Visualizing Software Architectures in Virtual Reality with an Island Metaphor

Andreas Schreiber[1](✉) and Martin Misiak[2]

[1] Intelligent and Distributed Systems, German Aerospace Center (DLR), Linder Höhe, 51147 Köln, Germany
Andreas.Schreiber@dlr.de

[2] TH Köln – University of Applied Sciences, Betzdorfer Straße 2, 50679 Köln, Germany
martin.misiak@th-koeln.de

Abstract. Software architecture is abstract and intangible. Tools for visualizing software architecture can help to comprehend the implemented architecture but they need an effective and feasible visual metaphor, which maps all relevant aspects of a software architecture and fits all types of software. We focus on the visualization of module-based software—such as OSGi, which underlies many large software systems—in virtual reality, since this offers a much higher comprehension potential compared to classical 3D visualizations. Particularly, we present an approach for visualizing OSGi-based software architectures in virtual reality based on an island metaphor. The software modules are visualized as islands on a water surface. The island system is displayed in the confines of a virtual table where users can explore the software visualization on multiple levels of granularity by performing intuitive navigational tasks. Our approach allows users to get a first overview about the complexity of the software system by interactively exploring its modules as well as the dependencies between them.

Keywords: Virtual reality · Software visualization · Software analysis

1 Introduction

Software is abstract and intangible. With increasing functionality, its complexity grows and hinders its further development. Visualization techniques, that map intangible software aspects onto visually perceivable entities, help to enhance the understandability and reduce the development costs of software systems [10]. Over the years a number of two and three dimensional visualization approaches have been proposed. However, the visualization of software in *virtual reality (VR)* still remains a sparsely researched field. While it offers a much higher comprehension potential compared to classical three dimensional visualizations, it also requires 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 substantially higher immersion degree, that this medium can achieve.

We present an approach for visualizing OSGi-based software projects in virtual reality. OSGi (Open Services Gateway Initiative), is a module-based, service-oriented framework specification for Java. OSGi is widely used in the *Eclipse* ecosystem. It centers on application development using modular units, called bundles. A bundle is a self-contained unit of classes and packages, which can be selectively made available to other bundles. The OSGi service layer handles communication between service components, which can reference or implement specific interfaces [18]. Popular implementations of the OSGi specification are *Apache Felix*, *Equinox*, or *Knopflerfish*.

Our goal is to provide a high level overview of the underlying software architecture to the user, while minimizing the experienced simulator sickness and enabling an intuitive navigation. Although our implementation targets OSGi-based software, the presented concepts are applicable to a large variety of use cases, such as other module-based software architectures.

Our main contributions are:

- We present a novel real-world metaphor based on an island system for visualizing module-based software architectures (Sect. 3).
- We present our approach for the exploration of software projects in VR, with focus on high level comprehension and improvements to user comfort (Sect. 4).

2 Software Visualization

Software visualization is a very large research field. Existing work can be classified based on multiple categories. A differentiation between static and dynamic aspects of a software can be made. While dynamic aspects capture information of a particular program run, static aspects are valid for all execution paths of a software. Additionally, they can be extended to capture the entire evolution of a software architecture.

Software visualization can be made on roughly three different levels of abstraction [7]. While the lowest abstraction level deals with the source code, the highest abstraction level deals with the entirety of the software architecture and belongs to the most important in software visualization [1]. They convey the underlying hierarchical component structure, the relationships between these components and the visual representation usually contains some form of code quality metrics.

A software visualization can consist of one or more views. Each view can use its own visualization approach and can therefore focus on different aspects of the software. Multi-view, as opposed to single-view approaches, can represent a broad range of information of varying granularity levels, however at the cost of imposing a significant cognitive burden and making a communication on common grounds between users more difficult [17].

Visualizations can be made in the two dimensional and three dimensional space. While 2D visualizations are easier to navigate and interact with, they do not scale particularly well for large data sets. To avoid a cluttered view, 2D visualizations rely on multiple views. 3D approaches resort to the added dimension to increase the information density of the visualization, without exposing the user to any additional cognitive load, as the task of processing 3D objects is completely shifted to the perceptual system [13]. They are more effective at identifying substructures and relationships between objects [5,22] and represent real-world metaphors more closely.

3 Visual Metaphors and Environment

3.1 Islands

After careful analysis of the requirements the visualization had on the metaphor, the concept of mapping the software system onto an island metaphor was chosen. The entire 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.

The island metaphor provides a hierarchical structure with three different levels (island, region, and building). The navigation between these layers should be based on our natural understanding of spatial relationships and be therefore dependent on the relative size of the elements in the users view frustum. Hence, the transition between the levels happens 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.

The metaphor is flexible enough to be extended for more than three abstraction levels. 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.

The islands metaphor has several advantages for software visualization. Islands express the aspect of decoupled entities, coexisting in the same environment very clearly, which makes them a good candidate for representing software modules. Additionally, islands can be relocated 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.

The islands of our visualization metaphor are aimed at having a high resemblance to their real-world counterparts and in thus, emphasizing the plausibility of the metaphor. 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 (Figs. 1 and 2). The individual cells of a region are designed to provide enough accommodation area

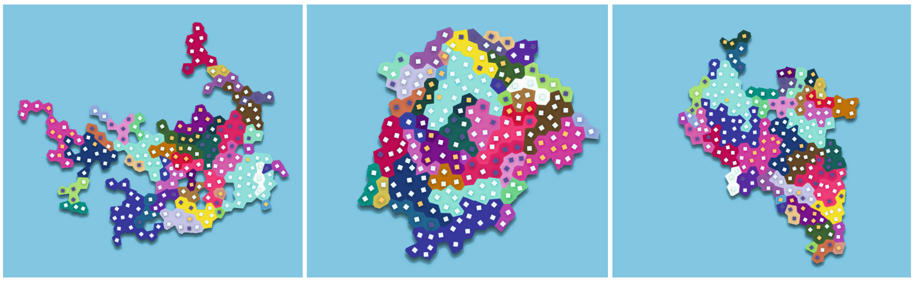


Fig. 1. (left) A minimal cohesion factor leads to very rugged islands with many holes. (middle) A very high cohesion factor reduces holes greatly and creates compact islands. (right) Our dynamic cohesion factor, combined with the claiming of large regions first, the island preserves some of the ruggedness, yet it minimizes holes.

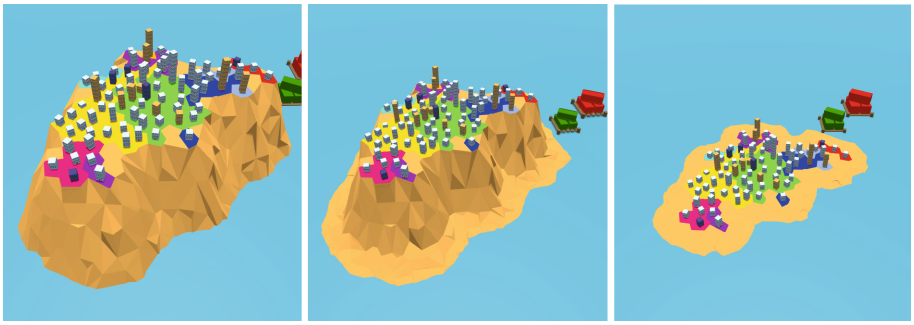


Fig. 2. A range of different coast shapes, created with specific height profiles.

for buildings to be placed on top. To maximize their perceivability from afar a multi-storey building representation is chosen. For the implemented prototype, a *Lines of Code* metric was chosen, where for every n lines of code, a storey is added to the building.

Island Construction. The island construction is based on claiming cells in a Voronoi diagram (analogous to the work of Yang and Biuk-Aghai [23], while we use a Voronoi diagram instead of a hexagonal grid as the underlying tile structure). Additionally, no hierarchical claiming is performed, as all regions of an island are considered equivalent. This reflects the way packages are interpreted by Java, as the hierarchical naming convention is only relevant from a developers perspective.

The first step in the construction is to create a Voronoi diagram from a point distribution. The most aesthetically pleasing islands were achieved with points exhibiting a blue noise characteristic. 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 next to already existing entities can be claimed.

To create rugged and irregular shapes for the package representations, the cells are selected probabilistically. To avoid non-continuous regions induced by a random selection mechanism, we employ an estimating function as described by Yang and Biuk-Aghai [23].

Before a new tile is selected, each eligible cell counts its already claimed neighbors. If a cell is surrounded with n claimed neighbors, the probability of it being a hole grows with n. A score S_n is calculated for each candidate, based on

$$S_n = b^n \tag{1}$$

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. Higher b values result in less holes, but also more regular and compact shapes.

To preserve the rugged appearance of an island, we use a simple extension of the cohesion factor. Defining b_{min} and b_{max}, the cohesion factor can be varied on a per region basis, depending 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 (Fig. 1 right). From a usability perspective, this layout is more advantageous for VR based interaction, as smaller regions are harder to select when surrounded by larger ones.

Once all packages have claimed their cells, the coast area can be added. This is done by claiming neighboring cells of the existing island boundary. Each time the boundary is expanded outwards a new height value is associated with its cells. A user defined height profile controls this process (Fig. 2), 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.

3.2 Visualization of Dependencies

Dependencies Between Modules. Due to the architecture oriented focus of the presented software visualization, the dependencies between individual modules are of high importance. 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. To visualize them, two orthogonal types of approaches are considered. An explicit and an implicit dependency visualization.

Explicit Visualization. Building on the simplicity of straight lines, import and export arrows (Fig. 3) are used to explicitly visualize the package dependencies

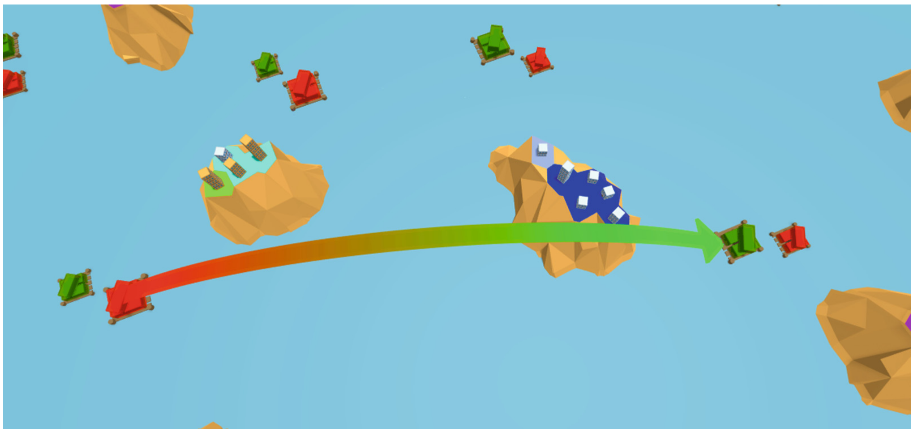


Fig. 3. Explicit dependency visualization via an arced arrow. The island on the right imports packages from the bundle on the left.

between bundles. In the geographic context, such arrows are encountered in flow maps [11] and visualize the movement of various resources or entities, from one point to another, while the arrow width is proportional to the moved volume. The resulting dependency visualization is similar to a discrete flow map, as implemented by Tobler [19]. 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.

Implicit Visualization. We use the island adjacency to implicitly represent the dependency strength (Fig. 4). The island layout is computed with the help of an iterative, force-directed graph layout algorithm, based on the work of Eades [3]. 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 nodes and the variables c_1 and c_2.

$$F_a = c_1 \cdot log(d/c_2) \tag{2}$$

F_a can be interpreted as a spring like force defined by the stiffness factor c_1 and the unloaded spring length c_2. In contrast to Eades, who focused on layouts with uniform edge lengths, we compute c_2 on a per edge basis to reflect the relative dependency strength between the nodes in question as

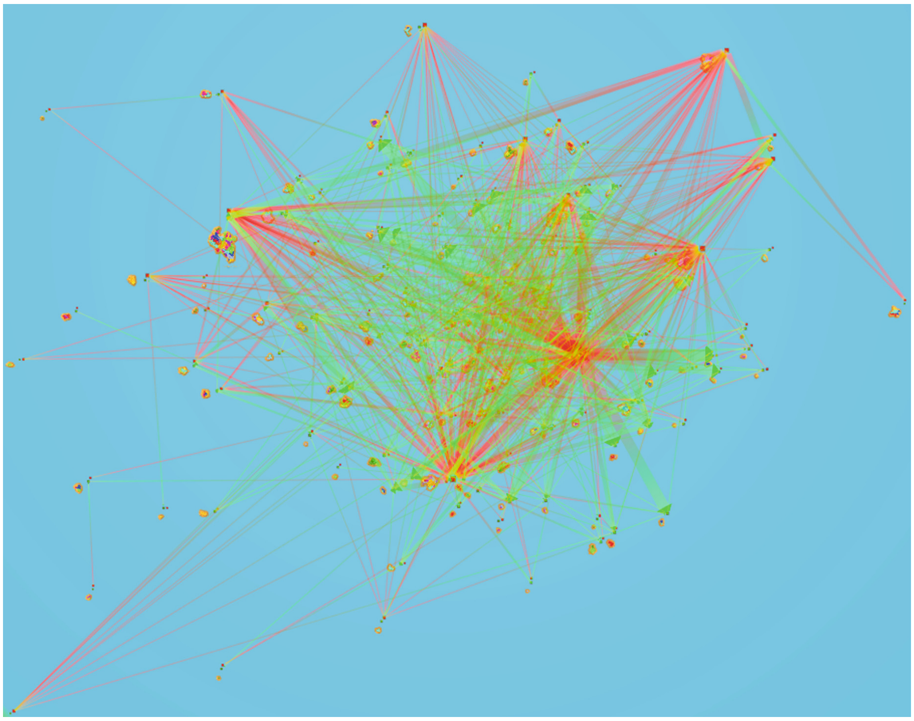


Fig. 4. Island placement based on the presented force-directed layout algorithm. Islands with the highest dependencies are accumulated in the middle, while independent islands are pushed outwards.

$$c_2 = c_3 \cdot \frac{i_{max}}{i_A + i_B}, \tag{3}$$

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. As the dependency between two nodes increases, the resulting unloaded spring length c_2 decreases. The user defined variable c_3 represents the lower bound on the spring length. 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.

For nonadjacent nodes a repulsion force F_r is introduced.

$$F_r = \frac{c_4}{d^2} \tag{4}$$

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

Once all forces for a particle have been accumulated, they are applied 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 $m = 1$.

Visualization of Service Dependencies. The main entities of the OSGi service layer are *service interfaces* and *service components*. As these components are linked to Java class types, we visualize them as special building types. To visualize the relationships between the service entities, we introduce the service connection node. These nodes hover above the service interface and service component buildings at a certain height and act as 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.

SIN nodes assume a central role as they form connections to the other two node types. 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.

3.3 Virtual Table

We choose a virtual table metaphor to integrate the software visualization into the virtual environment. 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. 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 is not a disadvantage, since it enforces to show the visualization in a space saving representation. While it can be helpful to see the fine grained details of software artifacts, it is the higher abstraction levels which contribute mostly to program and architecture comprehension.

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, as the room always provides a stable frame of reference [2,12]. This is especially important for the usability of the system for software comprehension, as users can stay longer immersed in the virtual environment without interrupting their train of thought.

As has been noted in Sect. 3.1, 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 will not be displayed. Only content inside of the table bounds is visible. This poses a 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 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.

3.4 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 functionality is implemented by simply discarding all rendered fragments of the software visualization, that exceed the table radius. When designing the environment, we avoided introducing an excessive brightness contrast. This helps in minimizing the "godray" effect, attributed to the used Fresnel lenses.

4 Interaction and Navigation

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. We build upon the available positional information of the input devices and integrate all interaction possibilities into the environment itself. This reduces the reliance on various button presses and keeps the user interface simple (Fig. 5).

The software visualization appears 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. Our navigational technique encompasses translation, rotation and scaling. It is very similar to the "Two-Handed Interface" technique described by Schultheis et al. [15]. In contrast to their work we constrain the rotation to one axis (Fig. 6 bottom). The scaling operation is especially important, as zooming is directly tied to the transition between the individual abstraction

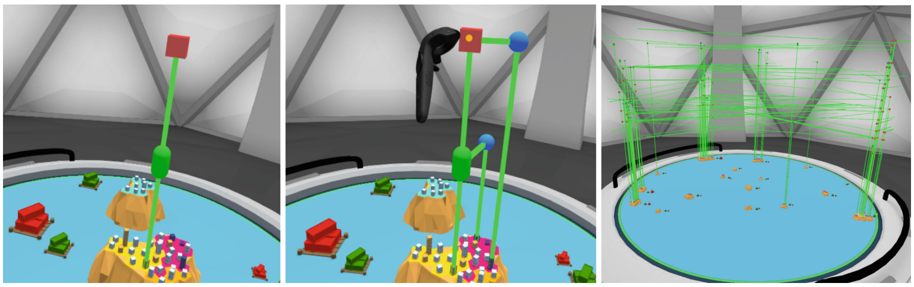


Fig. 5. (left) The two service nodes signalize that the component provides, as well as references a service interface. (middle) The connections to the two service interfaces are shown. Both are placed at different heights as the blue service interface nodes are assigned to two distinct service slices. (right) Multiple service connections shown simultaneously. (Color figure online)

layers of the software architecture. This mode of navigation basically follows a level of detail 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.

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 grabs the visualization and drags it in the direction he wishes to translate, releasing it again when finished.

To perform rotation and scaling, 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. In order to rotate the visualization, both controllers are moved in a circular motion around the pivot point. 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.

Due to the direct manipulation technique, the visualization is more likely to be interpreted as an object and not as part of the stable reference frame [6], reducing discomfort upon navigation.

4.1 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. The display of textual information in a virtual reality environment however, is a challenging task, due to the severe resolution limitations of current head-mounted displays (HMDs). For text to be clearly readable, it has to occupy a

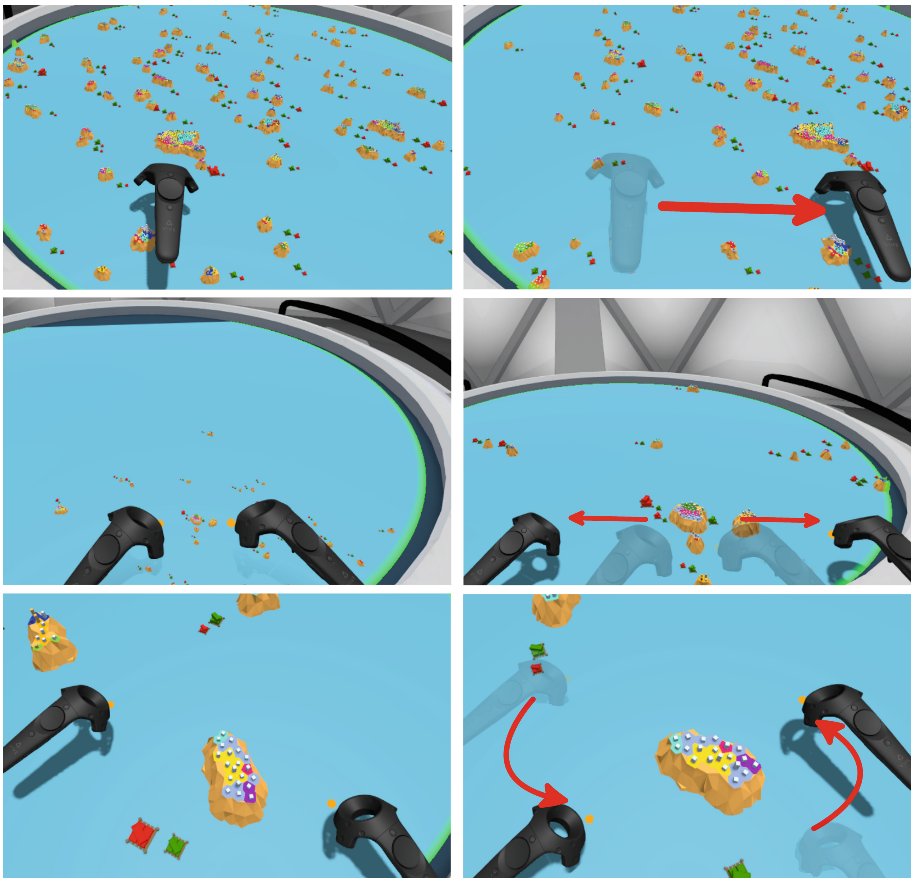


Fig. 6. Navigational operations: (top) Translation. (middle) Scale. (bottom) Rotation.

significantly larger angle in the users field of view, which prohibits the display of large quantities of textual information in the virtual world.

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, we display only the text labels of elements which are being hovered over by the controllers. This provides a better control and frees up the HMD for performing only navigational tasks, as opposed to gaze based selection. To display the names of elements further away, a laser pointer functionality is added.

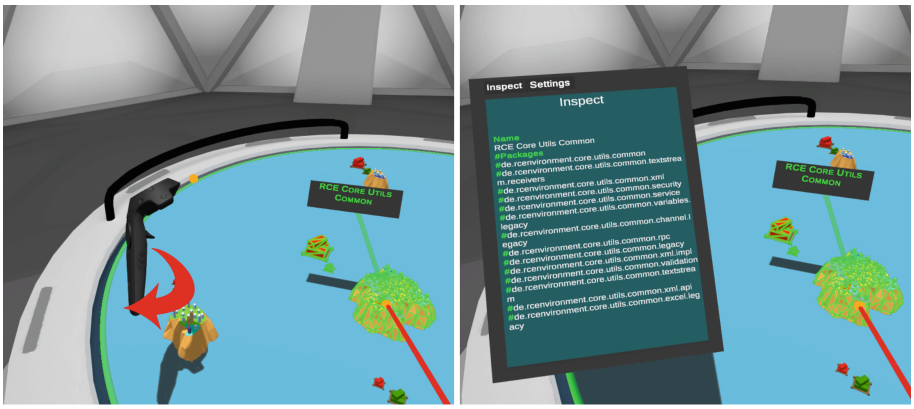
Each time a label is displayed, it adjusts its scale to take up a constant amount of display space, irrespective of its actual distance to the user. Thus, ensuring a consistent readability. However once a label is displayed, it will not further change its scale. This allows the labels to be perceived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

4.2 Virtual Personal Digital Assistant

While world space anchored text labels are good for displaying object names, they are not suitable for the display of larger amounts of text. However such a functionality is greatly needed, as some information are best presented 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 and must be placed somewhere, where it is not occluded by the environment or vice versa. To avoid the placement problem, we anchor the display to the virtual body of the user. More specifically, to his hand. 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) [6]. The panel can be thought of as a *virtual Personal Digital Assistant (VPDA)* or tablet (Fig. 7). To avoid unnecessary occlusion and unintentional interactions, the VPDA 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 VPDA, a classical tabs and windows system is employed, to organize information as well as provide additional functionality.



```
Name
RCE Core Utils Common
#Packages
#de.rcenvironment.core.utils.common
#de.rcenvironment.core.utils.common.textstrea
m.receivers
#de.rcenvironment.core.utils.common.xml
#de.rcenvironment.core.utils.common.security
#de.rcenvironment.core.utils.common.service
#de.rcenvironment.core.utils.common.variables.
legacy
#de.rcenvironment.core.utils.common.channel.l
egacy
#de.rcenvironment.core.utils.common.rpc
#de.rcenvironment.core.utils.common.legacy
#de.rcenvironment.core.utils.common.xml.impl
#de.rcenvironment.core.utils.common.validation
#de.rcenvironment.core.utils.common.textstrea
m
#de.rcenvironment.core.utils.common.xml.api
#de.rcenvironment.core.utils.common.excel.leg
acy
```

Fig. 7. To activate the VPDA, the underside of the controller is rotated into the users field of view, providing access to additional textual information and functionality.

5 Implementation

We developed our software prototype, *IslandViz*, using *Unity3D*. The targeted HMD is the *HTC Vive*. We validated our approach by visualizing the OSGi-based software project `RCE`[1]. To extract all relevant information from the software we used a tool based on the work of Seider et al. [16].

6 Related Work

Maletic et al. [8] presented a visualization of C++ code in a virtual environment. Classes are represented as floating platforms upon which additional geometric shapes are placed to visualize attributes and methods. While inheritance is implemented via platform adjacency, other dependency types use explicit connections. The presented system is displayed inside a CAVE environment and showcased only very simple software systems.

Fittkau et al. [4] proposed an approach for a live trace visualizations using a city metaphor in virtual reality. The visualization is presented to the user in a head mounted display (*Oculus DK1*). Since the used hardware does not incorporate any positional tracking, the visualized content is additionally transformed via gesture based controls. In combination with a gaze driven pointer, objects can be selected and interacted with.

Schreiber and Brüggemann [14] introduced an approach for visualizing software modules using the metaphor of electrical components. Modules are represented as blocks and the containing packages are stacked on top of each module. The stacked modules are visualized in virtual reality by various placement algorithms based on the relationship between modules. Modules and packages can be selected an interactively explored by showing service modules, classes, and dependencies between modules and packages.

Recently, Merino et al. [9] and Vincur et al. [20,21] presented a VR visualization for object oriented software (Java, C/C++) using a city metaphor. The approaches rely on VR hardware capable of positional tracking, as the main navigational mechanism is physical movement and interaction is based on the controller positions. In contrast, the main navigational mechanism in our work is the explicit transformation of the visualization itself and is therefore independent of the available physical tracking space. Additionally, we also support positional tracking.

7 Conclusion and Future Work

We presented our approach for exploring OSGi-based software architectures in virtual reality. Based on user feedback, we conclude that a software visualization in VR has great potential in the educational field, as insights into the world of software development can be, almost casually, conveyed to the public.

[1] http://rcenvironment.de/.

Further evaluation is needed to determine the practicability of the approach in aiding software comprehension tasks. For this, additional software metrics and functionality need to be incorporated, to construct more realistic scenarios of use.

An advantage of the islands metaphor is the presence of the ocean as the "base plane", spanning over the individual elements. Water possesses interesting optical properties, which could be used for filtering tasks. This could allow the user to reduce the visual complexity by submerging specific islands under the ocean.

A hand based interaction scheme should be investigated and compared to the existing controller scheme. Additionally, a hand based interaction would allow the use of a real physical table prop. This prop would be aligned to the virtual table and provide a form of passive haptic feedback, which has the potential of increasing the users presence in the virtual environment.

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.

With the recent release of JDK 9, Java finally receives native support for modules. With a slight change, *IslandViz* will be able to visualize a large quantity of future Java based projects, which is an exciting prospect.

References

1. Caserta, P., Zendra, O.: Visualization of the static aspects of software: a survey. IEEE Trans. Vis. Comput. Graph. **17**(7), 913–933 (2011). https://doi.org/10.1109/TVCG.2010.110
2. Duh, H.B.L., Parker, D.E., Furness, T.A.: An "independent visual background" reduced balance disturbance envoked by visual scene motion: implication for alleviating simulator sickness. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 85–89. ACM (2001)
3. Eades, P.: A heuristic for graph drawing. Congressus Numerantium **42**, 149–160 (1984)
4. Fittkau, F., Krause, A., Hasselbring, W.: Exploring software cities in virtual reality. In: 2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT), pp. 130–134, September 2015
5. Irani, P., Ware, C.: Diagramming information structures using 3D perceptual primitives. ACM Trans. Comput.-Hum. Interact. **10**(1), 1–19 (2003). https://doi.org/10.1145/606658.606659
6. Jerald, J.: The VR Book: Human-Centered Design for Virtual Reality. Association for Computing Machinery and Morgan & Claypool, New York (2016)
7. Koschke, R.: Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey. J. Softw. Maintenance **15**(2), 87–109 (2003). https://doi.org/10.1002/smr.270
8. Maletic, J.I., Leigh, J., Marcus, A., Dunlap, G.: Visualizing object-oriented software in virtual reality. In: Proceedings 9th International Workshop on Program Comprehension, IWPC 2001, pp. 26–35. IEEE Computer Society, Washington, DC (2001)

9. Merino, L., Ghafari, M., Anslow, C., Nierstrasz, O.: CityVR: gameful software visualization. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 633–637. IEEE (2017). http://scg.unibe.ch/archive/papers/Meri17c.pdf
10. Mili, R., Steiner, R.: Software engineering - introduction. In: Revised Lectures on Software Visualization, International Seminar, pp. 129–137. Springer, London (2002). http://dl.acm.org/citation.cfm?id=647382.724792
11. Parks, M.: 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). https://books.google.de/books?id=mgRENwAACAAJ
12. Prothero, J.D., Draper, M.H., Parker, D.E., Wells, M.J.: The use of an independent visual background to reduce simulator side-effects. Aviat. Space Environ. Med. **70**, 277–83 (1999)
13. Robertson, G.G., Card, S.K., Mackinlay, J.D.: Information visualization using 3D interactive animation. Commun. ACM **36**(4), 57–71 (1993). https://doi.org/10.1145/255950.153577
14. Schreiber, A., Brüggemann, M.: Interactive visualization of software components with virtual reality headsets. In: 2017 IEEE Working Conference on Software Visualization (VISSOFT) (2017)
15. Schultheis, U., Jerald, J., Toledo, F., Yoganandan, A., Mlyniec, P.: 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), pp. 117–124, March 2012
16. Seider, D., Schreiber, A., Marquardt, T., Brüggemann, M.: Visualizing modules and dependencies of OSGI-based applications. In: 2016 IEEE Working Conference on Software Visualization (VISSOFT), pp. 96–100. IEEE (2016)
17. Storey, M.A.D., Wong, K., Muller, H.A.: How do program understanding tools affect how programmers understand programs? In: Proceedings of the Fourth Working Conference on Reverse Engineering, pp. 12–21, October 1997
18. Tavares, A.L., Valente, M.T.: A gentle introduction to OSGI. SIGSOFT Softw. Eng. Notes **33**(5), 8:1–8:5 (2008). https://doi.org/10.1145/1402521.1402526
19. Tobler, W.: Experiments in migration mapping by computer. Am. Cartographer **14**, 155–163 (1987)
20. Vincur, J., Navrat, P., Polasek, I.: VR city: software analysis in virtual reality environment. In: 2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 509–516, July 2017
21. Vincur, J., Polasek, I., Navrat, P.: Searching and exploring software repositories in virtual reality. In: Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology, VRST 2017, pp. 75:1–75:2. ACM, New York (2017). https://doi.org/10.1145/3139131.3141209
22. Ware, C., Franck, G.: Evaluating stereo and motion cues for visualizing information nets in three dimensions. ACM Trans. Graph. **15**(2), 121–140 (1996). https://doi.org/10.1145/234972.234975
23. Yang, M., Biuk-Aghai, R.P.: Enhanced hexagon-tiling algorithm for map-like information visualisation. In: Proceedings of the 8th International Symposium on Visual Information Communication and Interaction, VINCI 2015, pp. 137–142. ACM, New York (2015). https://doi.org/10.1145/2801040.2801056