
Masterarbeit Medientechnologie

Immersive Exploration of OSGi Based Software Architectures in Virtual Reality

vorgelegt von

Martin Misiak

Erstgutachter: Prof. Dr. Arnulph Fuhrmann (Technische Hochschule Köln)

Zweitgutachter: Dipl.-Math. Sascha Zur (Deutsches Zentrum für Luft- und Raumfahrt, Köln)

September 2017

Contents

1	Introduction	3
1.1	Goals and Motivation	3
1.2	Thesis Context	4
1.3	Thesis Structure	5
2	Software Architectures	7
2.1	Java	7
2.1.1	Organization	7
2.2	OSGi	8
2.2.1	Bundles	8
2.2.2	Services	9
3	Software Visualization	11
3.1	Motivation	11
3.2	Classification	11
3.2.1	Software aspects	11
3.2.2	Granularity	11
3.2.3	Number of Views	12
3.2.4	Dimensionality	12
3.3	Metaphors	13
3.3.1	Abstract Metaphors	14
3.3.2	Real World Metaphors	15
3.4	Visualizing Hierarchical Data	16
3.4.1	Explicit Hierarchy Visualizations	16
3.4.2	Implicit Hierarchy Visualizations	17
3.5	Graphs	17
3.6	Approaches	18
3.6.1	3D Approaches	18
3.6.2	VR Approaches	19

4	Visualizing Software Architectures as Islands	21
4.1	Island Metaphor	21
4.2	The Virtual Table Metaphor	24
4.3	Sunburst Islands	26
4.4	Cartographic Islands	30
4.4.1	Construction	30
4.5	Package Dependencies	35
4.5.1	Explicit Visualization	36
4.5.2	Implicit Visualization	38
4.6	Services	41
4.7	Interaction	43
4.7.1	Navigation	44
4.7.2	Displaying Textual Information	46
4.7.3	Conclusion	48
4.7.4	Related Work	49
5	Implementation	51
5.1	Visualization Construction	52
5.1.1	SideThreadConstructor	53
5.1.2	MainThreadConstructor	54
5.2	Virtual Environment	55
5.3	Interaction	57
5.4	Hierarchical System	59
5.5	Graphics	60
5.5.1	Materials	60
5.6	Performance Optimizations	61
5.6.1	Inverse Navigation	61
5.6.2	Island Manager	61
5.6.3	Reducing Drawcalls	62
6	Results	65
6.1	Discussion	66
7	Future Work	69
	Bibliography	71

Chapter 1

Introduction

Although often overlooked, the understanding of existing software systems can occupy the majority of a developer's time, especially if he is new to the project. Due to the prevailing paradigm, where programming consists mainly of editing and writing text, the current IDEs expose at each point in time only a very small fraction of the system and in thus provide little help in understanding the underlying software architecture.

For this purpose, visualization techniques have been developed, which use a metaphor to map intangible software aspects unto visually perceivable entities to help provide a higher level overview of the system. Over the years a number of two and three dimensional visualization techniques have been proposed using various metaphors.

However the visualization of software architectures in virtual reality(VR) is a sparsely researched field. It offers a much higher comprehension potential than classical three dimensional visualizations. However it requires also a different approach, as the requirements on a usable VR application are much higher than those of a classical desktop application. This is derived from the higher immersion degree, that these applications provide. In particular, VR has a high potential of eliminating the navigational problem of traditional 3D visualizations, while maintaining the benefit of the additional dimension and even expanding it, by providing stereoscopic cues to the user.

In this master thesis, an approach will be developed to visualize module based OSGi software architectures in virtual reality. The validity of this approach will be demonstrated on a large OSGi based software project.

1.1 Goals and Motivation

For years now, the software visualization field employs mostly small variations of the city metaphor, where software artifacts are mapped to entities encountered in a typical metropolitan city. Although this metaphor is a good fit for many software architectures, there may be many more metaphors, which excel at visualizing specific architecture

types. Also, the city concept is not always intuitive, or even favorable to some users and thus, the software visualization field would greatly benefit from a larger variety of suitable metaphors. Therefore, the exploration of novel metaphors for software visualization is a central aspect of this thesis. This holds especially true in the context of virtual reality, as this medium is fundamentally different from classical "desktop 3D" and brings its own set of requirements for the design of an effective metaphor.

Virtual reality has changed a lot over the last years. Many scientific fields, especially medicine and education, took advantage of improved technology and reflected this in their research. Software visualization however is very sparsely researched in the context of VR. Since the early 2000's, there has been virtually no research in this field¹. Recently, a new generation of VR devices has reached the consumer market, resuming on the mass adaptation promise given by the first consumer generation in the late 90's. Indeed, the technology has matured and is currently noticeably present in academia and industry. Designing a software visualization, that fully leverages the benefits of modern VR technology is the second main theme of this thesis. While VR is already being used productively in various fields, the question remains if software engineering can too profit from this technology. And more specifically, which aspects of software engineering.

As a first step, this thesis focuses on a high level comprehension of software architectures based on OSGi. This can be very interesting for developers, who are familiarizing themselves with a new software project, as a VR visualization can quickly provide a much needed sense of context, something modern IDE's fail to do. A possible application field would also lie in the educational field. Due to the intuitive and natural interaction and navigation mechanism VR provides, the complexity of software projects and software development in general could be better conveyed to the public.

1.2 Thesis Context

The presented master's thesis was developed in cooperation with the German Aerospace Center(DLR), at the *Simulation and Software Technology* facility in Cologne. In it, the *Intelligent and Distributed Systems* department encompasses a group called *Distributed Softwaresystems*. This group develops the OSGi based software *Remote Component Environment*(RCE), which will be the exemplary subject of the presented visualization technique.

This thesis builds on the work of Marquardt [1], who developed an application capable of analyzing OSGi based software systems. This application will be used to

¹A search for "Software Visualization", "Virtual Reality" on Google Scholar[2001-2017] yielded only one relevant results in the first 400 articles.

obtain the input data for the visualization, as the focus of this work does not lie on software analysis.

1.3 Thesis Structure

The following two chapters will provide the basic knowledge in all, for this thesis, relevant fields. Chapter 2 provides a quick introduction to Java as well as the OSGi framework, which is based on it. Continuing with Chapter 3, the software visualization research field is presented to the reader. Classification criteria for existing work in this field are presented, as well as existing approaches. Additionally, visualization metaphors are discussed in detail. This chapter concludes the introductory section of the thesis.

Starting with Chapter 4, the proposed approach of visualizing OSGi based software architectures in VR is presented. This chapter lays out the conceptual, as well as algorithmic ideas, while referring to related work in academia. Chapter 5 focuses on implementation details, with a special focus on performance optimization, as this is a major concern when developing for VR. The structure of the application is presented and used libraries and resources are highlighted. Chapter 6 presents results, obtained at the end and discusses them. The last chapter covers future work.

Chapter 2

Software Architectures

2.1 Java

The Java programming language is a very popular class based, object oriented language, with support for concurrency. It was designed to be simple and robust, as it is aimed at the production environment. It shares similarities with C and C++ but has a different organization, as it is a more high-level language. It features automatic storage management and avoids unsafe constructs, like unchecked array indexing for example. Java is a statically typed language, which is compiled into a bytecode instruction set, defined by the *Java Virtual Machine*(JVM) specifications [2]. This bytecode format is what makes Java so appealing, as it can run on any platform that has a JVM, regardless of the computer architecture. This is a great benefit for developers, as they need to write and compile only one version of code.

2.1.1 Organization

Java programs are composed of multiple *packages*. A *package* can contain class types and additional sub-*packages*. Class type is used as a collective name for the types: *Class*, *Interface*, *Enum* and *Annotation*. They reside in *Compilation Units*, which form the input for the Java compiler. Usually one *.java* source code file corresponds to a Compilation Unit. Most compilers require the source file name to match the contained class type, which implies that only one class type per Compilation Unit is allowed. It is referred to as the *Top Level Type*, as this class type is still allowed to have inner classes of its own [2].

From the perspective of the Java language specification, the package names can be arbitrary. However, it is a universally accepted convention to name packages in a hierarchical fashion, starting with the reversed internet domain of the company creating it. Each additional domain introduced to the package name is separated by a dot from its parent domain. The resulting hierarchical structure simplifies searching tasks and

improves the overall comprehension of the code organization [2].

Packages are used to modularize Java code. Class types can have different access modifiers(*public*, *protected*, *default*, *private*), which control their visibility in other packages. For example a *public* class can be accessed in any other package, while the *default* modifier only allows a class to be accessed from within the package it is contained in.

2.2 OSGi

The Open Services Gateway Initiative, or OSGi, is a component based, service oriented framework specification for Java. It centers on application development using modular units, called *bundles*. The life-cycle of each bundle is controlled by the framework, enabling developers to add, replace or remove existing bundles at run-time. This functionality is especially important for embedded devices and application servers, both domains where OSGi is well represented. Bundles provide well defined services, which can be consumed by other bundles. This complements the modular concept of OSGi, as services can also be dynamically started and stopped [3]. Popular implementations of the OSGi specification are: Apache Felix [4], Equinox [5] and Knopflerfish [6].

2.2.1 Bundles

The modular system employed in Java only applies access modifiers to classes but not to packages. If a class is defined as *public*, it can be accessed from every available package. Providing access to a class only to specific packages would be a much more desirable behaviour. However Java lacks a mechanism for controlling access on package level. To alleviate this issue, OSGi introduced the bundle concept. A bundle is a self-contained unit of classes and packages, which can be selectively made available to other bundles. Every bundle contains a *Manifest* file, which among other information also defines the imported and exported packages. For a class to be accessible from other bundles, its containing package has to be in the export list. The packages in the import list define the dependencies of a bundle. A bundle can also be instructed to import all packages, that are exported from specific bundles. These bundles are listed in the *Require-Bundle* list. The following lines show a Manifest file defined in a bundle from the RCE project:

```
Manifest-Version: 1.0
Bundle-Name: RCE Core Component Scripting
Bundle-SymbolicName: de.rcenvironment.core.component.scripting
Bundle-Vendor: DLR
Bundle-Version: 8.0.0.qualifier
Export-Package:
```

```
de.rcenvironment.core.component.scripting
Bundle-ManifestVersion:2
Import-Package:
  de.rcenvironment.core.communication.common,
  de.rcenvironment.core.component.model.spi,
  de.rcenvironment.core.component.scripting,
  de.rcenvironment.core.notification,
  de.rcenvironment.core.scripting,
  de.rcenvironment.core.scripting.python,
  de.rcenvironment.core.utils.scripting,
  de.rcenvironment.toolkit.modules.concurrency.api,
  org.apache.commons.logging;version="1.1.1"
Service-Component: OSGI-INF/*.xml
Bundle-RequiredExecutionEnvironment: JavaSE-1.7
Require-Bundle: de.rcenvironment.core.component
```

2.2.2 Services

Services are declared through *Declarative Service Components*¹, which are stored in XML files. A Service Component can reference, as well as provide for multiple services. Java interfaces are used to define the service interfaces, which are referenced by the Service Components. For a Service Component to manifest, it must be implemented by a Java class.

The *Service Registry* is a key component of the OSGi runtime, which keeps track of already registered services. It enables bundles to publish and retrieve services, through Service Components. Upon retrieval, any method defined through the service interface can be invoked. The Service Registry of OSGi is characterized by its dynamic nature. The instant a bundle publishes a service implementation that another bundle is looking for, the registry binds the two bundles together [3].

¹The Declarative Service Component will be referenced henceforth simply as Service Component.

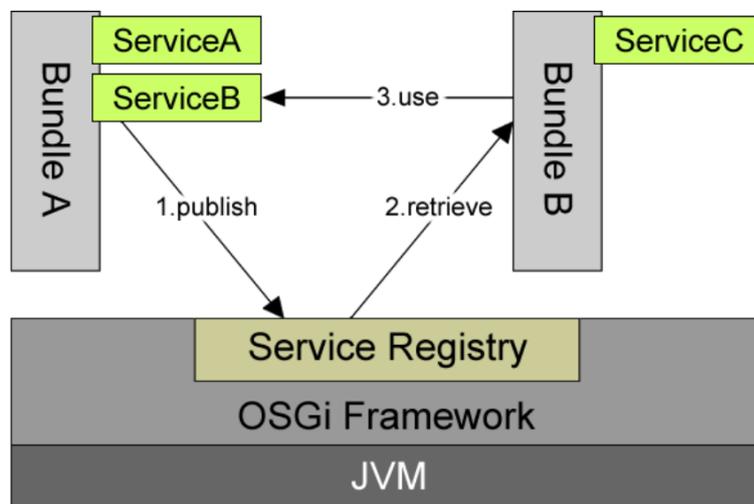


Figure 2.1: The OSGi Service Layer. Image taken from [3].

Chapter 3

Software Visualization

3.1 Motivation

Software is abstract and intangible. With increasing functionality, its complexity grows quickly and hinders its further development. For this purpose, visualization techniques have been developed. They use a metaphor, which maps intangible software aspects unto visually perceivable entities, to help enhance the understandability and reduce the development costs of software systems [7].

3.2 Classification

Software visualization is a very large research field. To ease the classification process when encountering new works, the following categories can be considered.

3.2.1 Software aspects

Due to the high complexity a software system can possess, visualizations tend to target only certain aspects of it. A differentiation between static and dynamic aspects of a software can be made. While dynamic aspects capture information of a particular program run and help in the understanding of execution behaviour, static aspects are derived from pre-execution sources (source code, annotations...) and are therefore valid for all execution paths of a software. Additionally, the static aspects can be extended to capture the entire evolution of a software architecture, by including data from multiple snapshots, acquired usually from a repository.

3.2.2 Granularity

Orthogonally to this, software visualization can be made on roughly three different levels of abstraction [8], where each level is better suited to aid a specific task in

software development. The lowest abstraction level deals with the source code and is paramount to any software project, as the IDE can also be viewed as a low level form of visualization. The middle level provides insight into the workings of an object/class, and is shown to be more effective at it, as opposed to reading the source code only [9]. Visualizations of the highest abstraction level deal with the entirety of the software architecture and belong to the most important in the field of software visualization [10]. They convey the underlying hierarchical component structure, the relationships between these components and the visual representation usually contains some form of code quality metrics.

3.2.3 Number of Views

A software visualization can consist of one or more views. Each view can employ its own visualization approach and can therefore focus on different aspects of the software. Multi-view approaches are able to represent a broad range of information of varying granularity levels, which makes them interesting to multiple stakeholders¹ of a software project, each requiring a different set of information [11]. However they also impose a significant cognitive burden on the user and make a communication on common ground between stakeholders more difficult, as each primarily uses a different view [12]. Single-view approaches on the other hand are easier to navigate and support the collective understanding of a software, as all stakeholders work with the same view, which displays all relevant information. This approach however, is more prone to information overload and can support stakeholder specific tasks only to a certain degree.

3.2.4 Dimensionality

Visualizations can be made in the two dimensional and three dimensional space. Two dimensional visualizations are easier to navigate and interact with, since most users are already familiar with a 2D pc desktop environment. Additionally, elements can be easily laid out to eliminate occlusion and to maximize readability of textual information. In order to avoid a cluttered view for quickly growing data sets, 2D visualizations started relying on multiple views. However these increase the complexity of the visualization, which can result in a cognitive overload for the user.

Approaches using three dimensions can improve the space problem by adding a third dimension. This increases the information density of the visualization, but more importantly, it does not expose the user to any additional cognitive load when processing 3D objects, as this task is completely shifted to the perceptual system [13]. Experimental results confirm the effectiveness of 3D visualizations, in terms of error

¹Different groups involved in a software project. Project managers, architects, developers, maintainers...

rates and execution speed, when it comes to identifying substructures and relationships between objects [14] [15] [16]. Another important aspect of 3D visualizations is their ability to represent real world metaphors more closely than a 2D visualization can. This results in the added benefit that these metaphors provide.

However visualizing in 3D also has its set of problems. The computational complexity is higher, although due to the widespread availability of dedicated graphics processors, this is less of an issue than 15 years ago. 3D visualizations expose more degrees of freedom, which can make navigation and interaction in these environments substantially more complex [17] [18], as most users have only experience in working with 2D desktop environments[19]. Occlusion is another inherent problem when visualizing in 3D, since it can cause objects to appear invisible to the user and therefore distort the view of the underlying data set. Despite these problems, 3D visualizations can provide a benefit over 2D visualizations, as many of their inherent problems can be avoided with a careful design [20] [21].

3.3 Metaphors

As software is abstract and intangible, a metaphor is needed to map its individual aspects unto visually perceivable entities. With their help, information can be conveyed in a representation, which is more familiar to the user, as well as is more easier to understand. A metaphor is a central component of every software visualization and its choice affects not only the appearance, but also the interaction and navigation possibilities, as well as the number of different software aspects that can be effectively displayed.

Metaphors can map software artifacts to abstract geometric shapes, or to real-world entities. Abstract metaphors allow for a greater flexibility in the mapping, as the animator² is less constraint by semantic shape requirements. It also eases the development of configurable visualizations, where users can remap software artifacts to different visual entities more easily, to fit their specific visualization needs. Such approaches however are mostly encountered in visualizations of general data-sets and not software.

Real-world metaphors rely on our natural and intuitive understanding of the physical world. Software artifacts are mapped to known real-world entities, which ideally exhibit similar structural and relational features. This creates a familiar context, where spatial factors in perception and navigation allow a faster recognition and understanding of software systems, while reducing the problematic aspect of disorientation [22]. To fully leverage a real-world metaphor, the same spatial relations and underlying notions between the real world and its virtual counterpart should be preserved. This quality is referred to as consistency [23]. To ensure consistency, constraints have to be placed

²The person responsible for the visualization

on the parameter mappings. For example when using a city metaphor, the parameters mapped to the width and height of a building should be constrained in their values, so that the resulting buildings are taller than they are wider. Otherwise there is a risk, that the visualized object will not be recognized as a building/skyscraper, which would have a negative impact on the plausibility of the chosen metaphor.

Mackinlay [24] proposed two criteria, expressiveness and effectiveness, for evaluating the mapping process of data values unto visual parameters. These have been later adopted by academia to measure the quality of visualization metaphors [23]. Expressiveness measures the capacity of a metaphor, to accommodate all required information into visual parameters [23]. An expressive metaphor needs to provide at least the same number of visual parameters as there are aspects one wishes to visualize. The metaphor can provide some additional unused parameter capacity, however it should not provide less capacity, than the amount of required aspects. This would result in ambiguous mappings, where multiple aspects are mapped unto one visual parameter. When using real-world metaphors, expressiveness also translates to the metaphor providing at least the same number of hierarchical levels as its virtual counterpart requires.

Effectiveness can be generally described as the overall efficacy of a metaphor to represent information. In contrast to expressiveness, it can also depend on the capabilities of the perceiver and is measured in relation to different criteria, like aesthetics, computational performance(time and resource usage) and visual understandability [24] [23].

The following section shows examples of abstract as well as real-world metaphors used in the context of software and data visualization.

3.3.1 Abstract Metaphors

Nested Cubes Metaphor

Proposed in the work of Rekimoto and Green [25], this metaphor focuses on the visualization of hierarchical data. For this the metaphor uses, as the name implies, nested transparent boxes with labels on them. The outermost box corresponds to the top level data and contains boxes which represent the data of the next lower hierarchical level. This nesting continues until the leaves of the hierarchy are reached, which are displayed as labeled tiles. The authors claim the nested cubes metaphor to be very natural and quick to understand, as the concept of a box as a container is very familiar in our daily lives. Also due to the use of transparency, deeper levels of the hierarchy are more opaque when viewed from the outside, which can be thought of as a level of detail filtering mechanism.

Seesoft Metaphor

Introduced by Eick et al. [26], the Seesoft metaphor provides a way of visualizing large amounts of source code in an abstract manner. The main idea is to map the individual lines of code to graphical lines, which are arranged in rows, mimicking the spatial arrangement of the underlying source code they represent. Each line can have a different color to represent a desired metric. The subsequent lines form a column, which represents the source code file, they originate from. The abstraction made by this metaphor is a small one, as a zoomed out view of the source code looks very similar. Here lies the major strength, as well as limitation of this metaphor. While it allows a direct linking to the underlying source code, it does not offer any higher levels of abstraction and its usage of available screen space is sub-optimal, as the occupied space is bound to the shape of the source code text.

Feng et al. [27] extended the two dimensional Seesoft metaphor into the third dimension. In their approach a source code file is represented as a two dimensional array of three dimensional objects, where the object shape, its height and color can be mapped to different metrics. These objects are referred to as "poly cylinders", and each of them represents a line of code. Due to the array/grid like arrangement of poly cylinders and the use of the third dimension, this extension to the Seesoft metaphor achieves a better utilization of available screen space and allows for more metrics to be simultaneously displayed. The trade-offs are, mutual poly cylinder occlusion(although reduced by the use of transparency) and a weaker linking to the underlying source code.

3.3.2 Real World Metaphors

Solar System Metaphor

The solar system metaphor was first introduced into the context of software visualization by Graham et al. [28], where it was used to visualize a Java based project. Each Java package is mapped to a sun, which is being orbited by several planets at different orbits. While the planets represent classes, the orbits represent the inheritance level within a package. The size of each planet is mapped to the number of lines of code in its underlying class and the color is used to differentiate between classes and interfaces. Representing connections and relationships however is difficult, as the metaphor does not provide a 'natural' mapping for it. Additionally, the chosen concentric layout does not make good use of available screen space, which lowers the scalability of the approach to larger software systems.

City Metaphor

One of the most frequently used real world metaphors for software visualization is the city metaphor [21] [29] [30] [31] [32] [11] [33]. There are several reasons for its popularity. The foremost would be the familiarity of the city concept. Most users know that a city can be organized into districts, where each district can contain multiple buildings. These three hierarchical levels are the basis for most implementations of the city metaphor. A city however, analogous to a software system, is a complex entity and can be viewed on different levels of abstraction. This makes the metaphor expressive enough, to provide an intuitive mapping for systems with different hierarchy requirements. For example if the software architecture is in need of four hierarchy levels, the districts could be additionally split into multiple streets, to accommodate for this need.

Due to the good approximation, that a simple geometric box can provide for the visual representation of a building, the computational intensity of a city metaphor implementation scales very well for larger systems. The metaphor however does not make efficient use of three dimensional space, as the individual buildings can only be laid out in 2D [10]. This also makes the displaying of relationships between the individual components problematic. On the other hand, the two dimensional layout restriction results in simplified navigational tasks and is less prone to information overload.

3.4 Visualizing Hierarchical Data

Many existing concepts and entities from the real-world can be organized in a hierarchical structure. The visualization of hierarchical information is a very large research area, which is highly interlinked with software visualization, as software systems are based on hierarchical structures. A distinction can be made between explicit and implicit techniques [34].

3.4.1 Explicit Hierarchy Visualizations

All techniques in this category originate from the early tree drawing algorithms [35] [36]. The basic concept is to represent a hierarchy by connecting node representations with their respective children via a graphical link, hence the name "node-link-diagram", which is frequently encountered in the literature. The main drawback of this visualization form is the ineffective use of display space, as the breadth of a tree grows exponentially with its depth. There is a large quantity of explicit visualization techniques, however they do not assume a major role in this thesis and will therefore not be further elaborated. The interested reader is referred to [37].

3.4.2 Implicit Hierarchy Visualizations

A more space efficient visualization is achieved by encoding the parent-child relations directly into the node positions, for example via containment, node overlap or adjacency [38]. Due to the better utilization of space, differences in node sizes are easier to perceive, making techniques of this category more efficient at performing node size related tasks [39] [34].

Treemap

The first technique on implicitly visualizing hierarchically structured data was brought up by Schneiderman and Johnson in 1991 [40]. It yields a complete use of the display space³, as the full hierarchy is mapped onto a rectangular region. Within, a tiling algorithm recursively slices rectangles into multiple smaller ones for each level of the hierarchy. Each node is mapped to such a rectangle, whose size depends on a user specified weight. A node always exhibits a weight which is equal or larger to the weight of its children combined. As a result, rectangles of children nodes are fully contained in their parents rectangle.

3.5 Graphs

From a mathematical standpoint, a graph is an unordered pair $G = (N, E)$ of a set of nodes $N = \{n_1, n_2, n_3, \dots\}$ and a set of edges $E = \{e_1, e_2, e_3, \dots\}$, where each edge is defined by a pair of nodes $e_a = (n_b, n_c)$. Based on its attributes, a graph can belong to several categories. The following are the most relevant for this thesis. If an edge consists of an unordered pair of nodes $(n_b, n_c) = (n_c, n_b)$, the graph is undirected. When the node pair is ordered, the graph is called a directed graph or digraph. If E is a multiset, where a specific edge can occur more than once, the resulting graph is a multigraph. Otherwise, the graph is called simple.

Graphs are highly utilized datastructures in the computational sciences, as many real world problems can be modeled with the help of a graph. The field of information visualization makes also extensive use of graphs, as not all information exhibit a strict hierarchical structure, which would lend itself to an implicit visualization. Most graphs are displayed in 2D, where the nodes are represented with graphical primitives, while the edges are displayed as straight lines between these nodes. The main concern when visualizing graphs, is the layout of the individual nodes. Large graphs inherently suffer from the problem of visual complexity, as edges become harder to trace due to line crossings, overlapping nodes and an overall cluttered layout.

³Assuming rectangular displays.

3.6 Approaches

3.6.1 3D Approaches

CodeCity

This work by Lanza and Wettel [21] belongs to one of the more known approaches to software visualization, as they also published an implementation⁴ alongside their paper. It had a tool like quality, which made it not only popular with other researchers, but also with actual software developers. Later, the approach was extended to address the visualization of design problems via "disharmony maps" [41], which changed the colors and transparencies of individual objects to reflect design problem data, computed using Marinescu's detection strategies [42]. The CodeCity approach was validated in a user study [43] consisting of 41 participants from academia and industry. The results report a statistically significant increase in correctness(+24%), as well as a decrease in completion time(-12%) for program comprehension tasks.

As the name suggests, the CodeCity approach uses a city metaphor to visualize a software project. While classes are represented as buildings, the packages they reside in form districts. This granularity was explicitly chosen, as both buildings and classes represent key elements in their respective domains. Class internals are not explicitly visualized, however they contribute to the metrics "number of attributes"(NOA) and "number of methods"(NOM) which are mapped to the width and height of the buildings. Color and transparency are used for user selection and also in the follow up work [41] to display design problems. Districts are displayed as platforms upon which the buildings are placed. The platforms can be stacked on top of each other to represent package hierarchies, which causes the buildings to be placed at different altitudes, creating a notion of topology. The layout of the city boils down to a 2D rectangle-packing problem and is constructed using a modified treemap algorithm, as the positions of the buildings do not reflect any relationships between them. Navigation is implemented in two ways. The user can either orbit/move/zoom the camera around the city, or he can navigate "on street level" among the buildings. Both navigation methods have constraints in order to avoid user disorientation. Object can be selected manually via a mouse pointer or by using an implemented query engine, which can also be used for filtering tasks. The approach however, does not visualize any low-level software artifacts such as methods or attributes, nor does it have a metaphor-aware representation of relationships between classes.

⁴<https://wettel.github.io/codecity-download.html>

3.6.2 VR Approaches

Exploring Software Cities in Virtual Reality

This work [44] can be considered as the most recent⁵ approach to visualizing software in virtual reality. The approach focuses on live trace visualizations using a city metaphor.

Analogous to CodeCity, packages are displayed as plates which can be stacked on top of each other. However the approach allows packages to be opened or closed. While open packages follow the layout described above, closed packages hide the content of their children(sub packages, classes and connections) and are displayed as boxes which encompass the volume of their children. In contrast to CodeCity the focus lies on trace visualization, which is reflected by the shape of the buildings. The base of a building has a constant width while its height is mapped to the active instance count of the represented class. Connections play an important role and are visualized as straight lines where the width of a connection represents the call frequency of methods inside a class.

The visualization is presented to the user in a head mounted display(Occulus DK1⁶), where his head rotation directly controls the view of the virtual camera. Since the used hardware does not incorporate any positional tracking, the position of the virtual camera is fixed and only its viewing direction can be changed. To alleviate this problem, the visualized system can be additionally moved, zoomed in and out, as well as rotated using gesture based controls. These rely on the movement and state of the hands, which are being tracked with a depth camera(Microsoft Kinect v2). Two hand states, hand closed or opened, can be recognized by the system, which is also the main selection/interaction mechanism. To determine which object the user wishes to interact with, the approach uses a virtual pointer which is centered in the middle of the users field of view. The authors also conducted a user study for their system, however it was of qualitative nature, where the participants rated their affinity towards the individual hand gestures. The main complaint of the system was the readability of text labels. Especially the labels of individual buildings, as they have been placed on top of them with a constant scaling. In combination with the low resolution of the head mounted display, the users were forced to zoom in by a large amount, which lead to loss of context and disorientation.

⁵At the time of writing this thesis and to the best knowledge of the author.

⁶The first generation HMD from Occulus targeted for the consumer market.

Visualizing Software Architectures as Islands

4.1 Island Metaphor

Requirements

The main emphasis of this work is to visualize OSGi based software architectures. As seen in section 3.3 a metaphor has to be expressive enough to provide mappings for all software artifacts the user is interested in. The required mappings of an OSGi based system encompass those of a classical Java system. However additional mappings are required to capture the added functionality of OSGi. These include the module layer and the concept of services. Overall, the chosen metaphor must be expressive enough to provide mappings for the following aspects:

- Class types (Classes, Interfaces, Enums)
- Packages
- Bundles (Modules)
- Import/Export relations between Bundles
- Service components together with providing and referencing relationships
- Service interfaces

Although representing a software system at finer granularities than at class level is a nice feature, it is not particularly helpful when it comes to the understanding of the underlying architecture. As pointed out in [21], classes are the cornerstones of the object-oriented paradigm and are therefore, in the context of this thesis, the finest granular software artifacts which need to be visualized.

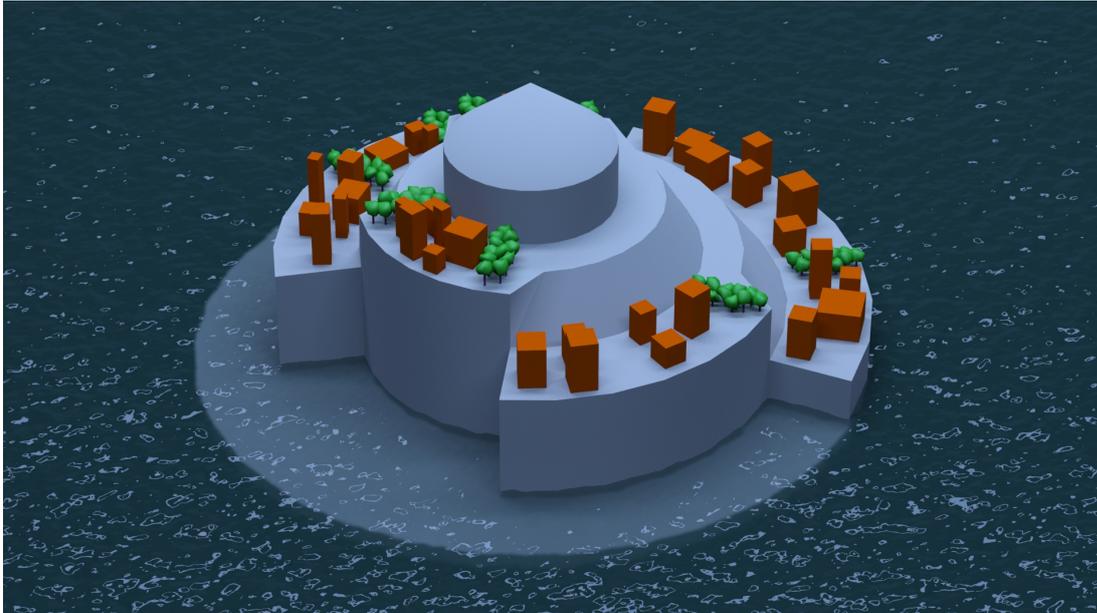


Figure 4.1: A rendering of the first island prototype.

The metaphor should also put more emphasis on the module layer as it forms a central part of OSGi and plays an essential role in the comprehension of software architectures based on it. As the application designed in this thesis targets a single-view visualization, the used metaphor would need to accommodate for this. Additionally, the metaphor should be based on a real-world concept, to take advantage of the familiar context these metaphors provide.

The Islands Metaphor

After careful analysis of the requirements the visualization had on the metaphor, the concept of mapping the software system onto a landscape metaphor was chosen. The landscape metaphor is a category of metaphors, which are, among others, characterized by the specific layout they impose on their elements. Each element is assumed to be a 3D object, which is laid out on a 2D plane. One example falling into this category would be the city metaphor, where each building is a 3D object, whose position can be exactly determined by its location on a 2D plane. While this layout allows for a more comprehensive visualization which is less disorienting for the user, it also has a sub optimal space utilization, as the height dimension remains mostly unused. This empty space however, can be efficiently used to accommodate the service connections specific to OSGi, without sacrificing the comprehensibility of the existing elements. As this thesis targets a single-view visualization, this quality is very important.

An islands metaphor is proposed to visualize OSGi based software systems in this work. As a member of the landscape class of metaphors it provides space for services in the height dimension and is an overall better fit for OSGi based systems than its main

contender, the city/cities metaphor.

The whole software system is represented as an ocean with many islands on it. Each island represents an OSGi bundle and is split into multiple regions. Each region represents a Java package and contains multiple buildings. The buildings are the representatives of the individual class types which reside inside of a package. Each region provides enough space to accommodate all of its buildings without overlapping, and hence the overall size of an island is proportional to the number of class types inside of a bundle. Each island has an import and export dock which handles incoming and outgoing package dependencies between the individual bundles. Service connections are displayed as connected nodes which hover above buildings that describe a service interface or implement a service component. These nodes, together with their connections, are distributed into multiple height layers. This is done to reduce the visual complexity and to enable the users to selectively enable or disable specific layers.

The island metaphor provides a hierarchical structure with three different abstraction levels (island, region and building). The navigation between these layers should be based on our natural understanding of spatial relationships and should therefore be dependent on the relative size of the elements in the users view frustum. Hence, the transition between the levels should happen implicitly, as the user moves closer or further away from an element, or the element itself is scaled. This avoids the introduction of additional complexity into the navigation and ensures the consistency of the landscape metaphor.

The metaphor is flexible enough to be extended if more than three abstraction levels are needed. Individual island groups can form archipelagos, which would provide an additional abstraction level. In the opposite direction, each island region could be interpreted as a country, which would open up even more possible hierarchical subdivisions.

There are several advantages of the islands metaphor as opposed to the cities metaphor. Islands provide a more intuitive representation of modules than cities do, since they express the aspect of decoupled entities, coexisting in the same environment more clearly. Additionally, despite the fact that both islands and cities are considered stationary in the real world, the concept of an island floating/moving along the ocean is more plausible than a city moving along the terrain. This opens up the possibility of dynamically relocating the islands at run-time, while maintaining a certain plausibility. A software evolution visualization could benefit from this property, as the island movements would reflect the dependency changes within the system.

Another important advantage of the islands metaphor is the presence of the ocean as the "base plane", which spans over the landscape elements. Unlike terrain, water possesses interesting optical properties, which can be used for filtering tasks. Due to the absorption and scattering effects which take place in this medium, objects which are deeper under water appear more blurred and their color converges towards the color

of the surrounding ocean. This allows to submerge islands, which are currently of no interest to the user, into the ocean. While submerged, the user can still locate these islands and also estimate their size, however fine details are hidden, which results in reduced visual complexity.

It should be noted, that this thesis presents two distinct island representations(see section 4.3 and 4.4). While one is more realistic and is based on a cartographic approach, the other is more abstract but can represent the package hierarchy.

4.2 The Virtual Table Metaphor

Virtual Reality Metaphor

When designing a software visualization for an immersive virtual environment, additionally to the selected software metaphor, it has to be put into consideration how the individual objects are represented in it. This mapping will be referred to as the virtual reality metaphor. Its choice affects many aspects. It encompasses the way how the visualization is integrated into the virtual environment, the interactions possible with it, as well as the navigational possibilities.

The choice for a suitable virtual reality metaphor is of course influenced by the choice of the software visualization metaphor itself. For example, the visualization could be transferred to the virtual environment in real-world scale. While this would seem natural and seemingly useful for a city metaphor, quite the opposite would be the case for a solar system metaphor. Here our natural navigation techniques would break down and the scale of individual objects, while surely providing a sense of awe to the user, would be far to huge to provide any reasonable overview of the whole system.

Virtual Table Metaphor

For this work, a virtual table metaphor was chosen to integrate the software visualization into the virtual environment. In it, the visualization is presented on top of a virtual table situated in an arbitrary room. The entire content of the visualization is confined to the extents of the table. In contrast to a real-world scale visualization, which is more likely to cause a feeling of presence of being inside the data, the table metaphor allows a more strategic/analytic view of the data. Indeed it was inspired by a strategic planing table that generals would use to plan their approach on the battle field. While they would be restricted to a 2D map which sits on top of the table, we can display arbitrary animated 3D objects on it. Although the table size may vary based on user preference, the metaphor itself imposes a restriction on the size of the visualization space. However this limitation does not have to be seen as a disadvantage, since it enforces the visualization to be shown in a more abstract, space saving, representation. While it can be helpful to

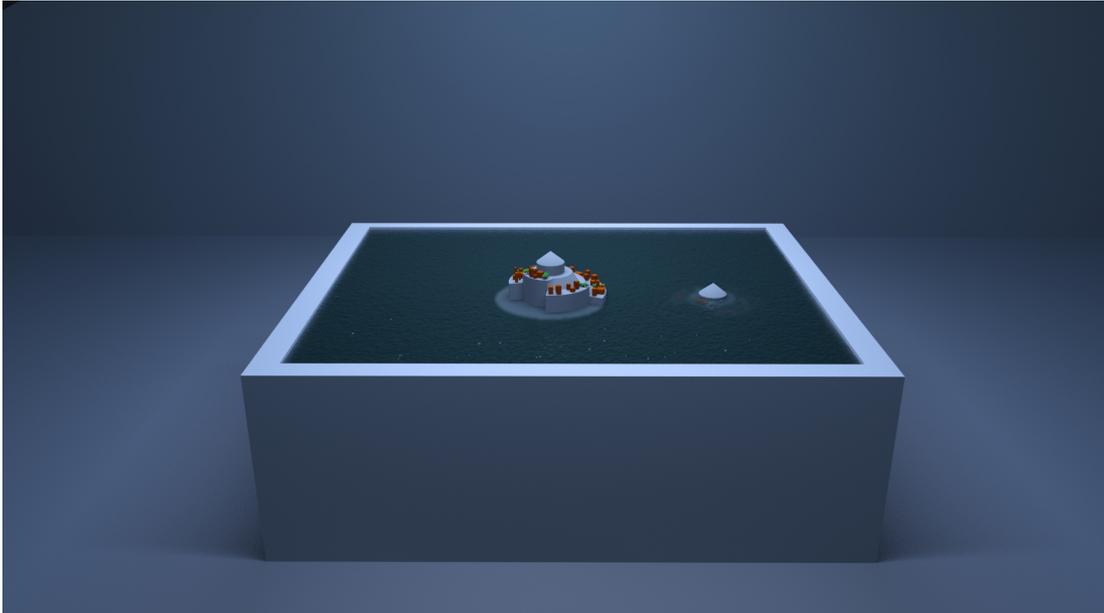


Figure 4.2: A rendering of the virtual table concept combined with the island metaphor.

see the fine grained details of software artifacts, it is the higher abstraction levels which contribute mostly to program and architecture comprehension. With that being said, the user should still be able to navigate freely between the individual abstraction levels as needed.

The virtual table metaphor provides a transparent transition between individual abstraction levels, as the user does not experience any relocation, since only the visualization in the confinements of the table has to be changed without altering the virtual room around it. This reduces user disorientation and motion sickness greatly, as the room always provides a stable frame of reference. This is especially important for the usability of the system in the context of software comprehension, as users can stay longer immersed in the virtual environment without interrupting their train of thought.

As has been noted in section 4.1, the abstraction levels should be directly connected to the relative scale of the elements, which translates to an up or down scaling of the visualization itself. If, due to a high scaling factor, parts of the visualization extend beyond the confined bounds of the table, they would not be displayed. Only content inside of the table bounds is visible. This poses a significant problem for the display of fine granular software artifacts while preserving their surrounding context. However it is the trade-off when using this metaphor.

On the other hand, the limited visualization volume does not force the user to move around excessively in the virtual environment in order to view the desired information. This makes the metaphor also very suitable for a seated or standing VR experience, which can improve user comfort and reduce the dependency on VR hardware capable of precise positional tracking.

From a perception based stand point, a limited visualization volume puts also a constraint on the scene depth, where salient perceptual cues mostly emerge from. This is an interesting property of the metaphor when viewed in the context of the accommodation/convergence conflict, that all modern stereoscopic displays exhibit. The depth constraint on the salient cues leads to a more predictable convergence range for the eyes. This opens up the possibility of adjusting the accommodation distance of the display to match the most probable convergence distance inside of this range. By doing so, the mismatch between accommodation and convergence is reduced, which has a positive impact on visual discomfort and fatigue, symptoms related to motion sickness [45]. When the accommodation distance cannot be adjusted, there is the possibility of going the inverse route and moving the virtual table to match the fixed accommodation distance given by the display. However this will most probably move the table out of the users physical reach, which makes interactions with it more difficult and unnatural. Ideally, the accommodation distance of a display would be fixed to approximately 1 meter, which is a reasonable distance from a table to still enable natural interactions.

As the visualization content is presented on top of the table, the user will most likely view it in a downward angle of up to 50 degree. This should prove advantageous, as these viewing angles are recommendations for working with desktop systems, based on research in the ergonomics field [46].

4.3 Sunburst Islands

The main driving force behind the design of the first island representation was the desire to show the package hierarchy within the island structure, as well as to exploit the full potential of the ocean filtering mechanism. For this, the individual hierarchy levels had to be distributed over the height dimension. The representation of the package hierarchy had to be top-down, where the root node sits at the highest place of the island and the leaves at the bottom. This would allow a submerged island to be continuously raised from under the ocean, while revealing its deeper hierarchy levels as more and more of the island is above sea level. The reverse also works when submerging the island back into the ocean, in order to hide the deeper hierarchy levels.

The chosen island layout is based on a sunburst diagram, as it provides the best fit from existing techniques, targeted at the visualization of hierarchical relationships. The ordering of the sunburst segments reflects the hierarchical package structure, present in the underlying bundle. In order to obtain a three dimensional island, each ring of sunburst segments is extruded in the height dimension. While the outermost ring receives the least extrusion, the rings around the center are extruded the most, resulting in a height profile similar to a pyramid.

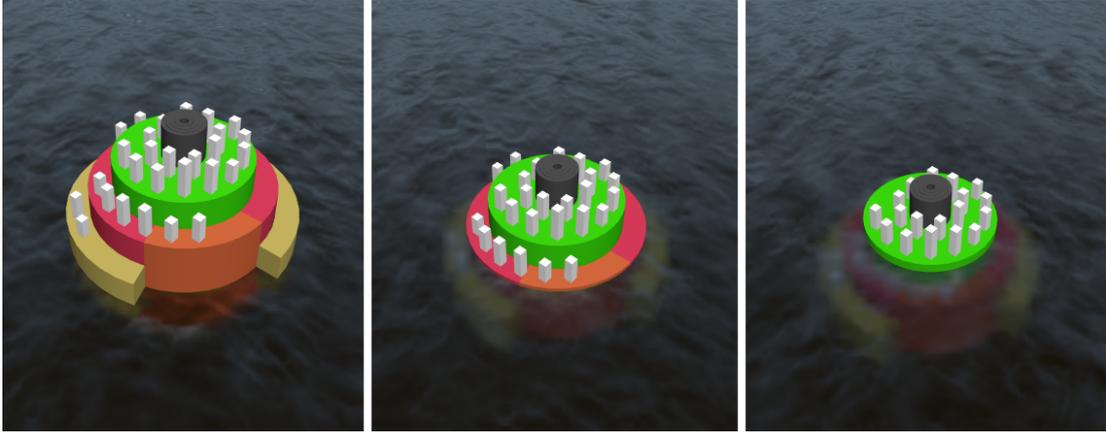


Figure 4.3: Screenshot from the implemented Sunburst Island algorithm. As the island is lowered into the ocean, packages in the deeper hierarchical levels are the first to be filtered out.

Construction

To visually represent the hierarchy of all packages contained in a bundle, the first step is to construct a domain tree. For this, the fully qualified name of each package is taken and broken down into individual domain names, which are stored in the nodes of this tree. To obtain the full package name for a node, the parent nodes have to be recursively traversed up the hierarchy, while accumulating the individual domain names. However not all node traversals yield a valid package name, as developers can introduce an arbitrary number of domains into the package name, which do not contain any classes. Therefore, a node in the domain tree must also store the information if the recursively accumulated package name is of a non empty package.

Once the domain tree is fully populated, the sunburst structure can be constructed. This is done by traversing the domain tree in level order and creating a sunburst segment for each node in the tree. Three parameters need to be determined for the construction of such a segment. The radial extent, width and height. Segments in the original sunburst diagram possess all equal widths, which only allows to compare the relative area of sibling segments contained in the same ring. The segments used for the island however need to offer an absolute area, as they are used to accommodate objects on their surface. Since the area is dependent on both radial extent and width, a heuristic is used to determine the radial extent first. When a new segment is created, its radial extent is always a fraction of its parents extent and is proportional to the area required by this segment and all of its children. If A_c is the area required by the current segment and all of its children, and A_p the area required by all children of its parent segment, then the radial extent of the current segment R_c is given by:

$$R_c = R_p \cdot \frac{A_c}{A_p} \quad (4.1)$$

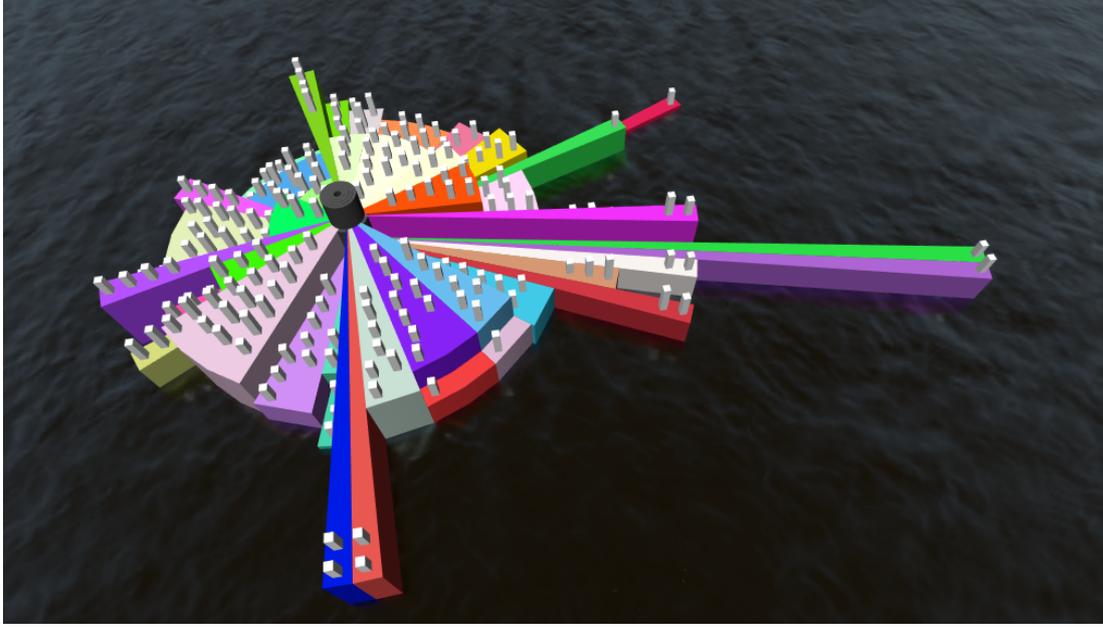


Figure 4.4: Screenshot from the implemented Sunburst Island algorithm showing one of the largest bundles in RCE. When bundles contain a large number of packages, the sub optimal use of space becomes apparent.

where R_p is the radial extent of the parent segment. With R_c calculated, the width of the segment can be easily determined based on the area equation of the segment. Although the width is the only unknown variable, the analytic solution is impractical as it returns a continuous width value, however the buildings that should fit on the surface of this segment are of discrete dimensions. Therefore a solution was used, where the buildings are iteratively laid out on the surface of the segment, extending the width each time the segment did not provide enough space for them. The width needs to be determined only for segments which contain buildings. Segments representing empty packages are assigned a constant width. To obtain a 3D structure, the segments need to be extruded along the height dimension. The extrusion amount is equal for all segments inside of a sunburst ring, which translates to domain tree nodes of the same hierarchy depth. The height H of a ring at depth level i is expressed by:

$$H_i = H_{i+1} + B_{i+1}^{Hmax} + c \quad (4.2)$$

where B_i^{Hmax} is the maximal building height found at depth level i and $c \in R_+$ is a constant user variable. The computed ring height is guaranteed to be higher than all objects contained in the next ring, which is very important for the ocean filtering mechanism to work properly. The last step in the construction is to distribute the buildings on to their respective segments. Luckily this is a trivial task, as the individual placement positions have already been computed during the segment width estimation.

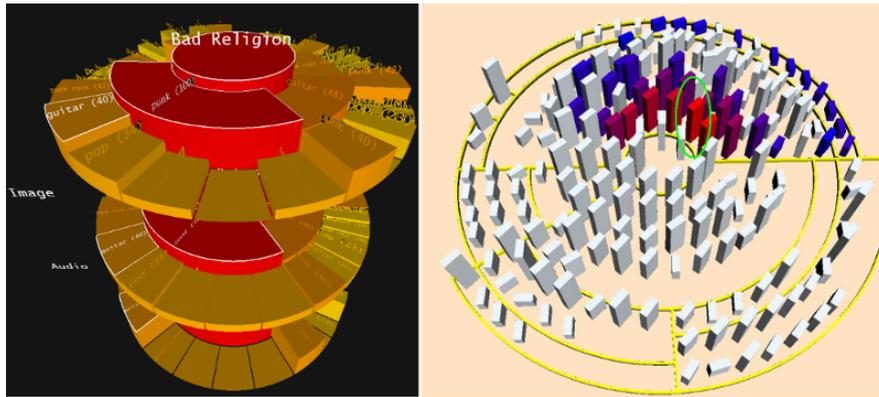


Figure 4.5: **Left:** The 3D sunburst as presented by Schedl et al. **Right:** Sunburst based software visualization by Langelier et al. Images are taken from [47] and [48]

Conclusion

The presented sunburst island representation showed promise in the fields it was designed for. Packages and their hierarchical structure could easily be recognized and the ocean filtering method worked as intended. However various problems lead to a reevaluation of this approach, which ultimately resulted in a completely new island representation (described in section 4.4).

The main problem of the sunburst layout is the sub optimal use of space, due to its radial layout. If a bundle contains many packages, the resulting segments have small radial extents. To compensate, the segments have to be extended in the width dimension until they are able to accommodate their contained buildings. This results in thin, long segments, which are especially problematic in islands whose segments exhibit a high variance in their building count, as this makes the long segments stand out. Although the underlying packages do not differ from packages represented with "normal" segments, the distinct shape can lead users to false conclusions about the package.

Another problem attributable to the radial layout is occlusion. The user cannot see buildings on the farther side of the island, as the extruded segments are occluding the view. This forces the user to navigate around it, which is cumbersome.

Related Work

Schedl et al. [47] extended the sunburst diagram into the third dimension to enable an exploration of a web page collection based on co-occurring terms. The height of each segment was used to represent an additional metric. The resulting 3D sunburst shares a visual similarity with the one presented in this thesis, as the overall height of all segments decreases with increasing distance from the center. However, the height is not normalized within a hierarchy level. Additionally, equal to the original sunburst algorithm, every segment has the same width.

For the purpose of visualizing Java based software architectures, Langelier et al. [48] employed a layout technique based on the sunburst diagram. Similar to this thesis, the individual sectors are mapped to Java packages and their area does not directly reflect any metric, but instead is used to accommodate the classes contained in the package. The classes are represented as simple boxes, where the height, color and twist are mapped to different metrics. In addition to sunburst, a tree map based layout algorithm is also presented. While the construction steps and rules for it are clear, the construction of the sunburst layout is not mentioned in such detail. This makes a direct comparison with this work rather difficult. With that being said, the angular extent of newly created segments is computed identically as to presented in this thesis. This makes the approach theoretically also prone to the "thin segment" problem when confronted with a system containing many packages. The results presented by Langelier et al. however do not exhibit this problem, although they mention the layout technique not having a sub optimal use of space, when confronted with systems containing many packages with very few classes. In contrast to this thesis, the sunburst is used only as a 2D layout and the individual segments are not extruded in the height dimension, resulting in a rather "flat" visualization.

4.4 Cartographic Islands

This approach aims at producing islands, which have a higher resemblance to their real-world counterparts. A cartographic approach is taken, where islands consist of multiple regions, which are subdivided into several cells. Each region represents a package and has an irregular, rugged shape, similar to countries when seen on a map. These regions share borders, and together, they determine the shape of the island. As opposed to the sunburst approach, package hierarchy is not represented, as all packages are laid out as independent regions. Also, all regions reside at the same height, which prohibits a continuous filtering mechanism as described with the sunburst islands. However the created islands look more realistic, which emphasizes the plausibility of the island metaphor. Additionally, each island and region has a very distinct shape, allowing a better memorability and thus a more efficient navigation. Space utilization is also higher than in the sunburst approach as the cells which make up every region are designed to provide enough accommodation area for buildings to be placed on top.

4.4.1 Construction

The island construction is based on claiming cells in a voronoi diagram. Every island is assigned an individual diagram. The first step in the construction is to create a voronoi diagram from a point distribution. It should yield enough cells to contain the whole

island and their size should be large enough to accommodate a building. Also, the most aesthetically pleasing islands were achieved with cells, which did possess only a low variance in their size and shape. To this end, a point distribution based on a regular grid is taken as a starting point. Each point is perturbed in a random direction to break the regular pattern and create more interesting cell shapes. However the introduced noise also results in a higher cell size variance. To reduce the size variance a few iterations of Lloyd's relaxation [49] are performed. One iteration of this algorithm computes the centroid of each voronoi cell and translates the corresponding voronoi cell sites to this position. A new voronoi diagram is computed based on the new site points and can be used again as input for another relaxation iteration. From a frequency based perspective, the cell size variations can be attributed to the low frequency portion of the point set. A Lloyd relaxation generates point sets exhibiting blue noise characteristics, which possess a small amount of low frequency noise, with increasing amounts towards higher frequencies.

In the next step, each package claims multiple cells of the created voronoi diagram, corresponding to the number of contained classes. Cells are claimed one at a time and only cells adjacent to already existing entities can be claimed. Possible candidate cells are stored in a list, which is adjusted each time a new cell is claimed. Claiming a cell removes it from the candidate list, adds its neighbouring unclaimed cells to it (only if they are not already there) and flags the cell with a reference to the package it now belongs to. When the process is started for the first time (first package of each bundle), the candidate cell list is initiated with a cell which sits approximately in the middle of the diagram. In order to create rugged and irregular shapes for the package representations, the next cell from the candidate list is selected randomly. An entirely random selection however works only for a small amount of cells. When dealing with larger packages, this selection mechanism does tend to leave cells in the interior of the shape unclaimed, which results in a non continuous area filled with holes. To reduce this problem, the cells can not be selected with a uniform probability distribution. By introducing an estimating function as described by Yang et al. [50], the holes can be greatly reduced, while preserving the ragged appearance of the regions. This results in more interesting shapes.

Before a new tile is selected from the candidate list, each eligible cell counts its number of neighbours, which have already been claimed. If a cell is surrounded with n claimed cells, the probability of it being a hole grows with n . A score is calculated for each candidate, based on its n :

$$S_n = b^n \tag{4.3}$$

where b is a user definable cohesion factor. Once the scores are known, a new cell can be selected, where the probability of each candidate is directly proportional to its score S_n . If b is set to 2, the chances of selecting a cell which is surrounded with 4 claimed

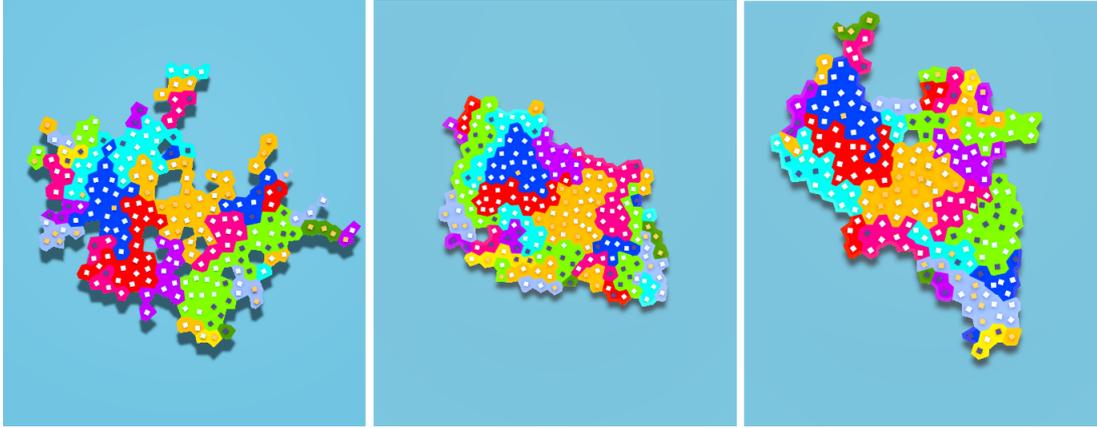


Figure 4.6: **Left:** A minimal cohesion factor leads to very rugged islands with a lot of holes. **Middle:** A very high cohesion factor reduces holes greatly and creates compact islands. **Right:** Dynamic cohesion factor makes larger regions more cohesive than smaller ones. Combined with the claiming of large regions first, the island preserves some of the ruggedness, yet it minimizes holes.

cells is two times higher, than selecting a cell with only 3 claimed neighbours. Due to this, a higher b value results in less holes, but also more regular and compact shapes. A package can run out of space when claiming its territory, as only cells adjacent to the already existing area can be claimed. This can happen when the start cell is situated in a hole or is surrounded by other already existing regions. In such a case, a backtracking [50] is performed, where all currently claimed cells of a package are released and the process restarts at a different starting location. A higher cohesion factor reduces the risk of backtracking as the number of holes is smaller and the individual regions are more convex.

To further reduce the risk of backtracking, while preserving the rugged appearance of an island, the cohesion factor can be varied on a per region basis. This is done by defining b_{min} and b_{max} , which are assigned to the regions based on their size. While the smallest region is assigned b_{min} , the cohesion factor is interpolated towards b_{max} for larger regions. Additionally, the regions are claimed in descending order, starting with the largest package first. This results in islands which contain smaller, irregular regions at their edge, while the larger, more regular regions reside in the interior. From a usability perspective, this layout is more advantageous, as smaller regions are harder to select when surrounded by larger ones.

Once all packages have claimed their cells, the islands need to assume a three dimensional form. This is done during the construction of the coast area. As regions are sequentially claimed, the candidate list accumulates all potentially claimable cells. When the final cell of the last region is claimed, this list consists mostly of cells situated at the boundary of the island. From here, the islands coast is created by iteratively claiming and expanding the boundary cells outwards. This process has the positive



Figure 4.7: A range of different coast shapes, created with specific height profiles.

side effect of filling any existing holes, as they are also part of the final candidate list. Each time the boundary ring is extended, a new height is associated to its cells. A user defined height profile controls this process, where each entry expands the coast by one cell and assigns the stored height value. In the final construction step, a polygonal mesh is generated from all claimed cells in the voronoi diagram using triangulation.

Representing Classes

As previously stated, all classes are represented as buildings. Each region is spread across a number of voronoi cells which is at least equal to the number of classes in the underlying package. The buildings are placed at the center of each cell, as this provides the maximal distance towards neighbouring buildings, in order to avoid the risk of overlapping. Due to the use of Lloyd relaxation, the voronoi site points can be used as a good approximation for the centroid of each cell. The maximal scale of a building is limited to the cell size it resides in. When islands, composed of many cells, are viewed in their entirety, individual buildings become less perceivable. As a result, they are harder to locate and to compare against each other. In order to maximize the perceivability from afar without exceeding the cell boundaries, a multi-storey building representation is chosen, encouraging a metric based expansion in the height dimension.

Conclusion

The cartographic islands solve the main limitation of the sunburst based approach, as they offer a much better space utilization. In addition, the created islands have a more realistic appearance, which improves the overall plausibility of the metaphor. As a result of the probabilistic construction process, each island, each region exhibit a very distinct shape. This offers a very strong navigational cue to the user, as bundles and packages can be identified based on their shape or proximity to other known shapes.

With the help of the height profile a wide range of coast shapes can be constructed.



Figure 4.8: Class types are represented as multi-storey buildings.

This also opens up the possibility of representing a bundle related metric based on the shape and height of a coast region. Although all island regions are located at the same height, the presented software visualization does not depend on this fact. A metric based height extrusion could bring aesthetically more pleasing islands, as well as additional information.

The point set generation used for the construction of a voronoi diagram, as well as the selection of the next claimable cell rely both on random numbers. It should be noted that all random numbers are picked from a sequence, which is generated beforehand with the use of a seed value. This allows to reliably reproduce the shape of all islands for a specific seed. To avoid repeating island shapes for bundles with equal package and class counts(also constructed in the same order), the seed is offset for each island by a value generated from the bundle name.

Related Work

The presented construction algorithm is based on the work of Yang et al. [50]. Their work focuses on the visual representation of general, hierarchically structured data in the form of a geographic map. While the authors employ a hexagonal grid as the underlying tile structure, this thesis uses a voronoi diagram. Although it is computationally more expensive, it allows for a wide variety of interesting cell shapes(hexagonal included). The main reason for this choice is, in comparison to Yang et. al, the small number of claimed tiles. Their method targets data sets with thousands of entries. At this resolution, irregular shapes can be created, despite the uniform shape of a single hexagon(Figure 4.9). The number of classes contained in a single package however is substantially smaller. To achieve interesting shapes, the tiles themselves have to exhibit a certain

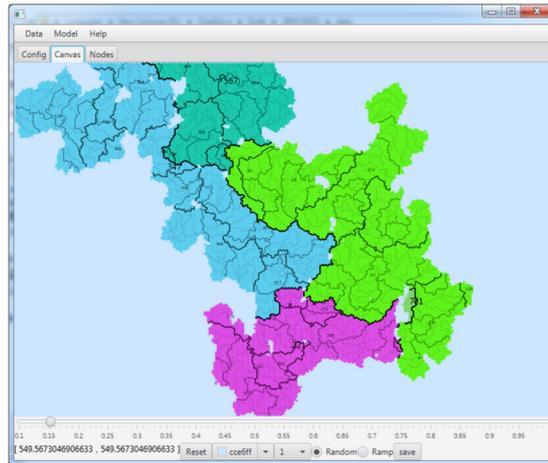


Figure 4.9: Hexagonal tile claiming approach from Yang et al. Image taken from [50].

amount of irregularity, an attribute which voronoi cells provide.

Another important difference results from the restrictions Yang et al. put on the input data. It must be in the structure of a directed tree, where the size of each branch node must be equal to the sizes of its children. A package hierarchy however does not conform to this structure, as each package can contain, in addition to multiple sub-packages, also a number of classes. This complication prohibits the use of the hierarchical tile claiming algorithm as proposed in the authors paper. Instead, this thesis uses a simplified version, where the hierarchy is neglected and all packages are considered equal.

4.5 Package Dependencies

Due to the architecture oriented focus of the presented software visualization, the dependencies between individual modules are of high importance. When dealing with relationships between entities, it is beneficial to interpret the entire system as a graph. In this case, islands represent the nodes, while the package dependencies are the edges among them. The graph has to be a directed one, as each edge must be able to store a two way dependency. Bundle A can import from bundle B, but B can also potentially import from A. The export information is given implicitly by reversing the order of the nodes, which define an import edge. Additionally, each edge should store a weighting factor to represent the strength of a dependency, based on the number of imported packages.

Building on the island metaphor, an import and export port is added to each island. These ports are situated along the coast line and manage the incoming and outgoing dependencies. In order to visualize them, two orthogonal types of approaches can be considered. An explicit and an implicit dependency visualization.

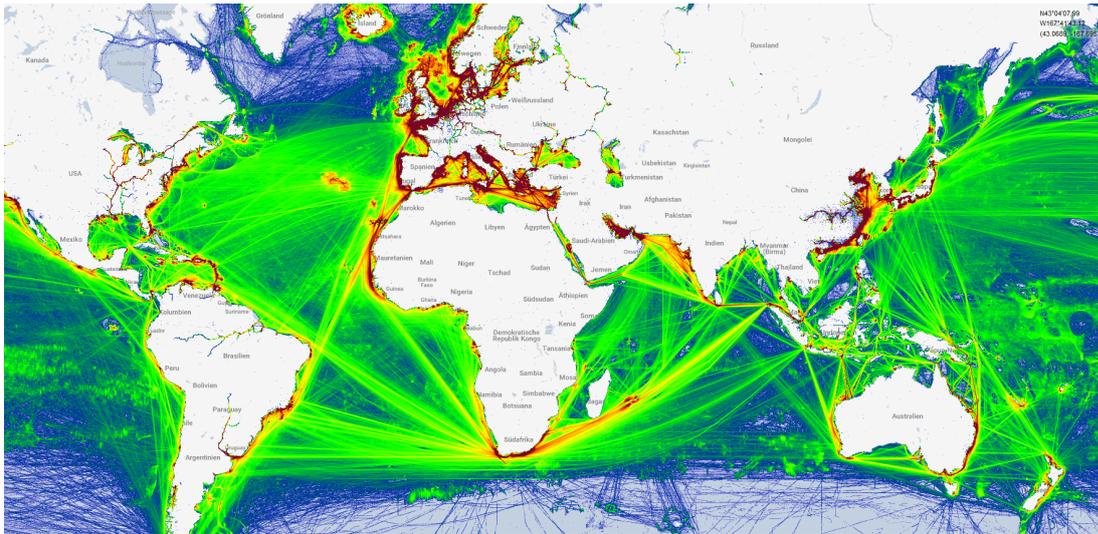


Figure 4.10: A density map of the worldwide ship traffic for the year 2016. Certain sub-routes are often shared between multiple ships, as they offer a more efficient traversal. Image taken from MarineTraffic[51].

4.5.1 Explicit Visualization

The most uncomplicated approach to visualize the existing dependencies, is by drawing straight lines between the entities in question. While these entities are islands, the connections should be drawn between their respective ports, where export ports are connected only to other import ports and vice versa. Straight lines however are very unfavourable, as all ports are laid out on the same height level. This causes lines to intersect with each other, as well as with other islands. As a result, significant visual complexity is introduced into the visualization.

Ship Routes

To further elaborate on the island metaphor, the dependency relationships can be visualized as ship routes between corresponding import and export ports. In contrast to straight lines, ship routes connect two points exclusively via the use of the ocean, hence avoiding intersections with islands. Additionally, they are planned in respect to certain goals, such as: minimizing fuel consumption, maximizing travel speed or maximizing cargo safety. These considerations are derived from spatial environmental factors, which affect every travel route. As a result, ships often share certain advantageous sub-paths. This effectively bundles nearby routes together, greatly reducing visual clutter and revealing high-level edge patterns(Fig. 4.10). Indeed, these are the characteristics of known edge bundling algorithms [52] [53] [54] [55], which are very good candidates for implementing this behaviour.

In addition, animated ship models can travel along these routes to help convey the metaphor to the user. The ships carry containers, which represent the imported

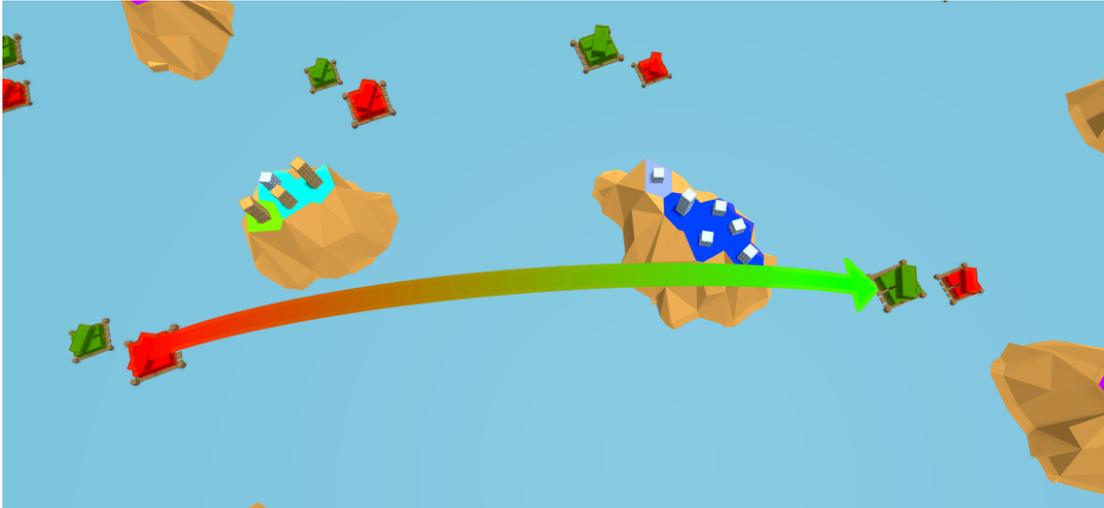


Figure 4.11: The island on the right imports a number of packages from the bundle on the left. This dependency is visualized as an arced arrow.

or exported packages, while their travel direction provides additional cues about the dependency type. This would only be feasible however for the dependency visualization of a few selected islands. When viewing the dependencies of multiple islands, the user would quickly lose track of individual ships and the visualization would become disorienting. Lowering the ship size would help in recovering the orientation, but would defeat the purpose of using ship models in the first place, as the model details would be indiscernible at such small scales.

Arrows

Another solution for the visualization of dependencies, building on the simplicity of straight lines, are import/export arrows. In contrast to lines, they have a finite width extent and an explicit direction. In the geographic context, such arrows are encountered in flow maps [56] and visualize the movement of various resources, entities from one point to another, while the arrow width is proportional to the moved volume. The resulting dependency visualization is similar to a discreet flow map, as implemented by Tobler [57]. In order to reduce the intersection problem of straight lines, the arrows follow a vertical arc. The start and end points are at the height of a port, while towards the middle segment the height increases, reaching its maximum halfway between the anchor points. The arrows maintain throughout a constant curvature. As a result, longer arrows also span a greater height range. A color gradient, together with the arrow head indicate the dependency direction. The width is mapped to the number of packages which are being imported or exported over the given connection.

Conclusion

Overall, the ship route approach is a very metaphor-aware way of representing the package dependencies between individual islands. Due to its closeness to edge bundling, it inherits its strengths as well as its weaknesses. It would however limit the islands to static positions, as moving them would require the bundling solution to be recalculated, which is even with modern implementations still too expensive for real-time purposes [58].

The arrow approach (Figure 4.11) on the other hand is more abstract, but still maintains a certain plausibility, due to its geographic origin. Its simplicity and high performance¹ were key aspects, which ultimately led to its implementation. It also supports the dynamic repositioning of islands at run-time, which opens up additional possibilities for the visualization. The reduction in visual complexity however, is inferior to an edge bundling approach. Still, the addition of arcs integrates well with the use of a tracked stereoscopic display, as depth and parallax cues significantly increase the comprehensibility, as opposed to "flat" lines [16].

4.5.2 Implicit Visualization

Relationships in a graph can be conveyed without explicit connections. Such an approach is very desirable, as it does not additionally introduce any significant visual complexity. The most common way of representing relationships implicitly, is through the node layout. Here, strongly dependent nodes can be grouped together, forming more easily recognizable clusters. Such an approach has the additional advantage of producing layouts which exhibit an increased amount of symmetry and a reduced amount of line crossings. Existing force-directed layout algorithms also strive for these qualities, as their main concern is the creation of aesthetically pleasing layouts. However they also target constant edge lengths and an uniform node distribution, which is prohibitive for node clustering. Nonetheless, with slight modifications, these algorithms present themselves as good candidates for the desired layout.

Dependency-Based Node Layout

The node layout is computed with the help of an iterative, force-directed layout algorithm. In it, nodes are interpreted as particles, which are influenced by attractive and repulsive forces from other particles. These forces are accumulated and applied to each particle at the end of the iteration. Attractive forces are exerted between nodes, which are connected by an edge. The force is dependent on the distance d between the two

¹This approach can be implemented with the use of GPU instancing. See Section 5.6.

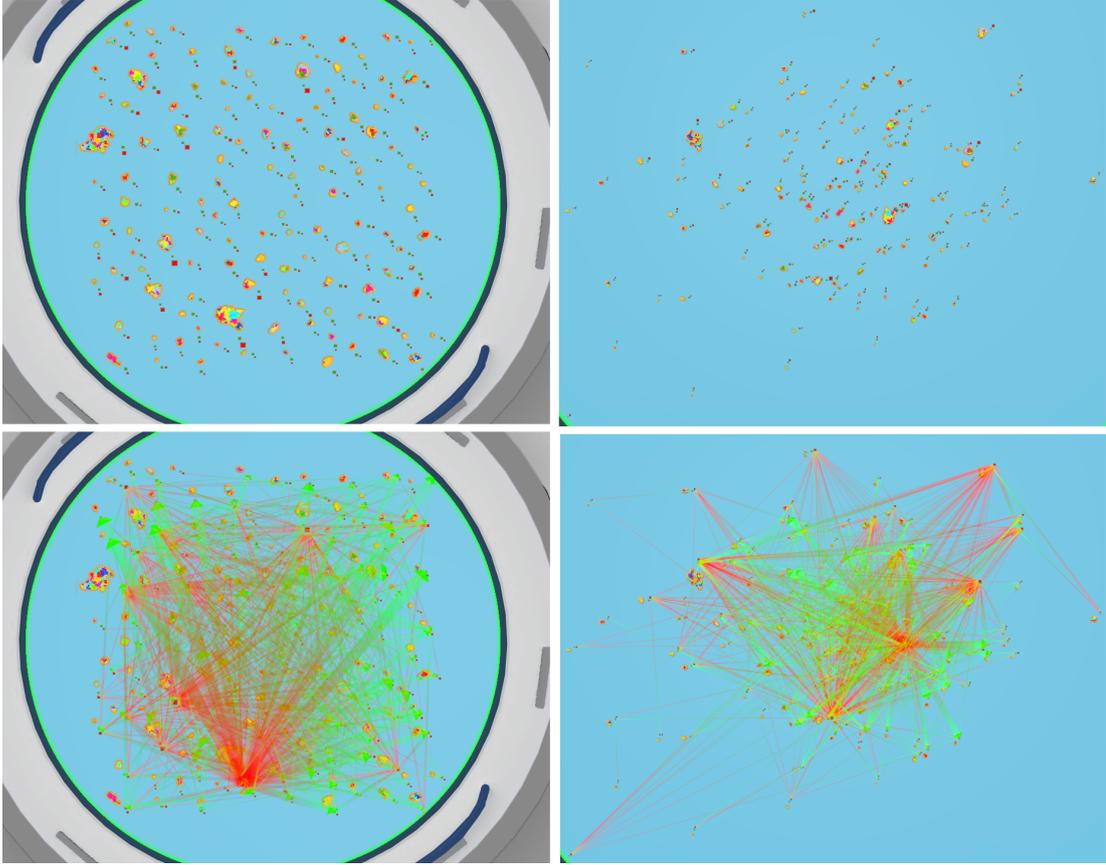


Figure 4.12: **Left:** Random island layout. **Right:** Force directed layout. The top row shows only the islands, while the bottom row also displays dependency arrows between them.

nodes and the variables c_1 and c_2 .

$$F_a = c_1 * \log(d/c_2) \quad (4.4)$$

F_a can be interpreted as a spring like force defined by the the stiffness factor c_1 and the unloaded spring length c_2 . Based on the equation, the spring analogy can be easily seen, as the force is zero, when the distance between the nodes d equals the unloaded spring length c_2 . If the distance is larger, the force is positive, resulting in an attracting behaviour. On the other hand, distances smaller than c_2 result in a repulsive force.

While c_1 is a user defined constant, the unloaded spring length c_2 is computed on a per edge basis and reflects the relative dependency strength between the nodes in question.

$$c_2 = c_3 * \frac{i_{max}}{i_A + i_B} \quad (4.5)$$

Where i_{max} is the project wide largest number of bidirectionally imported packages per edge, i_A is the number of packages bundle A imports from B and i_B the number of packages B imports from A. c_3 is a user defined variable. It can be seen, that c_2 shrinks linearly with the total number of imported packages between A and B, forcing

the nodes to move closer together, in order to achieve a force equilibrium. The lower bound for c_2 is c_3 , as this represents the closest distance two nodes exhibiting a maximal interdependency can assume. It should be noted that F_a is applied to both nodes, only if they are interdependent. If i_A or i_B is zero, the attraction force is applied only to one node.

While the spring force F_a ensures that two connected nodes do not intersect, there is no force governing the relation between unconnected nodes, as these may very well intersect in the layout process. To this end a repulsion force F_r is introduced between nonadjacent nodes.

$$F_r = \frac{c_4}{d^2} \quad (4.6)$$

The repulsion force is described by an inverse-square law, while its relative strength can be controlled with the user defined constant c_4 .

Once all forces for a particle have been accumulated, they are applied in order to determine the next position of the particle. This is done under the assumption of a constant time step Δt and a particle mass of $m = 1$. Each iteration of the algorithm moves the particles, with the intent of minimizing the amount of force they are exposed to.

Conclusion

This section presented a force-directed layout algorithm, which can be applied to a weighted graph. The visualization profits in two ways from it. Based only on the island positions, coarse assumptions about an islands dependencies can be made. Additionally, the produced layout significantly reduces the complexity of existing explicit connections. As can be seen in Figure 4.12, the force directed layout has moved the most independent islands away from the middle. As they are not attracted to other islands, only the repulsion force is responsible for their movement, driving them outwards from the dense island area in the middle. This area contains islands, which are highly dependent of others. The middle of the visualization provides on average the closest distance to all other islands.

Overall the new layout is very beneficial for the visualization, however it takes a long time to converge, as there are no hierarchical acceleration techniques at work. Implementing these would be a much needed improvement. Additionally, the possibility of a layout, driven by lexical similarity between islands could be explored. Developers impose classifications and categories onto the modules through their naming convention. Reflecting them in the relative island positions to each other could prove beneficial.

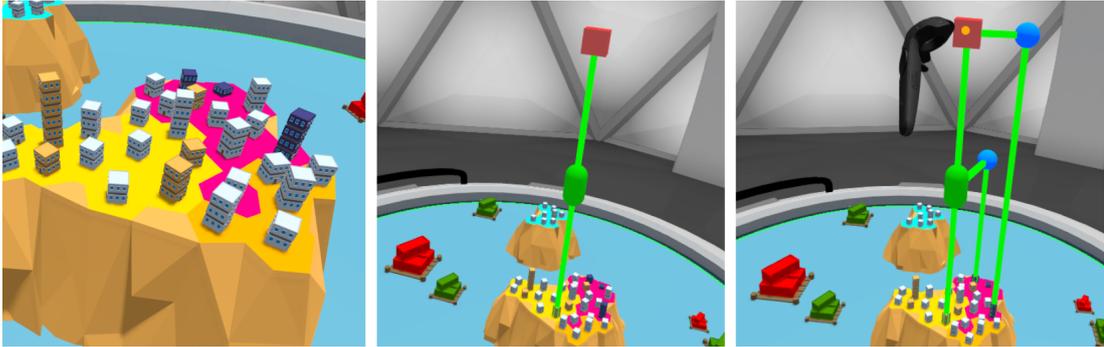


Figure 4.13: **Left:** Service components are represented as orange buildings, while blue buildings are service interfaces. **Middle:** The two service nodes above the service component signalizes that the component provides, as well as references a service interface. **Right:** The connections to the two service interfaces, the service component interacts with, are shown. Both are placed at different heights as the blue service interface nodes are assigned to two distinct service slices.

Related Work

The presented force-directed layout algorithm is based on the work of Eades [59]. However the heuristic described there focuses on undirected graphs, which exhibit a constant edge weight. The main difference to the algorithm presented in this thesis lies in the length of the unloaded spring c_2 . Eades employs a constant length, which leads to uniform edge lengths across the graph, as it is being considered an attribute of aesthetically pleasing graphs [60].

4.6 Services

As reviewed in section 2.2 the main entities of the OSGi service layer are service interfaces and service components. While service interfaces correspond directly to a Java class or interface, service components exist only as an OSGi specific declaration. However, this declaration must contain a reference to a Java class implementing it. The close coupling of service interfaces and components to Java top level types makes them suitable for a visualization in the lowest abstraction level. In the context of the island metaphor this translates to a representation as a building. Additionally to the two service entities, the relationships between them need to be visualized as well. The most simple approach would be to connect the service buildings using straight lines, however this would result in a very overloaded visualization with many crossed lines. With package dependencies also shown, the view would get even more convoluted, as both relation types are visualized at the same height level. Therefore, as mentioned in section 4.1, the service connections are distributed over the vacant height dimension.

This is done by introducing a third entity, the service connection node. These nodes

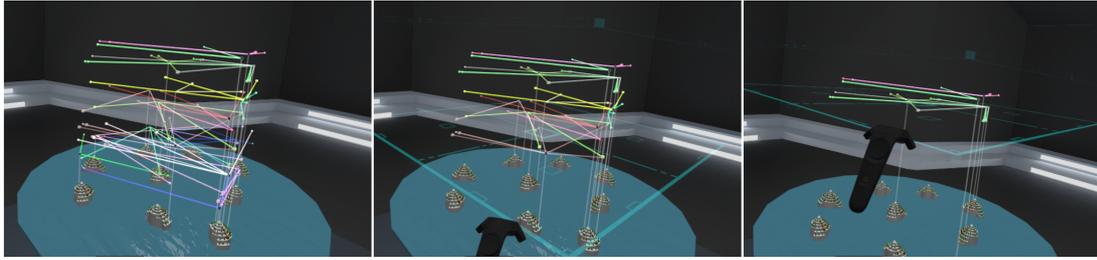


Figure 4.14: Image from an early prototype showing the service slice filtering mechanism. From left to right: The user adjust the height range in which services are displayed.

hover above the service interface and service component buildings at a certain height and act as height offsetted connection points for them. Each node has a visual downward connection to its parent building in order for the user to quickly locate its associated service entity. There are three distinct types of nodes. Service interface(*SIN*), service provide(*SPN*) and service reference nodes(*SRN*). They are assigned to different *service slices*, where each slice resides at a specific height. Service interface nodes assume a central role as they form connections to the other two node types. A connection to these nodes means that the building under the node is either providing or referencing the service in question. While only a single *SIN* can hover above a service interface building, a service component building can have multiple reference or provide nodes hovering above. This reflects the functionality of the OSGi service component, which can reference as well as provide for multiple services. All *SPNs* and *SRNs* connected to a *SIN* form a *service group* and are members of the same *service slice*. Only a few *service groups* are assigned per *service slice*. This reduces the visual complexity, as the nodes and their connections are evenly distributed over the available height dimension. Due to this design, there are no connections going across individual height layers. Connection crossing can only occur between the *service groups* that reside in the same *service slice*. However even this can be reduced as the individual *service groups* are independent and can be assigned to arbitrary slices. This way the total amount of connection crossings can be minimized by assigning only those *service groups* to the same layer, which exhibit a minimal amount of crossing.

To further reduce the visual complexity, a simple filtering mechanism for *service slices* is employed(Figure 4.14). In it the user can specify a start and end height between which all *service slices* will be visible. Slices outside of this range are hidden. The range can be adjusted at run-time, with an immediate impact on the visibility of the affected nodes and their respective connections.

Radio Metaphor

When viewed in the context of the island metaphor, the presented service visualization appears to diverge from this concept, as it assumes a rather abstract form. However with some minor visual adjustments a real-world interpretation is possible. To this end the radio metaphor is introduced.

Each *service slice* can be seen as a radio band, in which multiple radio transmitters(*service interface buildings*) operate. The band in which a transmitter operates is given by the height of its associated *SIN*. Radio receivers(*service component buildings*) must tune in to the same band a transmitter is broadcasting in order to establish a connection. This "tuning in" is represented with a *SPN* or *SRN* at the same height level as the broadcasting *SIN*. While a transmitter broadcasts only in one band, the radio receiver can tune in to multiple bands simultaneously, resulting in multiple nodes hovering above the *service component building*.

4.7 Interaction

To enable the user to fully focus on software comprehension, the cognitive load introduced by navigating and interacting with the virtual environment must be minimal. This requires both activities to be intuitive and natural. The visualization medium also has to be taken into account, as it has a major influence on the interaction design. Luckily, the virtual reality medium is a great fit for visualization tasks, as it provides a natural navigation mechanism and encourages the design of intuitive interaction schemes.

The presented interaction system is designed to be used in equal amounts with the hands, as well as a tracked controller. A functional transfer between the two input modalities is always possible, as buttons can be mapped to hand gestures². However it is not very efficient or usable, since hand gestures in practice do not have a 100% recognition rate [61] and can get quickly tiresome. Therefore it is essential to reduce the reliance on various button presses and hand gestures. To this end, the design focus lies on leveraging the common denominator of both input options. Their position in the virtual environment³.

Building upon this information, all interaction possibilities are integrated into the environment itself. This reduces the requirements to only one button or gesture, as the user only has to align the controller position with the interactable element and confirm his interaction intent via a button press or gesture. Surely it is possible to completely remove the necessity of a button press, as the navigation to the element

²Hand gestures refer to gestures involving mainly finger movement, as gestures involving the entirety of the hand can also be done with a controller.

³Assuming a VR technology capable of positional tracking

could be understood as an interaction intent. However this can often lead to erroneous interaction, especially for users new to the system.

Therefore, a two-phase "navigate then interact" system is employed, as it allows the display of descriptive information regarding the element without altering its state. This is especially important in software visualization, where the user is potentially confronted with thousands of unknown elements.

4.7.1 Navigation

The software visualization is presented in the confines of a virtual table, which is placed inside a room. Due to the use of virtual reality and its inherent navigational advantages, the user can walk around the table and inspect the visualization from different perspectives. However this navigational freedom has its limits when inspecting elements up close, as the human visual system has a limit to the distance it can focus on and fuse a stereoscopic image. Therefore it is crucial to be able to additionally manipulate the visualization itself.

The displayed island system has great resemblance to a cartographic map. Thus the proposed manipulation scheme should be familiar to the user, from the usage of digital maps. Translation, rotation and scaling is introduced to the visualization. The last operation is especially important, as zooming is directly tied to the transition between the individual abstraction layers of the software architecture. This mode of navigation basically follows a WYSIWYG⁴ scheme, where the elements belonging to a specific layer can be interacted with, as soon as they are large enough for the user to see and select.

Translation

The visualization can be translated along the axis defined by the table plane. This usually results in left, right, forward and backward panning, while the translation in the height dimension given by the table normal is prohibited. To apply the translation, the user simply grabs the visualization and drags it in the direction he wishes to translate, releasing it again when finished. Grabbing works by positioning the controller inside of the visualization volume and close to the table surface, while pressing the interaction button on the controller or doing a grab gesture with the hand. Once grabbed, the visualization follows the position of the controller until released (Figure 4.15 Top). It should be noted that the resulting panning direction is opposite to the translated direction. For example, dragging the visualization to the right, exposes new information to the viewport from the left side, having the same effect as panning to the left on a cartographic map.

⁴What You See Is What You Get

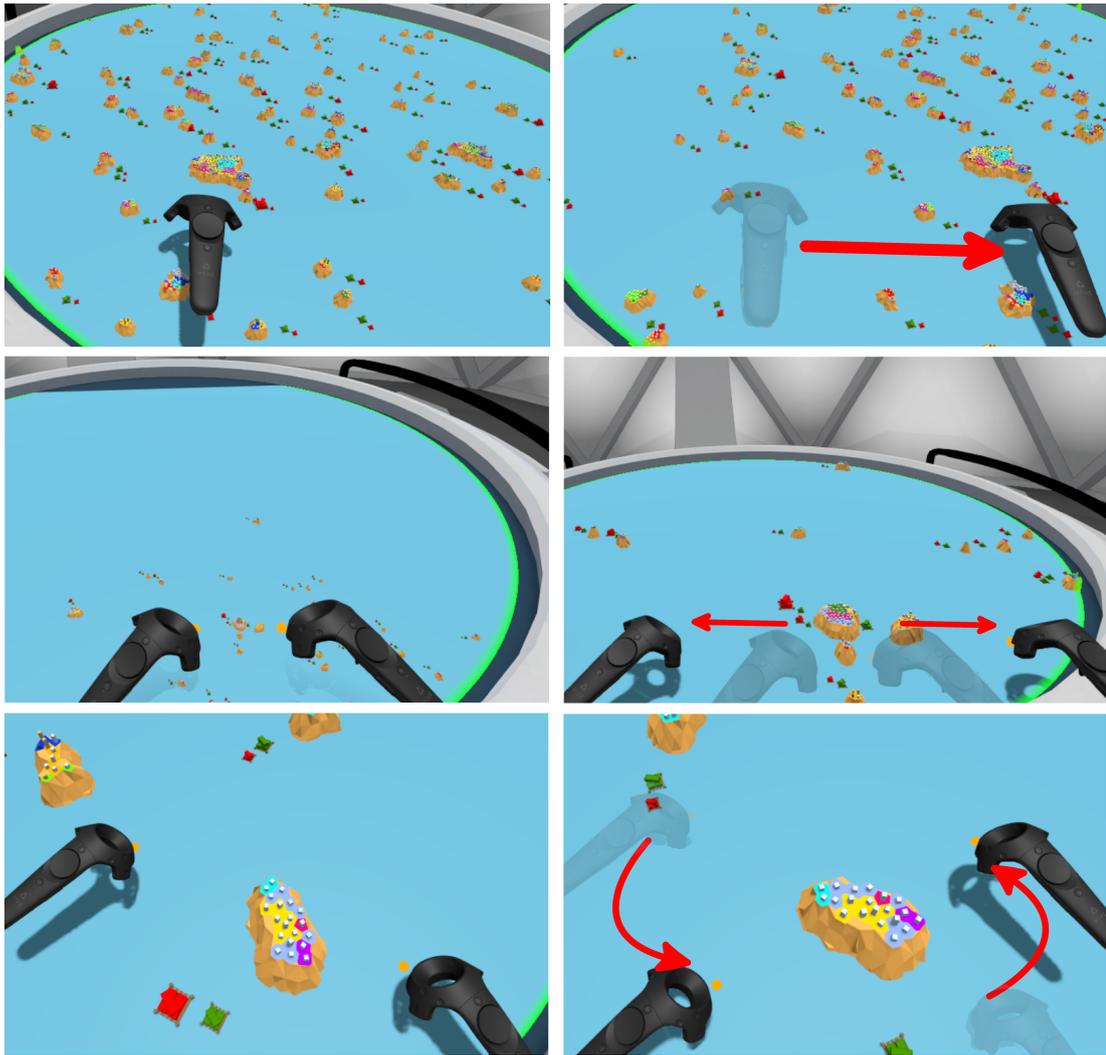


Figure 4.15: **Top:** Translation. **Middle:** Scale. **Bottom:** Rotation.

Rotation and scale

In order to perform these actions, the visualization needs to be grabbed with both controllers. Once grabbed, a virtual pivot point P is established between the controllers. Moving the controllers away from P , along the surface plane of the table, results in a scale increase. Moving them closer towards P decreases the scale. Both actions can be interpreted as a "stretching" or "compressing" of the visualization (Figure 4.15 Middle). In order to rotate the visualization, both controllers are moved in a circular motion around the pivot point (Figure 4.15 Bottom). As with a cartographic map, the rotation is constrained to the axis defined by the normal of the table surface. This control scheme allows both scaling and rotation to be performed simultaneously, while P acts as the transformation origin.

4.7.2 Displaying Textual Information

Displaying the names of individual elements is a crucial aspect of software visualization, as it establishes a connection to the underlying software artifact. Although an impression of the software architecture can still be obtained without them, the subsequent knowledge transfer to the project's source code would be barely possible. However the display of textual information in a virtual reality environment is a challenging task, due to the severe resolution limitations of current HMDs. For text to be clearly readable, it has to occupy a significantly larger angle in the users field of view, as opposed to text displayed on a monitor. This prohibits the display of large quantities of textual information in the virtual world.

Text Labels

Ideally, the user should be able to know which element he is looking at, without introducing any additional effort. Constantly displaying the text labels of every element however, is not a good solution. The required text size would quickly result in cluttered, overlapping labels, which would increase the overall visual complexity by a large amount. Instead, text labels have to be shown selectively, depending on the users interest.

One option would be to show only the labels of elements, which are being looked at, as this would require minimal effort on the user side. For this approach however, an eye tracking solution would be needed. Alternatively, the gaze vector⁵ can be used to approximate the view direction of the user. Although this method is widely used in the industry, it is a very rough approximation, often leading to imprecise selection.

Instead, the choice was made to display only the names of the elements, which are being hovered over by the users controllers. This provides a better control over the display of text labels and frees up the HMD for performing only navigational tasks. In order to display the names of elements further away, a laser pointer functionality is added. It can be accessed by a button press or hand gesture. It would also be possible to activate the laser when the user extends his arms forward, eliminating the need for an additional button. However such a control scheme would require a prior calibration, to determine the users arm length.

Each time a label is displayed, it adjusts its scale to take up a constant amount of display space, irrespective of its actual distance to the user. Thus, ensuring a consistent readability. However once a label is displayed, it will not further change its scale. This allows the labels to be perceived as 3D objects anchored in the virtual environment.

⁵The gaze vector is derived from the direction the users head is pointing. It is completely independent from the actual eye movement.

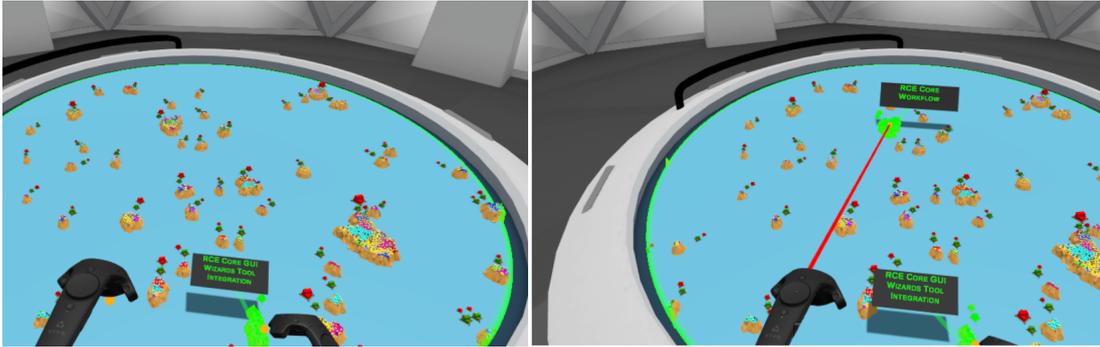


Figure 4.16: Elements highlight upon navigating to them. Interaction can be done through direct controller contact, or via a projected laser. As can be seen in these images, the text labels assume a constant size in the users view, to guarantee a consistent readability.

Virtual PDA

While world space anchored text labels are great for displaying object names, they are not suitable for the display of larger amounts of text. However such a functionality is greatly needed, as not all present data benefits equally from a visualization. There are information, that work best in their textual form.

A virtual monitor or panel can be anchored somewhere in the environment. When the user interacts with diverse elements, additional information is displayed on this panel. To ensure a good readability, the panel has to be very large. Due to its size, it has to be placed in the background to prevent it from occluding the environment. Depending on the environment, a good placement can be difficult, as the environment itself can also occlude the panel. Additionally, it can become quickly cumbersome for the user to frequently alternate his view between the panel and the environment.

Instead of anchoring the panel into the environment, it can be anchored to the virtual body of the user. More specifically, to his hands. This way the panel avoids occlusion problems through the environment, as the user can reposition the panel at any time, without any cognitive effort. The benefit of a large information storage capacity is preserved, as the close proximity of the panel results in large viewing angles.

The panel is attached to the non-dominant hand of the user, so it can be interacted with, by use of the dominant hand. This represents a "double-dexterity" interface, as the interacting hand can be brought to the panel, or the panel to it(or both) [62]. The panel can be thought of as a virtual PDA⁶ or tablet. To avoid unnecessary occlusion and unintentional interactions, the PDA is disabled per default and has to be explicitly activated by the user. This is done by turning the underside of the controller, or the palm of the hand, towards the user. Inside the PDA, a classical tabs and windows system can be employed, to organize information as well as provide additional functionality.

⁶Personal Digital Assistant

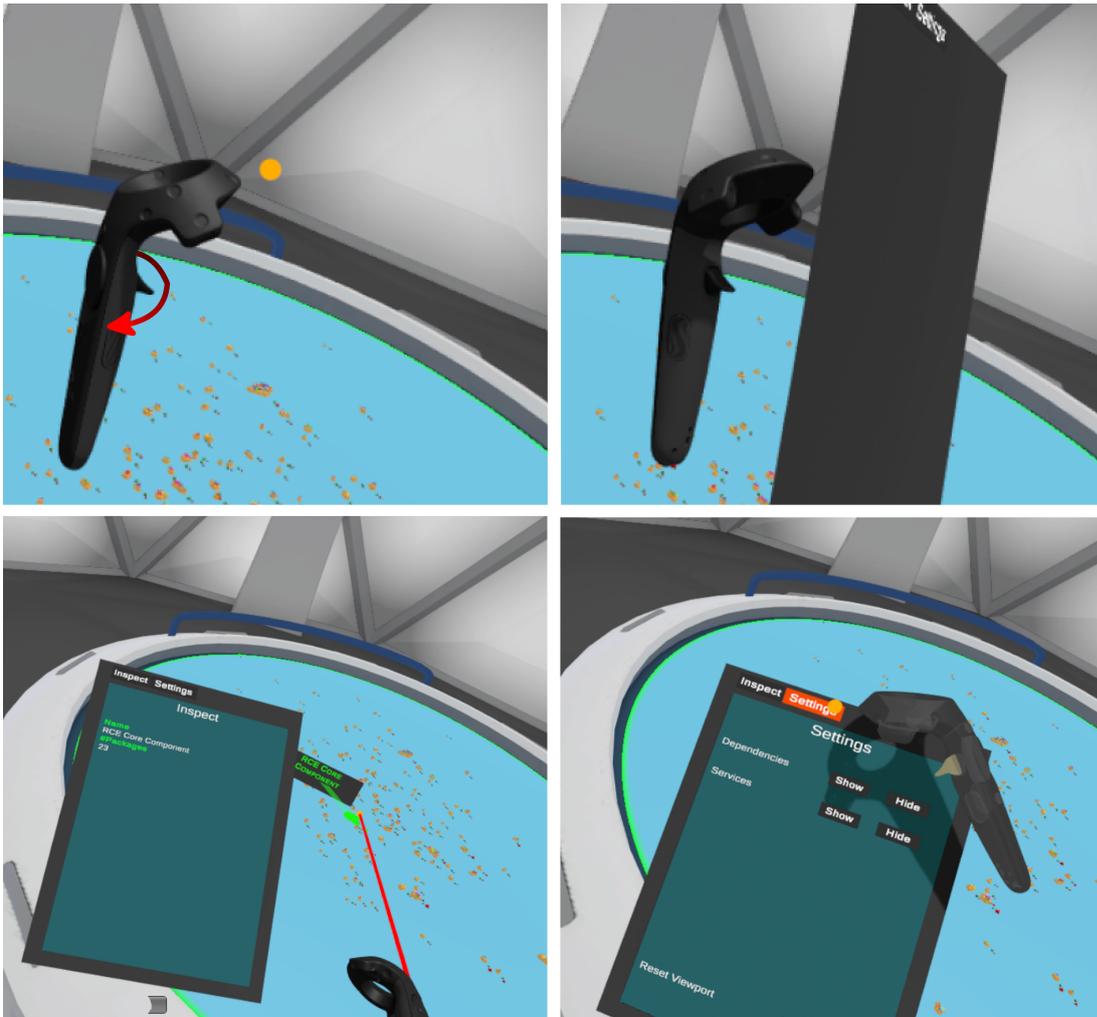


Figure 4.17: **Top:** To activate the PDA, the underside of the controller is rotated into the users field of view. **Bottom:** Extended object information can be displayed on the PDA. It can also provide access to additional functionality.

4.7.3 Conclusion

By building on the strengths of virtual reality, the presented interaction system is kept very simple, while still supporting all required exploratory and comprehension oriented tasks. Although the system reflects current industry standards and best practices on VR interaction [62], it can require an extended learning phase. Especially for users new to virtual reality, as it is a fundamentally different interaction paradigm than the classical WIMP⁷ approach.

The navigational scheme exhibits one problem, which is attributable to the used table metaphor. As the visualization is transformed, the virtual table and its constraints remain unchanged. This means that elements that leave the visualization boundary enforced by the table, are no longer shown, resulting in a loss of context when viewing elements contained in the lower hierarchy levels. A static map of the island world,

⁷Windows, Icons, Menus, Pointer

anchored in the virtual environment, could help regain the lost contextual information.

4.7.4 Related Work

Digital ArtForms developed the "Two-Handed Interface" THI, which is very similar to the presented navigation mechanism. Schultheis et al. [63] compared it against other one-handed VR interfaces for tasks involving object and viewport manipulation. Results have shown, that participants using THI performed the given tasks up to 5 times faster. However, the method requires a longer learning phase.

The problem of losing the overall context when displaying detailed information of a specific element is well known in the field of information visualization. Cockburn et al. [64] presented an extensive review of existing techniques aimed at reducing this problem.

Chapter 5

Implementation

The following chapter presents specific details on the implementation of the OSGi architecture visualization tool **IslandViz**, which was developed in this thesis. While the previous chapter describes the underlying concepts and algorithms, it also discusses multiple approaches for a given aspect. However due to time constraints, not all discussed approaches have also been implemented.

IslandViz was developed in *Unity*, a cross-platform 3D engine with an integrated development environment. The targeted HMD is the *HTC Vive*.

Unity

The Unity game engine is a very good platform for developing VR applications, as it comes with many integrated and easily accessible functions, aimed at the development of 2D and 3D programs. This is especially important, as the developer can focus more on content creation instead of low-level framework building tasks. Unity offers a built-in rendering-, physics-, network- and animation system. The engine is also flexible enough to accommodate for most development scenarios, as it can be extended via plugins, created by other developers. Unity has a very large user base, which translates to an extensive availability of documentation as well as plugins.

The physically based rendering system provides a forward as well as a deferred pipeline. Although it is possible to modify certain aspects of them¹, they are sufficient for most projects. New materials and rendering effects can be created by writing custom shaders. *ShaderLab* is a declarative language, in which all Unity shader files are written. It manages the connection to the Unity Editor as well as a multitude of other high level tasks². The actual shadercode however is written in the *HLSL/Cg* shading language.

¹An extensive modification of the rendering pipeline is cumbersome, as the system is not very transparent

²Blending modes, multiple fallback versions of the shader to support a wide range of devices, etc.

Development in Unity revolves around the central concept of a *GameObject*. It is the base class for every entity in a scene. While it has no explicit functionality by itself, it acts as a container for various components. All components derive from the *Component* base class and represent either built-in Unity functionalities, or custom written behaviour scripts. A combination of different components leads to a *GameObject* with a very specific functionality. For instance, a *MeshFilter* component is responsible for storing the geometrical mesh of an object, while a *MeshRenderer* is needed to make the associated mesh renderable. Additional functionality can be added by the developers with the use of behaviour scripts. A behaviour script is a source code file written in C#, JavaScript or Boo. All behaviours must be derived from the *MonoBehaviour* base class, which itself derives from *Component*.

5.1 Visualization Construction

Constructor Concept

Before the user can see or interact with a representation of the software architecture in virtual reality, a series of construction steps has to be completed. The construction however is very performance intensive and takes an extended period of time. During this time, the user cannot experience any slowdowns or missed frames, as it would lead very quickly to motion sickness. Therefore, the construction has to run on a separate processor thread. Methods defined in the *Unity* API however are not thread safe and can only be called from the main thread. To efficiently make the most use of additional threads, the computations have to be split into two stages. The performance intense calculations are performed on separate threads in the first stage. Once finished, the results are passed to the second stage, running on the main thread, where *Unity* methods can be invoked. With their help, *Unity* native objects are constructed. These stages are implemented as *SideThreadConstructor* and *MainThreadConstructor* singleton objects. Each manages their own subset of constructor objects, which are responsible for the individual processing steps (e.g. parse input data, compute island structure, distribute islands based on layout...). They are called sequentially from the governing *Side-* or *MainThreadConstructor*. To avoid unnecessary pooling on their side, the sub-constructors are given a callback function which is called once they are finished with their task.

5.1.1 SideThreadConstructor

Input Data Loading

As has been mentioned at the beginning of this thesis, an analysis tool, written by Marquardt [1], is used to extract all relevant software information into a JSON file. The first step is then, to read this input file and recreate the JSON data structure inside of the system memory. This is done in the *JsonObjConstructor* and is realized using the *JSONObject* library [65], which is very lightweight, as it contains only one class.

Creation of IslandViz Internal Datastructures

Once a *JSONObject* is created, it can be queried in a hierarchical fashion for information about the analyzed software project. This step is performed by the *OsgiProjectConstructor*. The following objects are created by it:

- **OsgiProject**: Manages all internal datastructures associated with a loaded software project
 - **Bundle**: A list of all available OSGi bundles
 - * **Package**: A list of all packages contained inside of a bundle.
 - **Compilation Unit**: A list of all public class-types contained in a package. A compilation unit corresponds to a single java file from which a .class file is compiled. The class-type can be an interface, enum or class.
 - **Service**: A list of all available OSGi services
 - * **ServiceComponent**: Each service stores a list of referencing and providing ServiceComponents.
 - **BidirectionalGraph**: A directional graph to store the package dependencies between individual bundles. Provided by a Unity compatible port of the *QuickGraph* library[66].

Each object in the hierarchy of the *OsgiProject* is doubly linked with its parent. This allows a more efficient traversal and ease of use for the developer, at the cost of a slightly higher memory footprint.

Calculating the Structure of Islands

The *IslandStructureConstructor* is responsible for the implementation of the cartographic island creation algorithm, outlined in Section 4.4. A central role is assumed by the *TriangleNET* [67] library, which is responsible for the creation and manipulation of the voronoi diagram, as well as the subsequent triangulation of the claimed cells. All computed information is stored in an *Island* object.

Calculating Island Positions

The last processing step on the side threads, is to executed the *GraphLayoutConstructor*. It performs the force directed layouting algorithm described in Section 4.5.2. The algorithm operates on the dependency graph created at the beginning by the *OsgiProjectConstructor*. There are 6 variables that have an influence on the resulting layout.

- **c1**: Strength of the attractive force F_A between connected particles.
- **c3**: The smallest length of an unloaded spring. Associated with the strongest dependency between two nodes.
- **c4**: Strength of the repulsive force F_R between unconnected particles.
- **c5**: A small Attract-to-center force, applied to all particles, to prohibit the particle system from expanding into an arbitrary direction.
- **c6**: Friction strength, applied to each moving particle.
- Δt : The time step taken each iteration.

The attractive as well as the repulsive forces depend on the distance d between two particles. As islands are not infinitely small, d should account for their spatial extent. To this end, the radius of both interacting islands is subtracted from d .

To calculate the new node positions P_{new} from the accumulated force F , Verlet integration [68] is used, as a classical explicit Newton approach was numerically too unstable. This approach is not dependent on velocity and requires only the current and previous positions to be stored. If velocity is needed, for example when calculating friction, it can be derived from the two positions and Δt .

$$P_{new} = 2 * P_{current} - P_{previous} + \Delta t * F \quad (5.1)$$

Once finished, the new node positions are stored back into the graph. Due to the modular implementation, the layout algorithm can be easily exchanged for another one.

5.1.2 MainThreadConstructor

After the last constructor in the *SideThreadConstructor* has finished its task, the *MainThreadConstructor* is started. It contains its own set of constructor objects, which are responsible for the creation of Unity *GameObjects*. Therefore, they must run on the main thread. All entities created by the Unity API are *GameObjects* and will henceforth be referred to simply as objects. Continuing to ensure a smooth visualization during the construction, *Coroutines* are employed. *Coroutines* are functions which can be interrupted at any time and resumed in the following frame. This way, complex calculations

can be split on to multiple frames. The downside is of course, that they take longer to complete.

Constructing Islands

The *IslandGOConstructor* operates on the island structure information computed previously. It creates all individual objects, which are contained inside of an island. The island hierarchy has the following structure:

- Island
 - Coast region
 - Import port
 - Export port
 - Multiple island regions
 - * Region area
 - * Multiple buildings

Before the existing region and coast meshes can be used, they need to be converted from the *TriangleNet* format to Unity's *Mesh* format. The *IslandGO*, *RegionGO* and *BuildingGO* components are assigned to their respective objects. These components associate the objects with the underlying software artifact data structures(e.g. *CompilationUnit*, *Package...*).

Constructing Services

The next construction step creates all service relevant objects, which include service nodes and their respective connections. In a first step, *SINs* are assigned to different height slices. Once finished, their adjacent *SRNs* and *SPNs* are created and connected.

Constructing Ports and Dependencies

In the last construction step, the export and import objects are created, as well as the dependency arrows. The required information is extracted from the underlying graph data structure.

5.2 Virtual Environment

Although the entire software visualization is displayed in the compounds of the table, the enclosing room plays an important role. In order to maintain the plausibility of all "magic" interactions the table is capable of, a futuristic design is chosen, where

the table is augmented with holographic functionality. With the software visualization interpreted as a hologram, the room for plausible interactions is very large.

This interpretation may also help with reducing motion sickness, as the visualization is manipulated. The basis for this assumption would be a differentiation between "tangible" and "intangible" elements in the environment. While the movement of tangible elements may induce the notion of a moving world, which is known for causing discomfort, the movement of intangible elements may have a lesser impact on the current perceived state of the world. However, this is purely speculative as this has not yet been the focus of VR research³. Although evidence show, that the degree of visual realism in virtual content, may be linked to motion sickness [69].

On the other hand, the presence of the virtual room certainly helps in reducing motion sickness [70] [71]. It acts as a stable rest frame for the user. According to the rest frame hypothesis, the human brain maintains a model of moving and stationary objects. The stationary objects form the rest frame upon which all object and self movement is judged. In this model, object movement is unproblematic, as long as it does not endanger the stability of the rest frame. Here, the room and the table form the rest frame. The visualization on the table can be freely transformed, as it is more likely to be viewed as an object and not as a part of the rest frame.

Aside from combating motion sickness, the room design can bring additional beneficial effects. The following list summarizes design guidelines and their benefits:

- Avoid an excessive brightness contrast in the lighting and materials - Helps to minimize the "godray" effect, attributed to Fresnel lenses.
- Environments should not be too dim - The human eye is more sensitive to slight brightness variations in darker regions. This would make the mura corrections pattern, applied to all pixels of the HMD's display, more noticeable.
- Avoid sharp geometric features and contours - Humans prefer round shapes with smooth transitions. Sharp features can convey a sense of threat and result in a negative bias towards the environment. [72].
- Use textured surfaces - Results from the field of cognitive sciences suggest, that patterns are mostly processed in a top-down approach. The processing may be carried out only to a certain depth, as the presence of a global pattern can exclude the further processing of local patterns [73]. In practice, textured surfaces may help reduce the perceivability of the "screen-door" effect, which is more apparent on untextured surfaces, as they do not exhibit any global pattern.

³To the best knowledge of the author

5.3 Interaction

While the interaction concept presented in section 4.7 is aimed at both controller and hand support, due to time limitations only controller support was fully implemented. Interactions in Unity are based on the internal physics engine. For objects to be included in the simulation, *Collider* components need to be assigned. If a physics interaction takes place, for example two colliders intersecting, specific functions in the affected objects are called by the Unity system. These functions can be overwritten by a custom behaviour component to implement the desired actions.

In order to develop for the *HTC Vive*, the *SteamVR* plugin is needed, as it provides access to the *SteamVR* runtime from within Unity. This plugin also contains an interaction system developed by *Valve*. It provides a good and convenient foundation for most interactions implemented in this thesis. The basic building blocks of the *SteamVR* interaction system are the components, *Interactable* and *Hand*. The *Hand* is attached to the controllers and performs a collision detection in regular intervals. Upon collision, specific callback methods are executed on objects containing the *Interactable* component. Instead of the controller, the collisions are checked against a small sphere, attached to the tip of each controller. These are the virtual mouse pointers. A number of reusable components was developed to implement specific functionalities of the interaction system.

- **InteractableViaClickTouch:** This component makes an object sensitive to clicks via the controller. Alternatively a long hover can be used instead of a click (for future use with hands). It intercepts the calls from the *SteamVR* interaction system and checks for button clicks. If a click was detected, a list of methods is executed. Components containing methods that need to be executed when the object is clicked, have to register them into this list.
- **PdaInspectable:** A connection to the users PDA is established through this component. It exposes a method to other components, which accepts a rich text string. Executing this method will display the formatted string on the PDA.
- **TextLabelComponent:** Objects which contain this component, display a name label when being hovered over. In order to avoid occlusion, the labels consider the controllers before being displayed. If the object is being hovered over with the left controller, the label offsets to the right and vice versa. Also, a line connecting the object with the label is displayed.
- **Highlightable:** This component allows objects to be highlighted while a controller is hovered over them. The highlighting works by producing a copy of the object and assigning a wireframe material [74] to it. Each time the original

object is hovered over, the wireframe copy is enabled and rendered alongside the original.

The following elements in the presented visualization can be interacted with:

- **Islands**
- **Island regions**
- **Buildings**
- **Import/Export ports**
- **Service nodes**

Display of Text

Two parameters control the size of the text labels and their containing text. In order to achieve a consistent readability, the labels adjust their scale, prior to display, based on the distance to the observer. Therefore the two parameters are given in angles, as they are distance independent. The user can specify the maximal horizontal angle A_h a label can assume, as well as a minimal vertical angle A_v the label must assume. Text which expands beyond A_h extends the label vertically, until it is able to accommodate for the complete text. A_v is responsible for controlling the font size.

Although the text color can be arbitrarily chosen, a green color was selected. Due to the sub-pixel arrangement on the HTC Vive, this color should provide the best readability. All text is rendered using the *TextMeshPro* plugin, which is based on a signed distance field alpha-test approach [75].

Navigation

Unfortunately, the navigation system could not be implemented with the help of the *SteamVR* interaction system. The reason for this being that an *Interactable* object can, at all times, be only hovered over by one *Hand*. As the navigational mechanism relies on the simultaneous use of both controllers, this presented a challenge. It led to the implementation of a custom navigational system which relied on the native physics engine of Unity.

The navigation sensitive area is enclosed inside of a *Collider*, which is positioned on top of the holographic table. Additional *Colliders* were also added to the virtual mouse pointer, responsible for triggering the table *Collider*. As a result of the implementation, two sets of *Colliders* are present. One responsible for the interaction system and one for the navigation system. Luckily, Unity supports grouping *Colliders* into layers. These layers interact, or do not interact with each other, based on a collision matrix. To

avoid unnecessary physics computations, the two systems are assigned separate, non interacting layers.

To facilitate an easier understanding of the implementation, the entire software visualization is put inside of a container, which also acts as the transform parent of all elements inside. This way, when the visualization is manipulated, only the transformation matrix of the container has to be changed. However objects related to services and package dependencies are put into separate containers. The reasoning behind this is to allow a transformation, that is independent from the visualization container. For example, when scaling the visualization container, the service nodes should scale along, but they should also maintain their absolute height. Otherwise, high scaling values would put the nodes into unreachable heights.

Services and Dependencies

Interaction with service and package dependencies is handled by four components: *ServiceLayer*, *ServiceNode*, *DependencyDock* and *ConnectionPool*. The *ServiceLayer* component is attached to buildings, which represent a service interface or a service component. They differ in color from buildings representing regular classes. Service interface buildings are blue, while service component buildings are orange. Interacting with such buildings expands a number of service node objects, which contain the *ServiceNode* component. Upon activation, they show or hide the connections associated with that specific node. The nodes themselves are represented as simple geometric primitives with distinct colors. Package dependencies are shown by activating the port objects. This functionality is provided by the *DependencyDock* component. Service as well as package connections are managed by the *ConnectionPool* component, which offers *add* and *get* methods that consider connection bidirectionality.

5.4 Hierarchical System

As outlined in the previous chapter, the interaction with the elements of the different software architectural layers(Island, Region, Building) are bound to the scale of the visualization. A completely zoomed out view on the visualization will allow only interaction with the islands, but not with their contained elements. After the view has been zoomed in on an island, the user can interact with regions, but loses the ability to interact with the entirety of the island. As such, only objects presented in the right scale can be interacted with. This hierarchical interaction mechanism also provides an optimization opportunity. It is implemented via the *HierarchicalComponent* component.

Every island, as well as all of its contained objects have a *HierarchicalComponent* attached. Each *HierarchicalComponent* holds a reference to a parent component of the

same type, as well as multiple children components. They are attached to objects which reflect the same hierarchical structure. For example a region's *HierarchicalComponent* holds a reference to the island's component and multiple references to components attached to the contained buildings. The component can perform two basic operations. Split and merge. The split operation disables the object the component is attached to and enables its child objects. Merge disables its child objects and enables the object itself. In practice however, the object is not disabled, as disabling a parent object prohibits also its children from being active. Instead, all of the object's components are disabled. This leaves the object with a minimal performance overhead, which is "as good as disabled". Splitting and merging can be performed based on an arbitrary metric. In this implementation, the distance to the user is responsible for these operations. The distance is measured in the coordinate system of the visualization container, to correctly account for scale changes.

5.5 Graphics

The implementation uses a forward rendering pipeline, in order to benefit from 8x MSAA, as anti-aliasing is very important for VR. To maximize performance, complex graphical effects are avoided and all materials are purely diffuse. A solid performance reserve should guarantee the visualization's scalability to larger projects without problems. Alternatively, a higher rendering resolution can be chosen for smaller projects, as this further improves anti-aliasing.

5.5.1 Materials

All used materials are based on the Unity standard shader. However its pixel shader was enriched with a clipping functionality. It is used to simulate the holographic effect of the visualization disappearing, once it reaches the bounds of the holographic table. All elements of the visualization, except objects belonging to the service and package dependencies, use the same material. This is done to improve performance, as having multiple materials introduces additional state changes on the GPU. The used material has a color palette stored in its diffuse texture channel. The UV coordinates of every object are collapsed to one point, which allows to assign each object a different color from the palette by simply changing its UV coordinate. To provide support for textures, the color palette would need to be extended to a texture atlas, where each object would occupy a specific UV range.

Unfortunately, no sufficiently working shader could be found to simulate a water material with refraction and absorption. Due to time constraints, implementing a custom

shader was not feasible. Therefore, the water filtering mechanism described in section 4.1 could not be implemented.

5.6 Performance Optimizations

Optimizing the performance played a major role in this implementation. Unlike classical 3d applications observed on a monitor, where the occasional stutter is no big problem, a bad performance in a VR application will make it most likely unusable. The main performance problems during the implementation were located on the CPU side. They were the result of the huge number of *GameObjects* that had to be created, in order to represent all software artifacts. Many optimizations were needed to achieve an acceptable framerate.

5.6.1 Inverse Navigation

One problematic area was the manipulation of the visualization. Although "only" the transformation matrix of the visualization container is changed upon manipulation, the transformation matrices of all contained children need to be recomputed before they are sent to the GPU. A related problem stems from the physics system. Internally, it uses a spatial subdivision data structure to hold all colliders present in the scene. This data structure accelerates the collision calculations. When objects change their position or scale, the structure has to be partially recomputed. As the software visualization holds nearly all participating colliders, the complete spatial acceleration structure has to be recomputed upon manipulation. This introduces additional computational time.

To solve this problem, the visualization container and all of its content are not manipulated at all. Instead, the user and the virtual environment are. From the perspective of an observer, moving an object to the left is equivalent to moving the observer to the right. This correlation holds true for rotation as well as scaling. So in order to achieve a specific transformation of the visualization container, the inverse transformation is applied to the observer and the entire virtual room. Although this approach introduces some conceptual and implementational complexity, it yields crucial performance improvements. The performance cost of transforming the observer and the environment is negligible in contrast to the transformation of the entire visualization. With the help of this method, the CPU computation time for the RCE data set is reduced up to 4ms.

5.6.2 Island Manager

The bounds for the software visualization are defined by the virtual table. Islands which leave these bounds, due to viewport manipulation, cannot be seen by the user anymore and are therefore redundant. The *Island Manager* object keeps track of all islands and

activates or deactivates the ones which leave or enter the visualization space. This functionality is implemented by iterating over N islands each frame and checking if they are inside or outside of the visualization space. With the RCE data set it proved sufficient to process only a single island per frame, which results in a virtually non-existent performance impact.

5.6.3 Reducing Drawcalls

Another problem originating from the number of scene objects, are draw calls. Although desktop processor can handle thousands of them, a complex software project can easily out scale this number.

Level of Detail

A well known approach for improving the rendering performance are Level of Detail (LOD) approaches. They replace the mesh of an object based on its distance to the observer. With increasing distance, a lesser detailed mesh is rendered, as the ability to discern fine detail also decreases. However reducing the geometric complexity of an object improves mostly only the GPU performance. An LOD approach can also be used for improving the CPU performance. The *HierarchicalComponent* described in section 5.4 provides the necessary foundation.

Draw calls can be reduced by combining the meshes of multiple objects into one large mesh. The downside is of course, that the individual objects cannot be interacted with, as they are replaced with a single object containing all of their meshes. This fits perfectly into the design of the *HierarchicalComponent*. Each object with this component is assigned a combined mesh of all of its children. In practice, this results only in a single draw call for a complete island, when viewed from further away. As the island gets closer, the *HierarchicalComponent* disables the merged island object and enables the region objects. These hold the combined meshes of their respective buildings. To also improve the GPU performance, a combined mesh should consist of the lower LOD versions of the objects it combines. So while each individual building, when inspected up close, provides a high degree of detail, the combined island mesh can consist of less complex versions of these buildings, as they are viewed from further away.

GPU Instancing

When rendering multiple objects described by the same geometry, redundant information is sent to the GPU with each draw call, as the underlying geometry does not change. First introduced in DirectX 9 and OpenGL 3.1, the instancing functionality eliminates this redundancy by duplicating the geometry on the GPU. This enables the rendering of

multiple objects, based on the same geometry, in a single draw call. The objects may vary in appearance, as each instance can query an attribute buffer stored on the GPU. In practice, this buffer often stores transformation matrices, allowing per instance change in position, rotation and scale. The technique is especially beneficial when rendering a large amount of simple objects, as the rendering on the GPU takes only a fraction of the time needed to issue the draw call itself. This would otherwise lead to a heavy under utilization of the GPU.

Unity introduced instanced rendering in version 5.4. With version 5.6.1 the use of this function became much more convenient, as the "standard" Unity material started to support this functionality. Prior to this, custom shaders had to be written. IslandViz uses instancing to render the package dependency connections, service connections and the service nodes. The instances of all three object types vary only in their transformation matrices, as this reflects the current limitation when using instancing in conjunction with the standard material. When visualizing the RCE data set with all package dependencies and services shown, instancing saves approximately 4300 object draw calls. This reduces overall CPU time by about 40%, yielding a very significant reduction. However even greater reductions can be achieved for larger software projects, as the number of relationships grows exponentially with the number of entities.

Single-Pass Stereo

In terms of rendering cost, stereoscopic displays require twice the computational cost as regular displays. To achieve binocular disparity, a slightly shifted image of the scene is rendered for each eye. The increase in computational cost affects both CPU and GPU. While the GPU has to process twice as many vertices and shade double the amount of fragments, the CPU has to issue each draw call twice. To improve the CPU performance, Unity introduced the "Single-Pass Stereo" rendering option in version 5.4. The default "Multi-Pass Stereo" rendering mode traverses the entire scene hierarchy twice. Each traversal submits draw calls to one of the eye render textures. In contrast, the "Single-Pass Stereo" mode traverses the scene only once, but issues for each object two draw calls. Although technically both modes result in the same number of draw calls, it is the cost of each draw call that matters. Since the single pass mode issues the two draw calls of every object in succession, the second call is much cheaper than the first. This is due to the inexpensive state change the second draw call requires, as only the transformation matrix needs to be adjusted. As an implicit consequence of this method, all objects are rendered into the same render texture. This texture is adjusted in its dimensions to accommodate both, the left and the right eye image. Existing shaders, which rely on sampling the render texture, need to be modified to reflect this change.

The single-pass rendering mode can reduce the CPU time spent issuing draw calls by up to 50%. This is a very significant improvement, especially in the context of

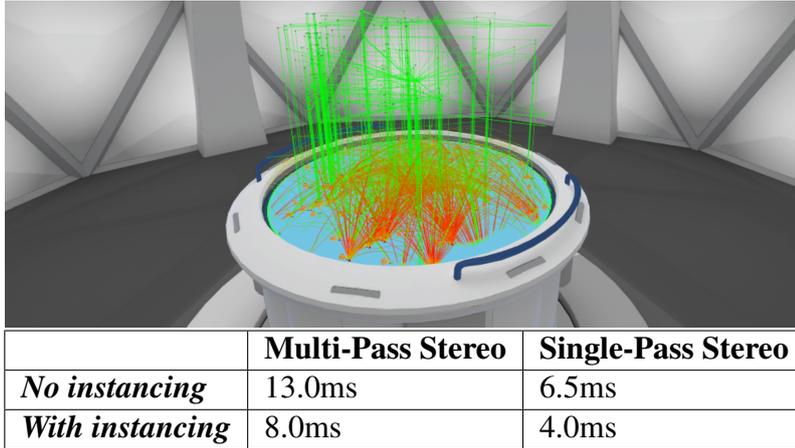


Figure 5.1: CPU computation time for the visualization of the RCE data set with all package dependencies and service connections shown.

software visualization, where scenes regularly exhibit a large number of objects. In conjunction with GPU instancing, these methods form a fundamental optimization step, reducing CPU computation time by up to 70%. Without them, the visualization of the RCE data set, with all its connections, would not be possible in the recommended frame rate of 90Hz(see Figure 5.1), which is unacceptable for a usable virtual reality application.

Chapter 6

Results

IslandViz was presented at the Free and Open Source software Conference (FrOSCon) 2017. This was a valuable chance to receive feedback from various users. However only a few people had a proper software engineering background with OSGi affinity. This made it difficult to acquire usable in depth feedback.

To sum it up, software developers have not been entirely convinced, that a medium any different than the classical desktop monitor with keyboard and mouse, could bring significant improvements in their day to day development efficiency. With that being said, the use case of providing a quick and coarse overview of an unknown software architecture in the VR medium was acknowledged by many to be quite possible with the presented implementation. The metaphorical mapping to represent modules with islands was found very intuitive and refreshing. Due to the lack of OSGi developers, the visualization of the service layer did not receive any substantial input.

A highly debated topic was also the multi-touch based navigation technique. It was observed, that usually older people had more problems with the navigation than younger people. More specifically, it took them significantly longer to master the navigation. However once learned, there have been less differences between the age groups.

A very frequently encountered problem was related to the hierarchical subdivision scheme. Users had difficulties in recognizing, when a subdivision has taken place, and the child elements could be interacted with.

The majority of users were surprised of how complex the visualized software was, especially with all packaged dependencies and service connections shown.

An important aspect of IslandViz was the minimization of motion sickness during use. To evaluate the potential success in this field, a Simulator Sickness Questionnaire was filled out by 16 participants. Each participant had an average VR exposure time of 10 minutes. In this time, the participants undertook a guided exploration of the RCE software project. Basic functionalities were explained, while the user could freely explore the software visualization and try out the different functions. It should be noted,

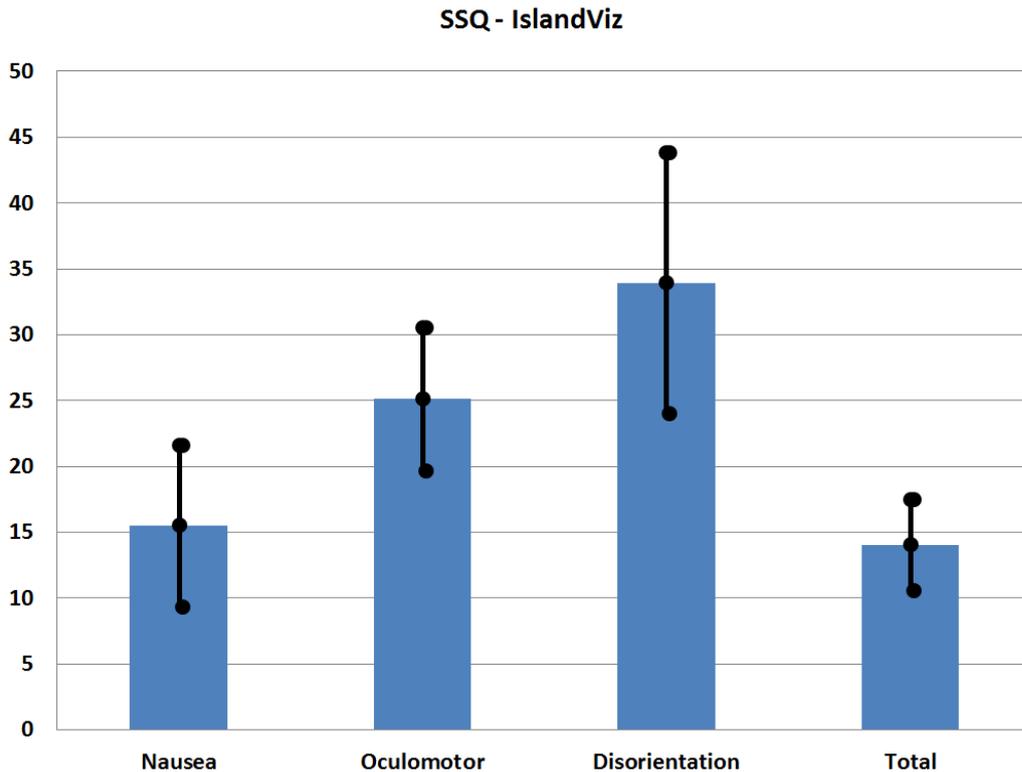


Figure 6.1: SSQ mean scores with standard deviation. 16 Participants, 10 minute exposure to guided exploration in IslandViz.

that for many this was their first experience with virtual reality. The SSQ results are summed up in Figure 6.1. The total score was ($M = 14.05$, $SD = 6.89$). The nausea related sub score was ($M = 15.5$, $SD = 12.22$), oculomotor sub score ($M = 25.11$, $SD = 10.84$) and the disorientation sub score ($M = 33.93$, $SD = 19.79$).

6.1 Discussion

Although not all gained insights were heavily related to the software architecture aspect, they provided valuable feedback on the state of the implementation. While the total SSQ score supports the claim, that the visualization does not introduce an excessive amount of motion sickness, having a look at the disorientation sub score helps in pinpointing the current issues.

Two possible factors can play into the high disorientation score. As has been stated, users complained about the difficulty in recognizing, which hierarchical level can be currently interacted with. This issue could very well impair the navigational abilities to a degree, that it would reflect in the disorientation sub score of the SSQ. The second contributing issue has already been identified in the conceptual phase as a possible

problem source. The limited visualization volume of the virtual table. When inspecting the visualization at high magnification factors, it is no longer possible to have contextual information about the remaining elements. The user is lost. A form of mini-map could be implemented, to help in bringing back some contextual information.

Regarding the issue with the transfer in hierarchical levels, its existence is not such a great surprise, as this functionality was reworked multiple times in the past. The implemented hierarchical system present at the FrOSCon demo was very recent and could surely use some additional parameter tweaking. With that being said, it would be very interesting to implement fixes for these two issues and do another study, with special focus on the disorientation score.

The observed trend, where younger users had less trouble with the navigation mechanism could also be expected, as the mechanism shares great similarities with touch based navigation on smartphones or tablets. A technology, young people have spent a significant portion of their lives with.

The user satisfaction regarding the island metaphor is a very positive sign, as one of the goals for this thesis, was to explore novel metaphors for software visualization. The practicability of the approach for aiding software comprehension tasks could not be clearly determined. While one reason surely lies in the difficulty of creating appropriate test scenarios with suitable participants, the main reason was the sparse functionality of the implementation. Additional software comprehension functionality would be needed to construct more real-life like scenarios of use.

However the extended presentation period revealed, that a software visualization in VR has great potential in the educational field. Here, insights into the world of software development can be, almost casually, conveyed to the public.

Chapter 7

Future Work

Although countless hours went into the implementation of the presented prototype, there is still a lot to be done. Due to time constraints, several things that have been conceptually discussed in chapter 4 did not make it into the implementation. Future work could resume on these ideas:

- Edge bundling approach for package dependencies.
- Accelerating the force directed layout algorithm.
- Island layout based on lexical similarity.
- Incorporating more visual metrics.
 - Map island shape and height to a metric
 - Regions representing exported packages should reflect this visually.
 - Different building types to differentiate between class, interface, enum.
- Implement water shader to enable the ocean filtering mechanism.
- Expand upon the radio metaphor for services.
- Implement hand based interactions.
- Focus+Context: When the visualization is zoomed in a mini-map or some other form of global positional information could help provide some of the lost context.
- Display additional class related information on the PDA.

Additionally, a few ideas which have not been previously discussed.

When the interaction system works with the users hands, it would be very interesting to experiment with a real physical table prop, which is aligned with the virtual table.

This form of passive haptic feedback could have the potential of increasing user's presence in the virtual environment.

The impact of visual fidelity on the resulting comprehension process is very sparsely studied. Two distinct visualization styles could be employed to study this question. While one would offer a more abstract and cleaner representation, the other would focus on a realistic representation.

A proper user study to evaluate the usability of the system is needed. However due to the limited availability of visualization software targeted explicitly at OSGi, it would be difficult to do a proper performance evaluation.

Due to the choice of the table metaphor, the visualization should be, at least on a conceptual level, easily portable to an Augmented Reality medium. Its performance and usability in it, could be the topic of future work.

And finally, the adaptation of this visualization metaphor to different software architecture types is a task, from which the whole software visualization community would benefit.

Bibliography

- [1] Tobias Marquardt. Extraktion und visualisierung von beziehungen und abhängigkeiten zwischen komponenten großer softwareprojekte, masterthesis. Technische Universitaet Dortmund.
- [2] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, 2014.
- [3] Andre L.C. Tavares and Marco Tulio Valente. A gentle introduction to osgi. *SIGSOFT Softw. Eng. Notes*, 33(5):8:1–8:5, August 2008.
- [4] Apachefelix, osgi framework implementation. <http://felix.apache.org>. Accessed: 2017-09-27.
- [5] Equinox, osgi framework implementation. <http://www.eclipse.org/equinox>. Accessed: 2017-09-27.
- [6] Knopflerfish, osgi framework implementation. <http://www.knopflerfish.org/>. Accessed: 2017-09-27.
- [7] Rym Mili and Renee Steiner. Software engineering - introduction. In *Revised Lectures on Software Visualization, International Seminar*, pages 129–137, London, UK, UK, 2002. Springer-Verlag.
- [8] Rainer Koschke. Software visualization in software maintenance, reverse engineering, and re-engineering: A research survey. *Journal of Software Maintenance*, 15(2):87–109, March 2003.
- [9] Stéphane Ducasse and Michele Lanza. The class blueprint: Visually supporting the understanding of classes. *IEEE Trans. Softw. Eng.*, 31(1):75–90, January 2005.
- [10] Pierre Caserta and Olivier Zendra. Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(7):913–933, July 2011.

- [11] T. Panas, T. Epperly, D. Quinlan, A. Saebjornsen, and R. Vuduc. Communicating software architecture using a unified single-view visualization. In *12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 217–228, July 2007.
- [12] M. A. D. Storey, K. Wong, and H. A. Muller. How do program understanding tools affect how programmers understand programs? In *Proceedings of the Fourth Working Conference on Reverse Engineering*, pages 12–21, Oct 1997.
- [13] George G. Robertson, Stuart K. Card, and Jack D. Mackinlay. Information visualization using 3d interactive animation. *Commun. ACM*, 36(4):57–71, April 1993.
- [14] Dean G Purcell and Alan L Stewart. The object-detection effect: Configuration enhances perception. *Attention, Perception, & Psychophysics*, 50(3):215–224, 1991.
- [15] Pourang Irani and Colin Ware. Diagramming information structures using 3d perceptual primitives. *ACM Trans. Comput.-Hum. Interact.*, 10(1):1–19, March 2003.
- [16] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Trans. Graph.*, 15(2):121–140, April 1996.
- [17] Andy Cockburn and Bruce McKenzie. Evaluating the effectiveness of spatial memory in 2d and 3d physical and virtual environments. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '02*, pages 203–210, New York, NY, USA, 2002. ACM.
- [18] Kenneth P. Herndon, Andries van Dam, and Michael Gleicher. The challenges of 3d interaction: a chi '94 workshop. 26:36–43, 10 1994.
- [19] Alfredo R. Teyseyre and Marcelo R. Campo. An overview of 3d software visualization. *IEEE Transactions on Visualization and Computer Graphics*, 15(1):87–105, January 2009.
- [20] Kevin Mullet. 3d or not 3d: “more is better” or “less is more”? (panel session). In *Conference Companion on Human Factors in Computing Systems, CHI '95*, pages 174–175, New York, NY, USA, 1995. ACM.
- [21] R. Wetzel and M. Lanza. Visualizing software systems as cities. In *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 92–99, June 2007.

- [22] C. Russo dos Santos, P. Gros, P. Abel, D. Loisel, N. Trichaud, and J. P. Paris. Metaphor-aware 3d navigation. In *IEEE Symposium on Information Visualization 2000. INFOVIS 2000. Proceedings*, pages 155–165, 2000.
- [23] Mapping information onto 3d virtual worlds. In *Proceedings of the International Conference on Information Visualisation, IV '00*, pages 379–, Washington, DC, USA, 2000. IEEE Computer Society.
- [24] Jock Mackinlay. Automating the design of graphical presentations of relational information. *ACM Trans. Graph.*, 5(2):110–141, April 1986.
- [25] Mark Green. The information cube: Using transparency in 3d information visualization. In *In: Proc. of the Third Annual Workshop on Information Technologies and Systems (WITS'93*, pages 125–132, 1993.
- [26] Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Trans. Softw. Eng.*, 18(11):957–968, November 1992.
- [27] J. I. Maletic, A. Marcus, and L. Feng. Source viewer 3d (sv3d) - a framework for software visualization. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 812–813, May 2003.
- [28] Hamish Graham, Hong Yul Yang, and Rebecca Berrigan. A solar system metaphor for 3d visualisation of object oriented software metrics. In *Proceedings of the 2004 Australasian Symposium on Information Visualisation - Volume 35, APVis '04*, pages 53–59, Darlinghurst, Australia, Australia, 2004. Australian Computer Society, Inc.
- [29] Marc Schreiber, Stefan Hirtbach, Bodo Kraft, and Andreas Steinmetzler. Software in the city: visual guidance through large scale software projects. In Stefan Kowalewski, editor, *Software Engineering 2013 : Fachtagung des GI-Fachbereichs Softwaretechnik, 26. Februar-1. März 2013 in Aachen. (GI-Edition ; 213)*, pages 213 – 224, 2013.
- [30] Pierre Abel, Pascal Gros, Cristina Russo dos Santos, Didier Loisel, and Jean-Pierre Paris. Automatic construction of dynamic 3D metaphoric worlds: an application to network management. In *EI 2000, SPIE Electronic Imaging, 23-28 January 2000, San Jose, USA / Proceedings of SPIE, Volume 3960 Visual Data Exploration and Analysis VII, Robert F. Erbacher, Philip C. Chen, Jonathan C. Roberts, Craig M. Wittenbrink, Editors, February 2000*, San Jose, UNITED STATES, 01 2000.

- [31] Kenichi Kobayashi, Manabu Kamimura, Keisuke Yano, Koki Kato, and Akihiko Matsuo. Sarf map: Visualizing software architecture from feature and layer viewpoints. In *ICPC*, 2013.
- [32] P. Caserta, O. Zendra, and D. Bodénès. 3d hierarchical edge bundles to visualize relations in a software city metaphor. In *2011 6th International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, pages 1–8, Sept 2011.
- [33] S. Alam and P. Dugerdil. Evospaces visualization tool: Exploring software architecture in 3d. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 269–270, Oct 2007.
- [34] H. J. Schulz and H. Schumann. Visualizing graphs - a generalized view. In *Tenth International Conference on Information Visualisation (IV'06)*, pages 166–173, July 2006.
- [35] C. Wetherell and A. Shannon. Tidy drawings of trees. *IEEE Transactions on Software Engineering*, SE-5(5):514–520, Sept 1979.
- [36] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, SE-7(2):223–228, March 1981.
- [37] Susanne Jürgensmann and Hans-Jörg Schulz. Poster: A visual survey of tree visualization. 01 2010.
- [38] H. J. Schulz, S. Hadlak, and H. Schumann. The design space of implicit hierarchy visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 17(4):393–411, April 2011.
- [39] T. Barlow and P. Neville. A comparison of 2-d visualizations of hierarchies. In *IEEE Symposium on Information Visualization, 2001. INFOVIS 2001.*, pages 131–138, Oct 2001.
- [40] Brian Johnson and Ben Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2Nd Conference on Visualization '91, VIS '91*, pages 284–291, Los Alamitos, CA, USA, 1991. IEEE Computer Society Press.
- [41] Richard Wettel and Michele Lanza. Visually localizing design problems with disharmony maps. In *Proceedings of the 4th ACM Symposium on Software Visualization, SoftVis '08*, pages 155–164, New York, NY, USA, 2008. ACM.

- [42] Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance, ICSM '04*, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.
- [43] Richard Wettel, Michele Lanza, and Romain Robbes. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 551–560, New York, NY, USA, 2011. ACM.
- [44] F. Fittkau, A. Krause, and W. Hasselbring. Exploring software cities in virtual reality. In *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, pages 130–134, Sept 2015.
- [45] David M Hoffman, Ahna R Girshick, Kurt Akeley, and Martin S Banks. Vergence-accommodation conflicts hinder visual performance and cause visual fatigue. *Journal of vision*, 8 3:33.1–30, 2008.
- [46] Dennis Ankrum R. *Visual Ergonomics in the Office - Guidelines. Occupational Health Safety. Volume 7, Issue 68, Pages 64-74.* 1999.
- [47] Markus Schedl, Peter Knees, Gerhard Widmer, Klaus Seyerlehner, and Tim Pohle. Browsing the web using stacked threedimensional sunbursts to visualize term co-occurrences and multimedia content. In *In: G. Kindlmann and L. Linsen (eds.) IEEE Visualization. Sacramento, CA: Poster Compendium, IEEE Computer*, pages 2–3. Society Press, 2007.
- [48] Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, pages 214–223, New York, NY, USA, 2005. ACM.
- [49] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2):129–137, March 1982.
- [50] Muye Yang and Robert P. Biuk-Aghai. Enhanced hexagon-tiling algorithm for map-like information visualisation. In *Proceedings of the 8th International Symposium on Visual Information Communication and Interaction, VINCI '15*, pages 137–142, New York, NY, USA, 2015. ACM.
- [51] Marinetraffic. <https://www.marinetraffic.com/>. Accessed: 2017-09-13.

- [52] Danny Holten and Jarke J. van Wijk. Force-directed edge bundling for graph visualization. In *Proceedings of the 11th Eurographics / IEEE - VGTC Conference on Visualization*, EuroVis'09, pages 983–998, Chichester, UK, 2009. The Eurographics Association & John Wiley & Sons, Ltd.
- [53] Weiwei Cui, Hong Zhou, Huamin Qu, Pak Chung Wong, and Xiaoming Li. Geometry-based edge clustering for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1277–1284, November 2008.
- [54] A. Lambert, R. Bourqui, and D. Auber. 3d edge bundling for geographical data visualization. In *2010 14th International Conference Information Visualisation*, pages 329–335, July 2010.
- [55] Ozan Ersoy, Christophe Hurter, Fernando Vieira Paulovich, Gabriel Cantareiro, and Alexandru Telea. Skeleton-based edge bundling for graph visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17:2364–2373, 2011.
- [56] M.J. Parks. *American Flow Mapping: A Survey of the Flow Maps Found in Twentieth Century Geography Textbooks, Including a Classification of the Various Flow Map Designs*. Georgia State University, 1987.
- [57] Waldo Tobler. Experiments in migration mapping by computer. 14:155–163, 04 1987.
- [58] A. Lhuillier, C. Hurter, and A. Telea. State of the art in edge and trail bundling techniques. *Comput. Graph. Forum*, 36(3):619–645, June 2017.
- [59] PA Eades. A heuristic for graph drawing. *Congressus numerantium*, 42:149–160, 1984.
- [60] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1998.
- [61] S R Anju and Surendran Subu. A study on different hand gesture recognition techniques. *International Journal of Engineering Research Technology*, Vol. 3 - Issue 4, 04 2014.
- [62] Jason Jerald. *The VR Book: Human-Centered Design for Virtual Reality*. Association for Computing Machinery and Morgan & Claypool, New York, NY, USA, 2016.
- [63] U. Schultheis, J. Jerald, F. Toledo, A. Yoganandan, and P. Mlyniec. Comparison of a two-handed interface to a wand interface and a mouse interface for fundamental

- 3d tasks. In *2012 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 117–124, March 2012.
- [64] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. A review of overview+detail, zooming, and focus+context interfaces. In *ACM Surveys*. Association for Computing Machinery, Inc., January 2008.
- [65] Jsonobject library. <https://github.com/mtschoen/JSONObject>. Accessed: 2017-09-17, License: GNU LGPL-2.1.
- [66] Quickgraph4unity library. <https://github.com/davidgutierrezpalma/quickgraph4unity>. Accessed: 2017-09-17, License: Microsoft Public License.
- [67] Trianglenet library. <https://triangle.codeplex.com/>. Accessed: 2017-09-17, License: MIT License.
- [68] Loup Verlet. Computer "experiments" on classical fluids. i. thermodynamical properties of lennard-jones molecules. *Phys. Rev.*, 159:98–103, Jul 1967.
- [69] M. Kavakli, I. Kartiko, J. Porte, and N. Bigoin. *Effects of digital content on motion sickness in immersive virtual environments*, pages 331–343. Athens Institute for Education and Research (ATINER), 2008.
- [70] Jerrold Prothero, M H Draper, Thomas Furness, Don Parker, and M J Wells. The use of an independent visual background to reduce simulator side-effects. 70:277–83, 04 1999.
- [71] Henry Been-Lirn Duh, Donald E Parker, and Thomas A Furness. An “independent visual background” reduced balance disturbance evoked by visual scene motion: implication for alleviating simulator sickness. In *Proceedings of the SIGCHI conference on human factors in computing systems*, pages 85–89. ACM, 2001.
- [72] Moshe Bar and Maital Neta. Humans prefer curved visual objects. *Psychological Science*, 17(8):645–648, 2006. PMID: 16913943.
- [73] David Navon. Forest before trees: The precedence of global features in visual perception. 9:353–383, 07 1977.
- [74] Wireframe shader by ucla game lab. <http://games.ucla.edu/resource/unity-wireframe-shader>. Accessed: 2017-09-26.
- [75] Chris Green. Improved alpha-tested magnification for vector textures and special effects. In *ACM SIGGRAPH 2007 Courses, SIGGRAPH '07*, pages 9–18, New York, NY, USA, 2007. ACM.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne fremde Hilfe und nur unter Verwendung der zulässigen Mittel sowie der angegebenen Literatur angefertigt habe. Mir ist bekannt, dass die Weitergabe von Rechten an dieser Arbeit oder von Auszügen aus dieser Arbeit an Dritte der Zustimmung von Herrn Prof. Dr. Arnulph Fuhrmann bedarf.

Ort, Datum

Rechtsverbindliche Unterschrift