IslandViz: A Tool for Visualizing Modular Software Systems in Virtual Reality

Martin Misiak*, Andreas Schreiber†, Arnulph Fuhrmann*, Sascha Zur†, Doreen Seider†, and Lisa Nafeie†
* Institute of Media and Imaging Technology
TH Köln
Köln, Germany
Email: martin.misiak@th-koeln.de, arnulph.fuhrmann@th-koeln.de
† Intelligent and Distributed Systems
German Aerospace Center (DLR)
Köln, Germany
Email: andreas.schreiber@dlr.de, sascha.zur@dlr.de, doreen.seider@dlr.de, lisa.nafeie@dlr.de

***Abstract*—We propose the tool `IslandViz` for exploring modular software systems in virtual reality. We use an island metaphor, which represents every module as a distinct island. The resulting 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 navigational tasks. Our approach allows users to get a first overview about the complexity of an OSGi-based software system by interactively exploring its modules as well as the dependencies between them.**

I. 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 [1]. 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 higher comprehension potential compared to classical three dimensional visualizations [2], [3], it also requires a different approach, as the medium provides a higher immersion degree [4].

We present an approach for visualizing OSGi-based software projects in virtual reality. 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.

In our previous work [5] we described the island metaphor, which we elaborate on further as follows:

- A novel real-world metaphor based on an island system for visualizing module-based software systems.
- An approach for the exploration of software projects in VR, with focus on high level overview and improvements to user comfort and usability.

II. Visualization Approach

A. Island Metaphor

The visualization metaphor has to be expressive enough to provide mappings for all relevant software artifacts. In the case of OSGi-based software, the metaphor needs to account for the following artifacts: *class types*, *packages*, *bundles*, *service components*, and *service interfaces*. We are interested in the import and export relations of individual bundles, as well as the referencing and providing relationships between service components and their respective interfaces. The metaphor must enable a software inspection on multiple abstraction levels, however its main emphasize should be the bundle layer, as it forms a central aspect of OSGi.

We propose an island metaphor for the visualization of OSGi-based software systems (Figure 1). The entire software 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, symbolizing the individual class types. The regions provide enough space to accommodate all contained buildings without overlapping, and hence the overall size of an island is proportional to the number of class types inside of a bundle. To maximize the building visibility from afar, a multi-storey representation is chosen, where a *Lines of Code* metric dictates the number of storeys.

To emphasize the plausibility of the metaphor, our islands strive for a high resemblance with their real-world counterparts. To this end, they are composed of irregularly shaped regions, similar to countries when seen on a map, which give each island a very distinct appearance. This offers strong navigational cues to the user, as bundles and packages can be identified based on their shape alone, reducing the dependency on naming labels.

B. Dependencies

1) Bundle Dependencies: Building on the island metaphor, ports situated along the island coast line are used to manage the incoming and outgoing package dependencies. The

978-1-5386-8292-0/18/$31.00 ©2018 IEEE
DOI 10.1109/VISSOFT.2018.00020

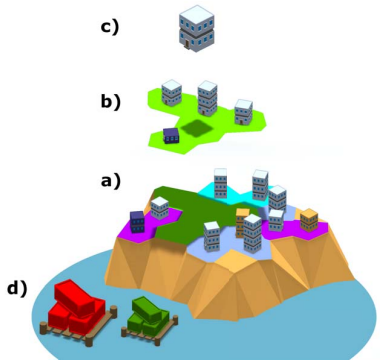

Fig. 1. Island metaphor: a) Bundles are represented as islands. b) Packages are represented as regions. c) Class types are represented as buildings d) Package imports and exports are represented with the help of ports.

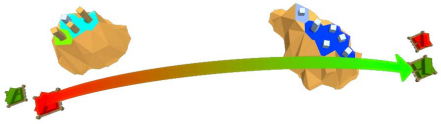

Fig. 2. Package dependencies are visualized via arced arrows. The island on the right imports packages from the island on the left.

dependencies themselves, are visualized with the help of arrows (Figure 2), similar to a discreet flow map [6]. 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 change of curvature is constant for all arrows. 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.

2) 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 (Figure 3 left). A service component can reference as well as provide for multiple service interfaces. We represent these relationships with a straight line connection. However to avoid intersections and to use the vacant height dimension, we reroute the connections through three distinct node types: service interface (*SIN*), service provide (*SPN*) and service reference nodes (*SRN*). These nodes hover above the buildings in question and are assigned to different *height slices*.

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 *height slice*. Due to this design, there are no connections going across individual height layers. Combined with the benefits of motion

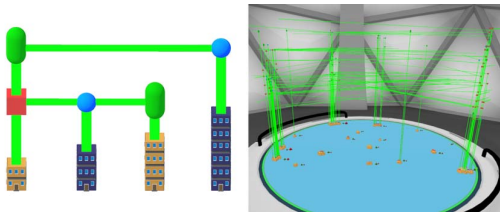

Fig. 3. Left: Service components (orange) and service interfaces (blue) are represented as different building types. The are connected with the help of service interface/provide/reference nodes (blue sphere, green capsule, red cube). Right: Service groups are distributed across individual height layers.

parallax and stereoscopic depth cues given by VR technology, a reduction of visual complexity is achieved [3].

C. Virtual Table

One of the main causes for simulator sickness is a sensory conflict between the visual and vestibular system, where the virtual environment is moving, but the person is not [7]. In the case of an environment filling visualization, such conflicts are encountered very often, as translation, rotation and scaling are performed. This problem can be alleviated by introducing a visual rest frame, which is consistent with cues from the vestibular system [8], [9].

To this end, our software visualization is presented on top of a virtual table situated in an arbitrary room (Figure 4). The entire content of the visualization is confined to the extents of the table, while the room acts as a stable rest frame. This way, the user experiences less sensory conflicts when inspecting the software on different abstraction levels or from different perspectives, as only the visualization in the confinements of the table has to be changed. The disadvantage of our approach is the artificial restriction of the available visualization space, which leads to a loss of context when inspecting fine granular software artifacts. On the other hand, the limited visualization space does not force the user to move around excessively to view the desired information, which makes the approach suitable for a seated, as well as a standing VR experience.

III. Navigation and Interaction

Due to the use of VR 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 very small elements, or a physical restriction of the employed VR hardware (such as available tracking space, or a limited cable length) is given. Therefore the ability to additionally manipulate the visualization itself is crucial.

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 (Figure 5 top, bottom, middle). It is very similar to the *Two-Handed Interface* technique described by Schultheis et al. [10]. In contrast to their work we constrain the rotation to

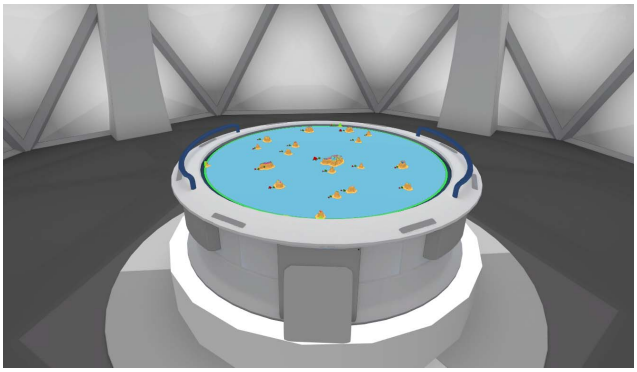

Fig. 4. The virtual environment consists of a room with a table at its center. The software visualization is displayed only inside of its bounds.

the up axis defined by the table normal. The visualization can be translated along the axes defined by the table plane. This usually results in left, right, forward, and backward panning, while the translation in the height dimension is prohibited. To apply the translation, the user grabs the visualization and drags it in the direction they wish to translate, releasing it again when finished.

To do rotation and scaling, the visualization is 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. The scaling operation is especially important in our implementation, as zooming is directly tied to the transition between the individual abstraction layers of the software (island, region, and building). This mode of navigation 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. 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 rest frame [4], reducing discomfort upon navigation.

Severe resolution limitations of current VR headsets prohibit the display of large quantities of textual information. We deal with this limitation in two ways.

First, we display only name labels of objects, which are being directly interacted with the controllers. To ensure a consistent readability, each label is scaled to cover a constant amount of display space. Objects further away can be interacted with via a laser pointer.

Second, for larger amounts of text, we use a display panel. It is attached to the non-dominant hand of the user, so it can be

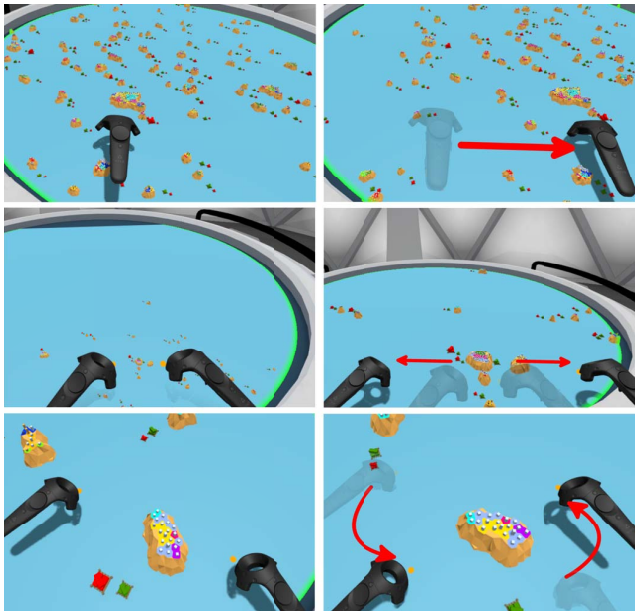

Fig. 5. (top) Translation. (middle) Scale. (bottom) Rotation.

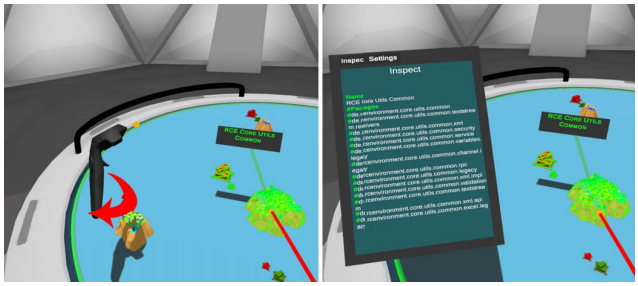

Fig. 6. To activate the VPDA, the underside of the controller is rotated into the users field of view, providing access to additional textual information and functionality.

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 [4]. The panel can be thought of as a *Virtual Personal Digital Assistant (VPDA)* (Figure 6). 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.

IV. Implementation

We developed our software prototype *IslandViz*[1] using the game engine *Unity3D*. Our approach was validated by visualizing the OSGi-based software project RCE[2] on the *HTC*

[1] https://github.com/DLR-SC/island-viz
[2] http://rcenvironment.de/

Fig. 7. (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.

Vive and *Oculus Rift* VR systems. To extract all relevant information from the source code of the visualized software, we used a tool based on the work of Seider et al. [11].

A. Island Construction

The island construction is based on claiming cells in a Voronoi diagram and is analogous to the work of Yang et al. [12]. However we use a Voronoi diagram instead of a hexagonal grid as the underlying tile structure and do no hierarchical claiming.

As a first step the Voronoi diagram is created from a predefined point distribution. For this, as well as a later triangulation step, we use a library based on the work of Shewchuk [13]. The most aesthetically pleasing islands were achieved with points exhibiting a blue noise characteristic.

In the next step, each package claims one cell per contained class in the Voronoi diagram. 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 use an estimating function as described by Yang et al. [12]. 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 (Figure 7 right). From an 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 is added. This is done by claiming neighboring cells of the existing island boundary. Each time the boundary is expanded outwards a new height value, stored in a user defined height profile, is associated with its cells. The coast area reinforces our metaphor and its shape could be used in future work to reflect a bundle based metric. In the final construction step, a polygonal mesh is generated from all claimed cells in the Voronoi diagram using triangulation.

B. Island Layout

The island layout is computed with the help of an iterative, force-directed graph layout algorithm, based on the work of Eades [14]. It reflects the strength of existing package dependencies (edges) between the islands (nodes). We chose this layouting criterion, as the service dependencies are less prone to overlapping and therefore profit less from an optimized layout. 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 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

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

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 is the number of packages bundle 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. F_a is applied to both nodes, only if they are interdependent. If either i_A or i_B is zero, the attraction force is applied only to the importing node. For nonadjacent nodes, a repulsion force based on an inverse-square law is introduced. Because of the applied forces, islands without package dependencies and solely exporting ones are placed on the outskirts of the layout, while importing islands are accumulated in the middle (Figure 8).

V. Related Work

The software visualization field has made extensive use of real-world metaphors. Among the most frequently used is the city metaphor, which was influenced by the work of Wettel et al. [15]. Although we use a different metaphor, we share its building-based class representation. The work of Panas et al. [16] visualizes source code directories with the help of landscape plates, which can hold multiple cities. This

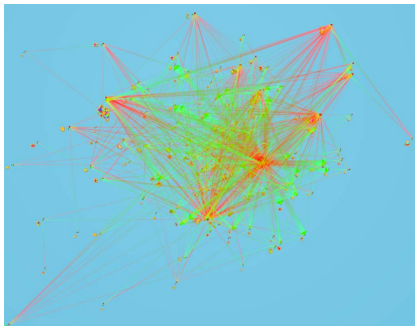

Fig. 8. Island placement based on the presented force-directed layout algorithm. Islands with the highest number of package dependencies are accumulated in the middle, while independent islands are pushed outwards.

could be considered metaphorically similar to our work, as the landscapes are visually separated by an optional layer of water. Kuhn et al. [17] cluster software artifacts, based on lexical similarity, to create 2D cartographic software maps. While the cartographic metaphor is similar to ours, we create distinct islands, based on hierarchical information.

In recent years, a number of software visualization approaches for VR have been published. Fittkau et al. [18] proposed a live trace visualization using a city metaphor. The visualization is displayed inside the *Oculus Rift DK1*, while the interaction is gesture based and uses a gaze driven pointer. Schreiber et al. [19] proposed an approach for visualizing software modules in VR, using an electrical component metaphor. Modules are represented as blocks and the containing packages are stacked on top of each module. Merino et al. [20] and Vincur et al. [21] presented a VR visualization for object oriented software using a city metaphor. Both approaches rely on physical movement as their main navigational mechanism. In contrast, we use an explicit transformation of the visualization itself and are therefore independent of the available tracking space.

VI. Conclusions and Future Work

We presented our approach for exploring OSGi-based software systems in virtual reality. We used an island metaphor to emphasize the modular aspects of OSGi and implemented an interaction technique to preserve user comfort, while inspecting large software systems. In future work, we would like to extend the functionality to other module based architectures (such as Java 9), as well as determine the practicability of our approach in aiding software comprehension tasks. To support a larger number of software projects, we are replacing the custom software analysis tool in our pipeline with *jQAssistant*[3]. Additionally, it would be very interesting to explore our visualization technique in an Augmented Reality medium.

References

[1] R. Mili and R. Steiner, "Software engineering - introduction," in *Revised Lectures on Software Visualization, International Seminar*. London, UK, UK: Springer-Verlag, 2002, pp. 129–137.

[2] C. Donalek, S. G. Djorgovski, A. Cioc, A. Wang, J. Zhang, E. Lawler, S. Yeh, A. Mahabal, M. Graham, A. Drake *et al.*, "Immersive and collaborative data visualization using virtual reality platforms," in *Big Data (Big Data), 2014 IEEE International Conference on*, 2014, pp. 609–614.

[3] C. Ware and G. Franck, "Evaluating stereo and motion cues for visualizing information nets in three dimensions," *ACM Trans. Graph.*, vol. 15, no. 2, pp. 121–140, Apr. 1996.

[4] J. Jerald, *The VR Book: Human-Centered Design for Virtual Reality*. New York, NY, USA: Association for Computing Machinery and Morgan & Claypool, 2016.

[5] M. Misiak, D. Seider, S. Zur, A. Fuhrmann, and A. Schreiber, "Immersive exploration of OSGi-based software systems in virtual reality," in *Proceedings of the 25th IEEE Virtual Reality (VR) conference*. IEEE, 2018.

[6] W. Tobler, "Experiments in migration mapping by computer," *The American Cartographer*, vol. 14, pp. 155–163, Apr. 1987.

[7] J. J. LaViola Jr, "A discussion of cybersickness in virtual environments," *ACM SIGCHI Bulletin*, vol. 32, no. 1, pp. 47–56, 2000.

[8] H. B.-L. Duh, D. E. Parker, and T. A. Furness, "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*. ACM, 2001, pp. 85–89.

[9] J. Prothero, M. H Draper, T. Furness, D. Parker, and M. J Wells, "The use of an independent visual background to reduce simulator side-effects," *Aviation, space, and environmental medicine*, vol. 70, pp. 277–83, 1999.

[10] 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)*, March 2012, pp. 117–124.

[11] D. Seider, A. Schreiber, T. Marquardt, and M. Brüggemann, "Visualizing modules and dependencies of OSGi-based applications," in *2016 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE, 2016, pp. 96–100.

[12] M. Yang and R. 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*, ser. VINCI '15. ACM, 2015, pp. 137–142.

[13] J. R. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator," in *Applied Computational Geometry: Towards Geometric Engineering*, ser. Lecture Notes in Computer Science, M. C. Lin and D. Manocha, Eds. Springer-Verlag, May 1996, vol. 1148, pp. 203–222.

[14] P. Eades, "A heuristic for graph drawing," *Congressus numerantium*, vol. 42, pp. 149–160, 1984.

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

[16] 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)*, July 2007, pp. 217–228.

[17] A. Kuhn, P. Loretan, and O. Nierstrasz, "Consistent layout for thematic software maps," in *15th Working Conference on Reverse Engineering, 2008. WCRE'08.* IEEE, 2008, pp. 209–218.

[18] F. Fittkau, A. Krause, and W. Hasselbring, "Exploring software cities in virtual reality," in *2015 IEEE 3rd Working Conference on Software Visualization (VISSOFT)*, Sept 2015, pp. 130–134.

[19] A. Schreiber and M. Brüggemann, "Interactive visualization of software components with virtual reality headsets," in *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 2017.

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

[21] J. Vincur, P. Navrat, and I. Polášek, "VR city: Software analysis in virtual reality environment," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 2017.

[3] https://jqassistant.org/