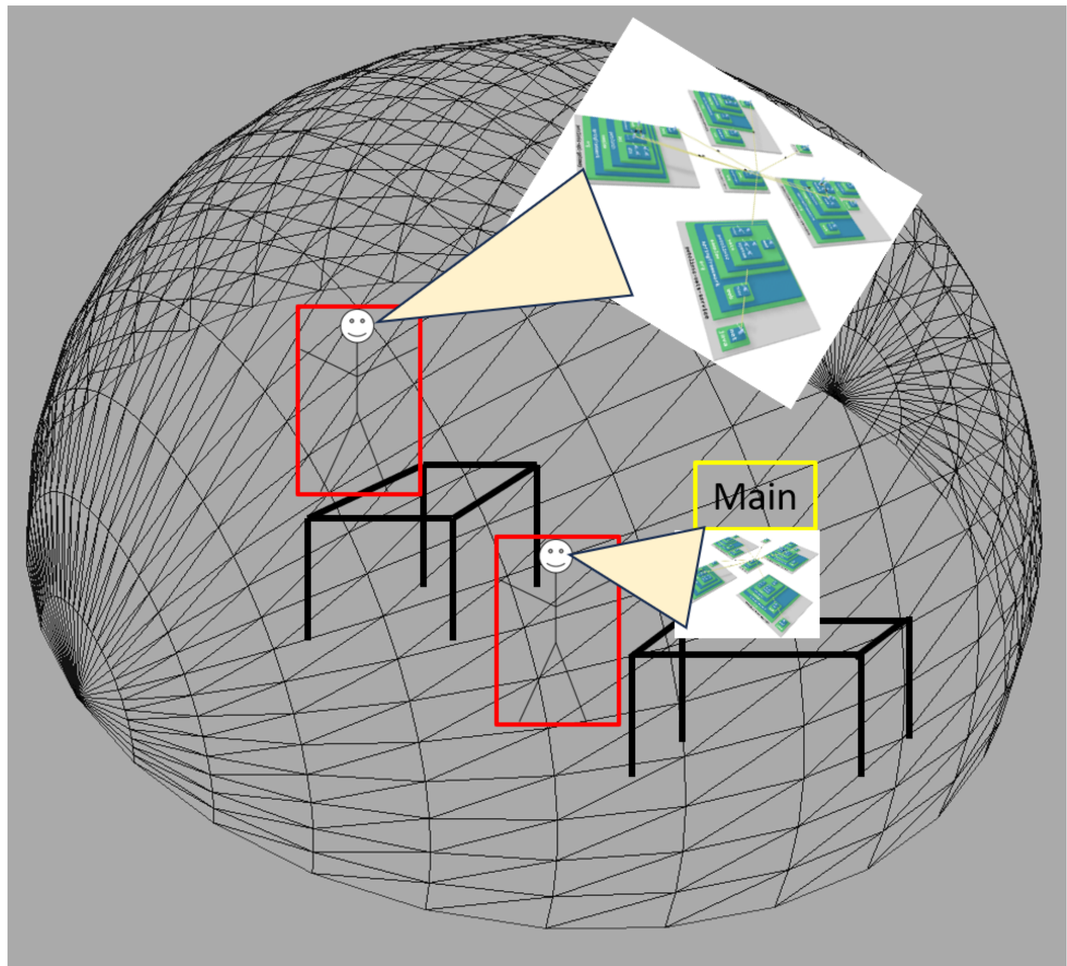

Figure 6.2. Conceptual visualization of the scenario of ARENA2 during the study, in which one of the participant tend to focus the Main instead of collaborating or using the dome's visual.

Chapter 7

Related Work

To begin with the logically related project, we mention the DEV, as referenced in Section 3.2.2. With the DEV, it is possible to import externally processed geographical data and then visualize it in a web-based interactive environment. As mentioned, this visualization tool is used in ARENA2, the co-located meeting room designed to facilitate immersive collaborative experiences. The DEV offers visualization interactions similar to those of ExplorViz, such as moving, rotating, zooming, or changing the image of the structural information.

Connected to this topic, is the recently published paper of ARENA2 itself [Kwasnitschka et al. 2023]. Kwasnitschka et al. 2023 discusses the development and evolution of the dome-based, spatially immersive visualization labs at GEOMAR for the marine geosciences. Starting with the *GEODOME*, it moved to *ARENA* and its more advanced version, ARENA2, which this thesis is partially about. These visual laboratories adapt techniques from planetariums for scientific research. The domes are excellent places for communication, by audience engagement. ARENA2 represents the latest development, which overcomes limitations by using more flexible approaches for software implementation. The aim is to make these domes not only for special visual experiences but also for education, extending the collaborative possibilities to other subjects like computer science, which we are currently doing through this thesis.

To address the challenges of collaborative software visualization within co-located environments, SourceVis [Anslow et al. 2013]. The Main goal is to enhance understanding and communication in software development topics. With its multi-user design, SourceVis utilizes multitouch tables to facilitate collaborative interactions similar to pair programming, where traditionally one user dominates the interaction. SourceVis offers a variety of visualization types, providing diverse ways to display specific information about a software system. There are primarily three categories of visualizations that enable the following:

- (a) Exploration of the software's terminology.
- (b) Analysis of the dependent structure through views like system hotspots and class blueprints.
- (c) Tracking the evolutionary process of the system, which also supports better management of structural changes.

Similar to ExplorViz, users can drag, rotate, or zoom into the externally imported system, making the software visualization more engaging and easier to discuss. SourceVis requires an introduction and some working experience. It takes time to adjust to the new visual and structural information, which needs to be analyzed and taken into account when considering the evolution of the system and understanding its structural snapshot. Additionally, the multitouch table provides a low resolution of 1280x800 pixels. This is a stark contrast to ARENA2, which consists of five projectors with a resolution of 2560x1600 pixels each.

Syde [Hattori and Lanza 2010], addresses the challenge of collaboration in software development within a group of multiple software developers. It is well-suited for project teams that are geographically divided. In this thesis, we explore co-located collaboration using an immersive visualization space to enhance the experience of working together on a software development task. However, permanent co-location is often not possible, necessitating tools that support the process between these immersive collaboration experiences. To increase team awareness, Syde shares changes across developers and various parts of the system as a key feature. This provides real-time updates on what other developers are doing. Synchronous development helps in several ways:

- (a) It aids in resolving development conflicts.
- (b) It models software evolution based on sequences of changes.
- (c) It classifies types of code changes to enhance conflict detection.

This additional information about the software leads to better management for the software developer. While Syde influences the entire development process of multi-developer project teams, it does cause a cognitive load. Overall, Syde is a tool offering detailed, real-time insights into changes being made by the team. These features can enhance communication quality, reduce code conflicts, and improve software evolution throughout the stages of development.

Jeliot 3 [Moreno et al. 2004b] [Moreno et al. 2004a] is a tool for program visualization, which supports novice software developer to understand Java software. It specializes in the visualization of Java code, transforming lines of code into easy to understand graphical representations. The demo version of Jeliot 3 is providing a theater mode, where we can input Java code and compile it, and it visualizes the programs executing according to the order it should be running. It displays all facets of program execution, which shows us the abstraction of the Java Virtual Machine's behavior. For example, it stops and asks the user for input, if the compiling process of the Java code is designed to do so. Beside the user interface, there is also a display of a call tree. This mode provides a tree shaped graphical representation of all the function or method calls that occur during the program's execution. This helps software developers to understand the flow of the program and the relationships between different parts of the code, particularly how methods and functions are interconnected.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

With ExplorViz focusing on collaboration and visualization of information, its implementation in ARENA2 is a next step. ARENA2 enhances the experience of communication among software developers by providing structured visualization of the software's architecture [Hasselbring et al. 2020]. The expansive visualization space of the dome allows for an overview of large software systems, aiding the development process when considering global structural changes to the software [Kwasnitschka et al. 2023]. Simultaneously, the dome allows for focusing on smaller parts of the software by zooming in, presenting these structural components in a precise and spatially expansive manner, while also offering more flexibility due to the interaction in a three-dimensional space.

In the end, however, the software landscape is displayed much like it would be on a monitor. The environment of ARENA2 provides a more interactive, large-scale meeting room that can be compared with traditional planetary visuals, making it attractive but not fully use the capabilities of the environment ARENA2. In terms of scaling, the dome offers a significantly better opportunity for visual engagement among numerous software developers, thanks to the size of its projection space. Compared to a projection from a single projector, the dome's hemispherical layout offers more space, involving every individual of the development team in a more immersive way. It singles out the scaling factor of ARENA2, thereby providing more space and ensuring equal visual involvement for project groups consisting of more than just a few individuals.

As indicated by the limitations of the evaluation, the Synchronization Feature of ExplorViz shows some limitations when implemented in ARENA2 environment. Furthermore, the evaluation involved only a few participants, making it essential to repeat the evaluation once all future work related to the Synchronization Feature has been completed.

8.2 Future Work

The initiation of synchronization in ARENA2 is automatic, but currently relies on a workaround to handle asynchronous calls (see Section 5.1.3). To address this, process timeouts have been implemented to allow services to finish their setup and to allow for the processing of web requests and responses. Shifting away from using timeouts to a

more conceptual approach—one that waits only for the relevant processes to complete rather than employing a general waiting period—would not only reduce the potential for errors but also improve efficiency, as the system would wait only as long as necessary for processes to finish.

Given the limited time and our initial lack of expertise in three-dimensional modeling, we chose not to refactor the graphical manipulation from matrices to the use of a virtual camera (for further details read Section 5.3.1), such as a PerspectiveCamera (see Section 3.3.1). However, making this change would be a logical next step for the Synchronization Feature. Since the entire rendering space is based on Three.js (see Section 3.3.1) and its various camera types, refactoring from a WebGL [*WebGL model view projection - Web APIs | MDN*] matrices approach to a Three.js virtual camera approach would maintain type consistency and be beneficial in the long run.

As mentioned in Chapter 6, synchronizing the right-click menu and pop-up fields is essential for improving the user experience when using ExplorViz within ARENA2. Additionally, we pointed out the issue of truncated names for parts of the software landscape. This limitation forces users to shift their focus from the dome's visual display to the Main monitor in order to access the full names through pop-up fields. Such a workaround seems inefficient, and we therefore encourage further development to address both of these issues.

To reduce redundancy when specifying of query parameters for the Main and projector instances in ARENA2, it is advantageous to save the identifier for the room and the software landscape set by the Main. This reduces the adaptions of scripts to start the synchronization within ARENA2, when changing the software landscape.

As for future work, optional developments are highlighted in Section 2.4. One technical limitation is the inability to use a mouse cursor in ARENA2. To address this, control via an Xbox-controller could be considered.

If the Synchronization Feature is to be implemented in other environments, extending the frontend with input forms for uploading or modifying configurations could be beneficial. This would allow for more efficient adaptation to new environments or surfaces.

Appendix A

Evaluation Tasks

Table A.1. Metric tasks of the Evaluation.

Number	Description
1	How many packages does the component with the most packages contain?
2	How many classes does the software have in total?
3	Name a component that contains exactly as many classes as packages.

Table A.2. Oversee tasks of the Evaluation.

Number	Description
1	"petclinic-visits-service" manages, among other things, visits of animals. Which two other topics regarding animals are offered by the remaining services?
2	The service programs (excluding "discovery-service") have a similar structure. From which components can one still easily recognize that they take on different task areas?

Table A.3. Understanding tasks of the Evaluation.

Number	Description
1	Describe the process of how the sale of an animal takes place. Please name your starting point and all relevant parts of the software.
2	One can no longer register an animal for medical care! What could be the reason? Describe possible causes, please.

Appendix B

Evaluation Statements

Table B.1. Evaluation Likert based statements to General Usability.

Number	Statement
1	The navigation and interaction with the software model are intuitive and unambiguous.
2	The ARENA2 offers additional advantages for the display of software models that cannot be achieved with conventional monitors.
3	Interacting with the software model in ARENA2 feels better than on a conventional monitor.

Table B.2. Evaluation Likert based statements to Collaboration.

Number	Statement
1	ExplorViz is useful for collaboration.
2	ARENA2 is useful for collaboration.
3	It is helpful for collaboration to use ExplorViz in ARENA2.
4	It is beneficial to use ExplorViz in ARENA2 with larger teams to analyze software.

Table B.3. Evaluation Likert based statements to Cognition.

Number	Statement
1	ExplorViz helps in understanding software.
2	ARENA2 helps in understanding software.
3	The representation of software models in ARENA2 helps in understanding the underlying software.

Table B.4. Evaluation Likert based statements to Affection.

Number	Statement
1	It's fun to use ExplorViz in ARENA2.

Appendix C

Evaluation Open-Text Questions

Table C.1. Evaluation questions to Collaboration, which number identifies the corresponding statement.

Number	Question
1	Which features of ExplorViz are helpful for collaboration?
2	Why does ARENA2 support collaboration?
3	How does ARENA2 support collaborative work with ExplorViz?
4	How large should the team be at most?

Table C.2. Evaluation questions to Cognition, which number identifies the corresponding statement.

Number	Question
1	Why does ExplorViz support understanding software?
2	Why does ARENA2 support understanding software?
3	How does the combination of ExplorViz and ARENA2 enhance the understanding of software analysis compared to using ExplorViz alone?

Table C.3. Evaluation questions to Affection, which number identifies the corresponding statement.

Number	Question
1	What do you enjoy when you use ExplorViz in ARENA2?

Bibliography

[Ang and Tourassis 1987] M. H. Ang and V. D. Tourassis. Singularities of euler and roll-pitch-yaw representations. *IEEE Transactions on Aerospace and Electronic Systems* AES-23.3 (1987), pages 317–324. (Cited on page 16)

[Anslow et al. 2013] C. Anslow, S. Marshall, J. Noble, and R. Biddle. Sourcevis: collaborative software visualization for co-located environments. In: *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*. 2013, pages 1–10. (Cited on page 61)

[*ARENA - GEOMAR - Helmholtz Centre for Ocean Research Kiel*]. *Arena - geomar - helmholtz centre for ocean research kiel*. (Last visited on 2023-06-17). URL: https://www.geomar.de/en/arena. (Cited on pages 1, 7)

[Bortz and Döring 2006] J. Bortz and N. Döring. *Forschungsmethoden und evaluation für human- und sozialwissenschaftler*. 4th edition. Springer-Lehrbuch. Springer Berlin, Heidelberg, 2006. (Cited on page 53)

[*Digital Earth Viewer*]. *Digital earth viewer*. (Last visited on 2023-09-28). URL: https://digitalearthviewer-glodap.geomar.de/. (Cited on pages 11, 12)

[*Docker: Accelerated Container Application Development*]. *Docker: accelerated container application development*. (Last visited on 2023-09-21). URL: https://www.docker.com. (Cited on page 50)

[*Ember.js - A framework for ambitious web developers*]. *Ember.js - a framework for ambitious web developers*. (Last visited on 2023-09-02). URL: https://emberjs.com/. (Cited on page 13)

[*GEOMAR Helmholtz Centre for Ocean Research Kiel*]. *Geomar helmholtz centre for ocean research kiel*. (Last visited on 2023-06-17). URL: https://www.geomar.de/en/. (Cited on pages 7, 47)

[*GitHub - FasterXML/jackson: Main Portal page for the Jackson project*]. *Github - fasterxml/jackson: main portal page for the jackson project*. (Last visited on 2023-09-23). URL: https://github.com/FasterXML/jackson. (Cited on page 32)

[Hasselbring et al. 2020] W. Hasselbring, A. Krause, and C. Zirkelbach. Explorviz: research on software visualization, comprehension and collaboration. *Software Impacts* 6 (2020), page 100034. (Cited on pages 1, 2, 7, 63)

[Hattori and Lanza 2010] L. Hattori and M. Lanza. Syde: a tool for collaborative software development. In: *2010 ACM/IEEE 32nd International Conference on Software Engineering*. Volume 2. 2010, pages 235–238. (Cited on page 62)

[Hemingway and O'Reilly 2018] E. Hemingway and O. O'Reilly. Perspectives on euler angle singularities, gimbal lock, and the orthogonality of applied forces and applied moments. *Multibody System Dynamics* 44 (2018), pages 31–56. (Cited on page 18)

[*How To Use The Guides - Getting Started - Ember Guides*]. *How to use the guides - getting started - ember guides*. (Last visited on 2023-09-02). URL: https://guides.emberjs.com/release/getting-started/. (Cited on page 13)

[*JavaScript*]. *Javascript*. (Last visited on 2023-05-29). URL: https://www.javascript.com/. (Cited on pages 12, 13)

[Krause-Glau et al. 2022a] A. Krause-Glau, M. Bader, and W. Hasselbring. Collaborative software visualization for program comprehension. In: *2022 Working Conference on Software Visualization (VISSOFT)*. 2022, pages 75–86. (Cited on pages 2, 8, 53, 57)

[Krause-Glau et al. 2022b] A. Krause-Glau, M. Hansen, and W. Hasselbring. Collaborative program comprehension via software visualization in extended reality. *Information and Software Technology* 151 (2022), page 107007. (Cited on pages 1, 8)

[Kwasnitschka et al. 2023] T. Kwasnitschka, M. Schlüter, J. Klimmeck, A. Bernstetter, F. Gross, and I. Peters. Spatially immersive visualization domes as a marine geoscientific research tool. In: *Workshop on Visualization in Environmental Sciences (EnvirVis)*. The Eurographics Association, 2023, pages 17–24. (Cited on pages 7, 8, 61, 63)

[Liu and Zhao 2013] L. Liu and Z. Zhao. A new approach for measurement of pitch, roll and yaw angles based on a circular feature. *Transactions of the Institute of Measurement and Control* 35.3 (2013), pages 384–397. (Cited on page 17)

[*Mathematics Stack Exchange*]. *Mathematics stack exchange*. (Last visited on 2023-09-29). URL: https://math.stackexchange.com/. (Cited on page 47)

[Moreno et al. 2004a] A. Moreno, N. Myller, and E. Sutinen. Jeco, a collaborative learning tool for programming. In: *2004 IEEE Symposium on Visual Languages - Human Centric Computing*. 2004, pages 261–263. (Cited on page 62)

[Moreno et al. 2004b] A. Moreno, N. Myller, E. Sutinen, and M. Ben-Ari. Visualizing programs with jeliot 3. In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. AVI '04. Association for Computing Machinery, 2004, pages 373–376. (Cited on page 62)

[*The Ember CLI - Introduction - Ember CLI Guides*]. *The ember cli - introduction - ember cli guides*. (Last visited on 2023-09-02). URL: https://cli.emberjs.com/release/. (Cited on page 13)

[*Three.js – JavaScript 3D Library*]. *Three.js – javascript 3d library*. (Last visited on 2023-09-28). URL: https://threejs.org/. (Cited on pages 12, 14)

[*three.js editor*]. *Three.js editor*. (Last visited on 2023-09-28). URL: https://threejs.org/editor/. (Cited on pages 14–16, 18, 19, 26, 31, 40, 49)

[*VESA - Interface Standards for The Display Industry*]. *Vesa - interface standards for the display industry*. (Last visited on 2023-06-21). URL: https://vesa.org/. (Cited on page 14)

[*WebGL model view projection - Web APIs | MDN*]. *Webgl model view projection - web apis | mdn*. (Last visited on 2023-09-27). URL: https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API/WebGL_model_view_projection. (Cited on pages 10, 18, 19, 40, 64)

[*Welcome to Flask — Flask Documentation (2.3.x)*]. *Welcome to flask — flask documentation (2.3.x)*. (Last visited on 2023-08-17). URL: https://flask.palletsprojects.com/en/2.3.x/. (Cited on page 52)

[Wettel and Lanza 2007] R. Wettel and M. Lanza. Visualizing software systems as cities. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. 2007, pages 92–99. (Cited on pages 9, 57)

[Zirkelbach et al. 2018] C. Zirkelbach, A. Krause, and W. Hasselbring. On the modernization of explorviz towards a microservice architecture. In: *Combined Proceedings of the Workshops of the German Software Engineering Conference 2018*. Volume Online Proceedings for Scientific Conferences and Workshops. Online Proceedings for Scientific Conferences and Workshops. CEUR Workshop Proceedings, 2018. (Cited on page 10)