

Metaphors for Software Visualisation

software (definition)	software as an animal	software as a food	software as a person	favourite music	hated music	role
Automatisierung von Arbeitsschritten zur Erleichterung und Effizienzsteigerung	 Albatros - alt und schnell	Knäckebrot	alt, sieht doof aus, trotzdem erfolgreich	Klavier	Techno	Entwickler
Die Spitzhacke des modernen Mannes	Eierlegendewollmilchsau	Nudelsalat	Superman	Successful blink	Console Piep	Entwickler
Software soll eine Lösung zu einer beschriebenen Aufgabe reproduzierbar bereitstellen	 Kraken Eine haarlose boose Kaize, die nicht macht was man sagt und alles kaputt macht wenn man mal nicht hinguckt	 da die Thematik recht trocken ist	-	House, Electronic	Hip-Hop, Death Metal	Tester
Automatisierung von Prozessen bezogen auf Daten	Krokodil - an Land langsam, im Wasser schnell Ein Elefant, da es schwierig ist während der Entwicklung die Richtung zu ändern.	 Nuss - innen weich, aussen hart	-	Electro, Jazz, Funk	Punk, Heavy Metal	Entwickler
Ein Produkt, welches mit Hilfe eines Computers eine abstrakte Idee umsetzt	 Papagei, weil gemachte Fehler von Checkpoint von Checkpoint weitergegeben und immer wiederholt werden.	McDonalds Burger - sieht gut aus, schmeckt nicht.	-	 Rock, alles was mir gefällt	Schlager, Ballemann-Hits	Entwickler
Hardware + Menschen miteinander verknüpfen	Chameleon, weil jeder Anwender und Anforderer etwas anderes da	Currywurst ohne Curry - halt mit Fehler	Roadrunner	Tonbandgerät	Elektronisches Produktmanagement	Tester
Alles, was auf einem Computer läuft und nicht Hardware ist	 Spider	Pizza, mit jeder Menge Belag, aber nicht ausgewogen Pile of wheat - material is storage, you can make a lot of things from	Die gute Märchenfee, weil jeder sich was anderes wünscht	Blues, Swing, Soul, Rock	Heavy Metal	Product Owner, Anforderungsmanagement
Liefert für eine fest definierte Eingabe eine fest definierte Ausgabe (möglichst fehlerfrei und performant)	Etwas großes was alles kann	Stinkender Käse	-	folk, pop	classical	Entwickler, Application manager
				Punk Rock, Metal	Folk und Pop	Entwickler

Master Thesis Documentation
by Lucia Mendelova

University of the Arts Bremen
Master of Arts in Digital Media (MA)
WS 2016/2017

Supervisors:
Prof. Andrea Sick, Prof. Dennis Paul,
Prof. Gabriel Zachmann

What is software?

“What you
don’t see
is what
you get.”

(somebody at Reddit 2015)

Abstract	9	
Preface	10	
1.1. Defining the Scope of Work	11	
1.2. Introduction	12	
1. 3. Related Work: 3D metaphors for Software visualisation	14	
2.1 Readme / Data	18	
2.2 Development // Chronicles of Failures or How I Worked with Data	19	
2.3. Methodology: VRID Approach	21	
3.1. Giant Dwarfs and Other Soft Objects	24	
3.2. VRID Model for the Metaphor “Giant Dwarfs and Other Soft Objects“	26	
3.3. The Little Big Helpers or What the Giant Dwarfs Could Be Good For	28	
3.4. Soft World	30	
3.5. VRID Model for the Metaphor “Soft World“	32	
Discussion // All I Wanted	34	
Epilogue	37	
Bibliography	38	

Acknowledgements

My gratitude goes first to my supervisors, Prof. Andrea Sick for patient listening and translating my thoughts into sort of understandable line of arguments, Prof. Dennis Paul for his open support for whatever crazy idea I came up with and Prof. Gabriel Zachmann for giving me a chance without knowing what to expect. I am especially indebted to HEC GmbH, the bravest traditional software development company, to support me during my studies even when I kept repeating that I would not produce anything useful. Heiko Mueller, Juergen Schaefer and my colleagues at HEC who got my back, answered my questions and helped where they could.

I profited greatly from support, help, fun and discussion with my classmates and friends, from whom I have learned almost the most during my studies. To mention just those without whom this thesis would not be possible: I am very grateful to Lukas Seiler for help with shader scripting, Lennart Jäger for a patience by blocking DK2 for couple months, Kristina Karabova for prompt proofreading and many other friends and family for their support and inspiration.


Most importantly, I thank my life-companion Ufo, for whom all hardware must be as silent as it gets and without whose patience and support I would not be able to finish my very last studies. Thank you for your humour when I lost mine (I do not remember if I ever have one ha ha, see?).

Abstract

Software visualisation makes the intangible tangible. It shows abstract and complex structures of code translated to graphical representation of entities and their relations. To do so, software visualisation uses metaphors. Similar to figures of speech are visual metaphors like figures of digital interface. The former support through patterns of speech patterns for thought, the latter through visual patterns supports structure attention and multitude of cognitive processes (see Tinell 2015). Not that there would be a single magical picture or a metaphor of a system which would explain every nuance of the software development, but through generating interactive and narrative images, we can talk about systems. This is important not only for specialized software developers, but for the growing amount of different social groups. Current situation is significantly marked by rising production of high-complex software which in turn creates and shapes a constitutive part of our lives. To gain understanding of software development with help of experimental visual metaphors and tools could be more than useful. The main research question of this thesis is therefore: what kinds of metaphors and interactions can be used for visualising software in virtual environments? The goal is to deliver experimental software visualisation models (based on different metaphors) in VR which will help discuss software within different users.

Preface

“Software is deeply woven into contemporary life - economically, culturally, creatively, politically - in manners both obvious and nearly invisible. Yet, while much is written about how software is used, and the activities that it supports and shapes, thinking about software itself has remained largely technical for much of its history. Increasingly, however, artists, scientists, engineers, hackers, designers, and scholars in the humanities and social sciences are finding that for the questions they face, and the things they need to build, an expanded understanding of software is necessary.”
(Bratton 2015, p.12)

Coming to the field of digital media with a background in the philosophical theory of knowledge and design, with the topic for my Master thesis I fulfil all the cliché necessary to fit in the Bratton’s quotation. I am not sure if this is why Prof. Zachmann asked me if I can imagine to work on the topic of “Immersive Metaphors for Software Visualisation” - either way, I answered “yes”. Working with the weird, hyped, fascinating and highly commercialized technology of virtual reality which I got to during my studies almost accidentally and could not get over it (reassembling or a stock photo  has many philosophical problems and is more fun to play with than programming because of the god’s feeling you get walking through the worlds you have created), I decided to look at the software development and its tools from within. To taste the corporate life completely, I also accepted a scholarship of one of the biggest local software development companies and started my life behind two big screens, well-formed business chair, grey carpets and great coffee. What follows is also a partial documentation of my experience in VR and software development.



1.1. Defining the Scope of Work

The official task of this thesis was to find metaphors for software visualisation in VR for software developers. In the process of getting familiar with the topic I was searching for metaphors which would help me understand what software development actually consists of. This way, the final project turned out to be more of a software visualisation for beginners than for professionals. It introduces software in a simplified and playable way, so that different users coming to the code later in its life can understand, memorize and discuss the software structure.

In summary, deliverables comprise the literature review on (1.) the contextual information about 3D software visualisation. Such research is used to establish the rationale for (2.) the creation of a model of visual metaphors in VR in order to measure the advantages and limitations of the medium when used for the software visualisation of complex systems (testing will happen between submitting and presenting the thesis).

The structure of this report is following: Chapter one introduces the field of research and related work; defines the key terms in this research area (software visualisation, metaphors, current state of 3D and VR/AR software visualisation); Chapter two describes used dataset, scratch out the working process and finally introduces the methodology for designing VR interfaces. This methodology is then applied to created metaphors which are introduced and described in Chapter three. The final chapter shows the discussion on the further development possibilities; at the end comes a short epilogue.

1.2. Introduction

“Software is intangible, having no physical shape or size.”

Thomas Ball, Stephen Eik 1996

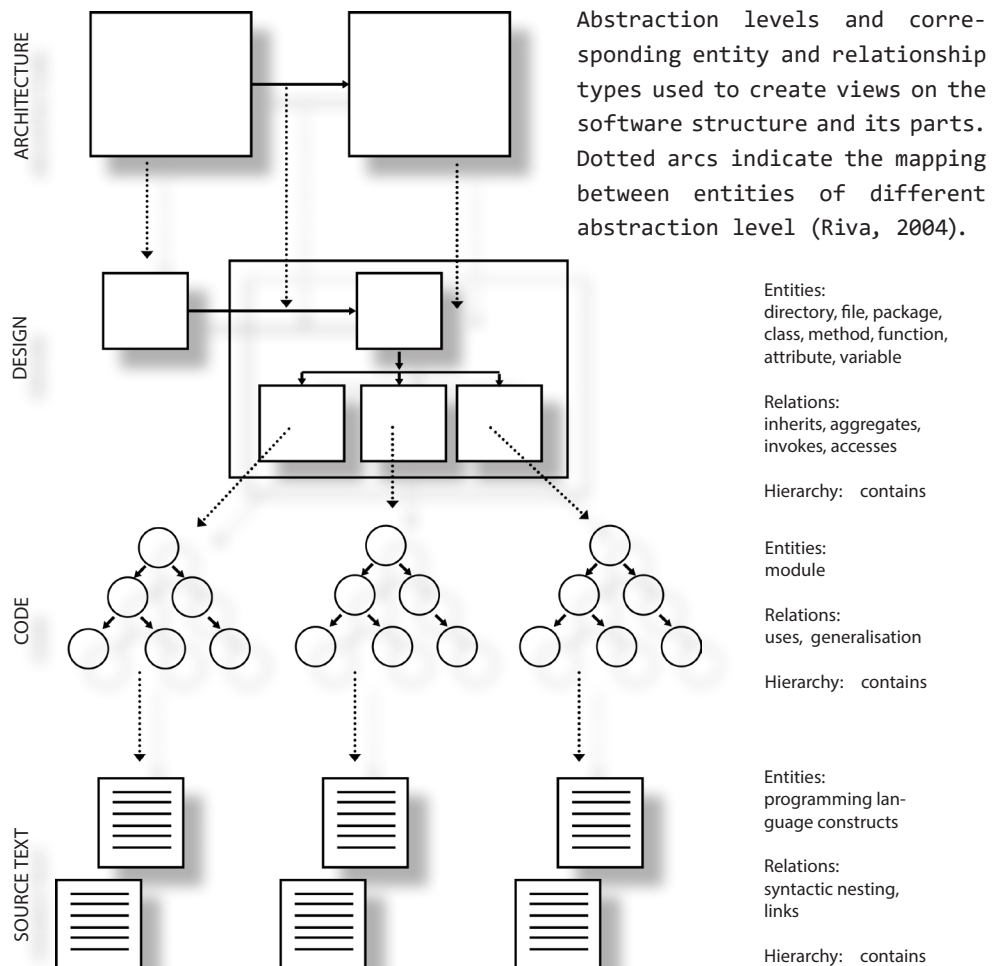
Do not think of a pink elephant. Do not think of Visual metaphors do not provide so much freedom for imagination, but often catch more attention, are easy to remember and serve as a common reference. Ball and Eik made a good point thirty years ago and so are today’s software usually pink elephants, big bowls of mud, octopuses as well as schemes of connected triangles and rectangles in UML style. Even people working everyday within the realm of software development report how hard it is sometimes to get familiar with a software they have not seen before. Imagine you are standing in front of a running giant mechanism built out of 60k small parts which somebody else created nine years ago and many others have been working on since then, and now you just have to fix “this small bug”. Imagine you cannot stand in front of that mechanism, because there are no parts to see - all that you are facing are lines of code on your screens. As time passes by, you learn to see all the objects and their functions in the code quite quickly. However, this is how other integral actors (not necessarily developers), i.e. important parts of the software development processes, can feel about it - and you need to talk together about the bug to fix it.



If the main challenge besides the complexity is the intangibility of software (often referred as the “Ball’s Dilemma”), the way how to visualise it, is the use of metaphors. “The essence of metaphor is understanding and experiencing one kind of thing in terms of another.” (Lakoff & Johnson 1980). Using one symbolical representation to refer to something else is what computer science terminology is also strongly based on. We use windows, documents, folders, packages without even registering they do not refer to the corresponding physical objects but to the user interface visual representation of the computer functionality. A “visual metaphor” is an image based on analogy to the rhetorical figure of metaphor. (Yet) instead of a lexical term, it uses a particular graphical representation to transfer some abstract information to a new context. Graphical representation can refer to the illustration of abstract entities or to models in a spatio-temporal three dimensional realm. This is what happens when we start to think about metaphors for “post-wimp” interfaces - e.g. for virtual reality. We move into a computer-generated three dimensional space and there comes the very justified question: why to do so, actually?

Answering in a very simplified manner: firstly, human perception has a strong spatio-temporal affinity - so to say, we are optimised for seeing the world and patterns in three dimensions. Since VR operates by simulating 3D environment it has a huge immersive potential and as such “leads to a demonstrably better perception of a data-scape geometry, more intuitive data understanding, and a better retention of the perceived relationships in the data.” (Davidoff&Norris 2014, IEEE) That is to say, if we simulate conditions which are familiar to the interaction and orientation in our embodied environments, we could gain the understanding of abstract structures. Secondly,

a decision to play with virtual reality and simulation was inspired by the concept of habitability (Gabriel 1996), which within software development refers to the feeling of being inside of the software system, looking at the code and feeling comfortable because you are confident about it (it's easy to change it, for you know what is where and how it works, you are at "home" there). According to Gabriel, software needs to be habitable, because it always has to change. Requirements change with the context, parts of the design are proven wrong by experience and rather than about technical issues, the unpredictable events are often more about people and society (ibid, p. 12). Software varies and develops under the demands of different personal, institutional and market pressure. Comfortable understanding of the software structure which reassembles the way how we learn and acquire knowledge through our embodied actions in the world can bring the necessary flexibility and comfort in changing the systems. And lastly, it is really fun.



1.3. Related Work

Software Visualisation & Its 3D Metaphors

According to Diehl a narrow definition of software visualisation is “a visualisation of algorithms and programs” and the wide one is: “a visualisation of artifacts related to software and its development process” (Diehl 2007). Software visualisation approaches in general can be categorized into three distinctive groups: static visualisation, dynamic visualisation, and evolution visualisation.

Static visualisation deals with static parts and relations of the system which can be visualized without running the program, like source-code, data structures, static call graphs, and system modules.

Dynamic visualisation shows the behaviour of a programme for a given input and has successfully been used for algorithm animation, architecture visualisation augmented with run-time information, and visual debugging and testing.

Evolution visualisation depicts how software changes over the time usually based on metrics such as code age, number of bug fixes, structural change, and evolutionary coupling (ibid, p. 4-7).

Current software visualisation is dominated by two dimensional geometrics metaphors such as boxes, circles, arrows and graphs (the well-known and also very usable UML style). What follows is a brief overview of metaphors used within three dimensional software visualisation.

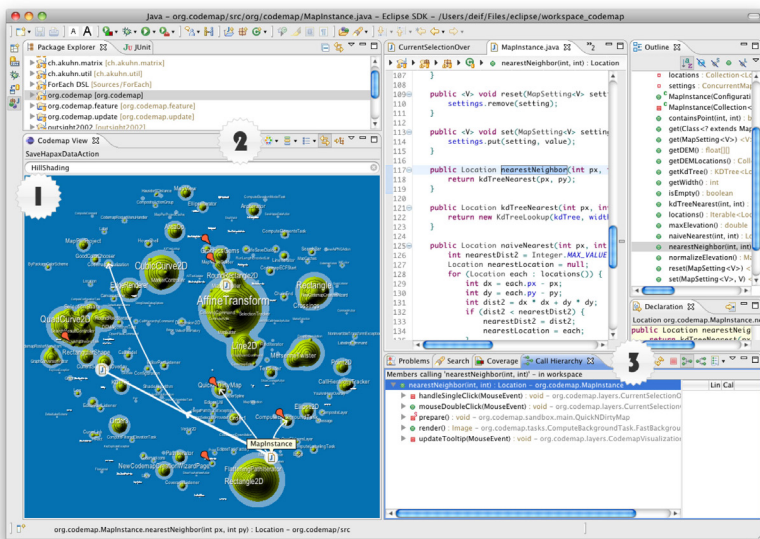
“Map Metaphor” (img. 1.1) – maps are widely used as an orientation tool to display spatial information. Humans are used to interpret such data in order to navigate in the real world. This direction is also taken by “Landscape Metaphor” which can be seen as a category of maps but goes further to the concrete “real-world” story line. Here within 3D space, real-world objects such as rivers, hills or houses and buildings represent abstract code entities. These are mostly used as a static visualisation, with an option to evolution visualisation or even some dynamic aspects of software (see Teyseyre, Campo 2009).

Under the category landscape metaphor are often subsumed also the “Code Cities”, which are one of the most used 3D software metaphors. Originally created by Richard Wetzel (2009 img. 1.2), they also exist today as an open source Eclipse and SonarQube plugin (1.3, 1.4) and also in augmented reality version (SkycrapAR at img. 1.5) or as a start-up idea (Seerene and its “CapaCity” version of code cities – img.1.6). Some of the city metaphors go pretty far in their level of naturalism (e.g.

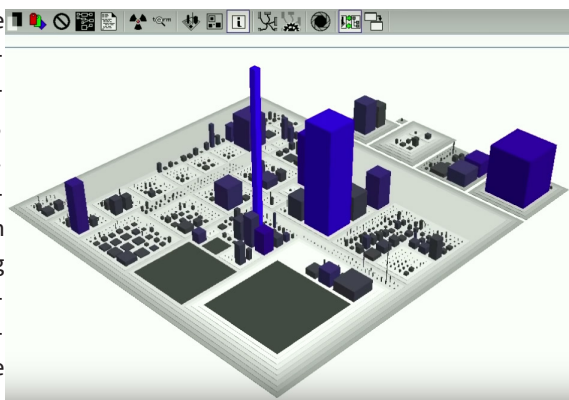
Thomas Panas – img. 1.7). “Galaxy or solar system metaphor” (img. 2) is one of the last often used metaphors in such a narrative approach to software visualisation. Galaxy is usually produced using algorithm for force directed graph layout or clustering algorithm to place nodes in three dimensions. The relation between the nodes is usually presented by proximity (two related nodes are placed close together).

1.1 Image of a “Code-map” (inspired by an older software cartography tool called “Cartographer”) and Eclipse plug-in for Java (based on EclEmma). Domain mapping: the hills represent classes, their position on the map is determined by their vocabulary,

the size of the hill refers to the lines of code metrics of corresponding classes. Colours could represent different metrics (see Enri&Kuhn 2010, p.73)



1.2. Domain Mapping in the case of Code Cities (can differ within applications): packages are mapped to districts, classes represent buildings. A polymetric view principle applied to the classes maps on the base size of the building metrics representing the number of attributes and the number of methods represents the height of the class/building.



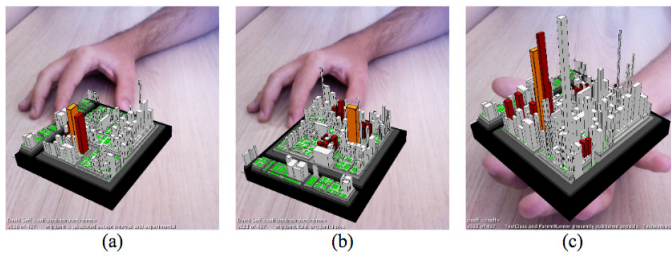
CODE-MAP: <http://scg.unibe.ch/archive/masters/Erni10a.pdf>
 CODE CITIES: <http://www.inf.usi.ch/faculty/lanza/Downloads/Wett2010b.pdf>

1.3 - 1.4. Eclipse and SonarQube code cities plugins (both open source,

in the case of Eclipse just for non-commercial usage). Mapping: Folders or packages are shown as districts, files as buildings. The building footprint, height and colour are dependent on arbitrary sonar or eclipse metrics.



1.5. SKYCRAPAR

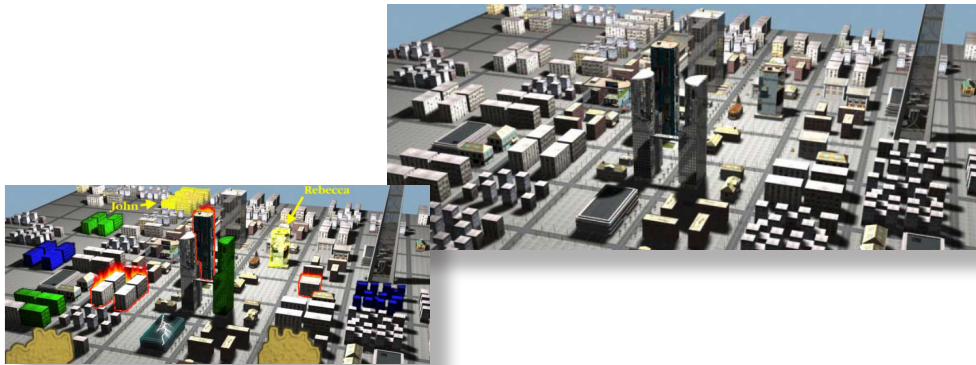


An Augmented Reality Visualisation that employs the city metaphor to represent evolving software

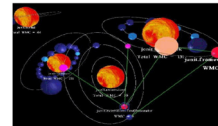
1.6. SEERENE corporate style software-visualisation. Big words, flat graphics, user-friendly interface and an idea which has been tested in many open-source plugins. "CodeCity metaphor" this time as a "CapaCity" startup package.



ECLIPSE: <https://marketplace.eclipse.org/content/codacity>
 SkycrapAR: <http://reuse.cos.ufrj.br/wbvs2012/papers/wbvs03.pdf>
 SEERENE: https://www.researchgate.net/publication/4026411_A_3D_metaphor_for_soft

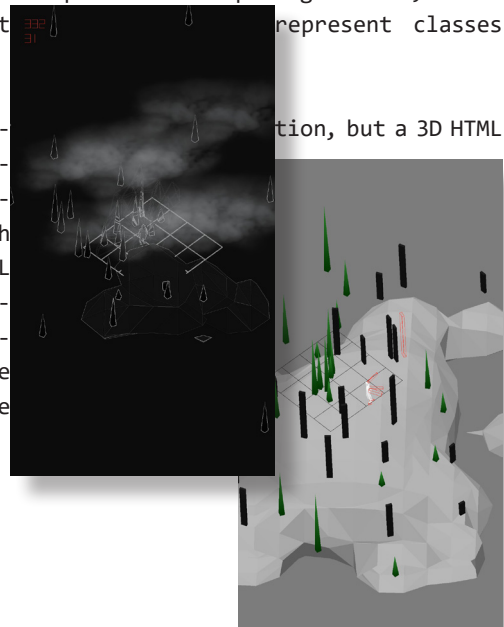


1.7. A very narrative code city version by Thomas Panas showing hot execution spots and work distribution.



1.8. A Solar System Metaphor for 3D Visualisation of Object Oriented Software Metrics. The main code entities (i.e., packages and classes) are represented as planets encoding software metrics in the planets' size and colour. This visualisation represents the software as a virtual galaxy consisting of many solar systems. Each solar system represents a package. The central star of each solar system represents the package itself, while the planets in the orbit around it represent classes within the package.

1.9. Not really software visualization, but a 3D HTML interpretation. LITTLEBROWSER is an experimental web browser and game engine hybrid. Made with Processing 3 and using the ProHTML library by Tex, an autonomous crawler navigates web pages and translate their main elements into game objects: links become gates, divs are clouds, images turn to trees...



GALAXY: <http://dl.acm.org/citation.cfm?id=1082108&d1=ACM&col>
 quotation source: <https://arxiv.org/ftp/arxiv/papers/1601/1601.07742.pdf>
 PANAS: https://www.researchgate.net/publication/4026411_A_3D_metaphor_for_soft
 LITTLEBIGBROWSER: https://frm.fm/a/n_o_r_m_a_l_s/l_i_t_t_l_e_b_r_o_w_s_e_r

2.1. Readme: Data

The current version of modelled environments is based on the data about BKB Distributions Portal, a logistic tool for Carriage Company B+K Group from Krefeld (Germany) for shipping their consignments to Aldi stores. I have always been drawn by political themes and I basically live in Aldi, so this fits well. This software was developed in Java at HEC Software Development between 2015 - 2016 as an extension of B+K Group native distribution software. General data about the software (out of Snapshot 1.2.3. from 16th July 2016 on which most of the static data visualisation are based): 23k lines of code, 325 files and 363 Java classes, 80,6% coverage through 139 unit tests, and all this in layered architecture (data-bank, than business logic and UI on the top as well as a small Web app).



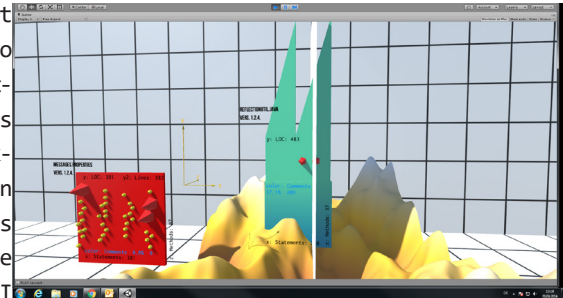
Data Metrics about the system comes from SonarQube, an open source quality management platform. Data were delivered as a Csv export of ten static snapshots of the software evolution. Each of the snapshots delivered the following data about each file: LAYERDIRECTORY, FILENAME, DATEOFLASTCOMMIT, LINES, LINESOFCODE, ~~LINESOFCODEPERLANGUAGE~~, FILES, ISSUES, CRITICALISSUES, MAJORISSUES, MINORISSUES, OPENISSUES, CODESMELLS, BUGS, TECHNICALDEBT, MAINTAINABILITYRATING, ~~SCALEDEVELOPMENTCOST~~, TECHNICALDEBTRATIO, STATEMENTS, COMMENTLINES, COMMENTLINESPROCENT, RELIABILITYREMIATIONEFFORT, RELIABILITYRATING, SECURITYRATING, FUNCTIONS, CLASSES, ~~CLASSDISTRIBUTIONCOMPLEXITY~~, ~~COMPLEXITYFUNCTION~~, ~~COMPLEXITYCLASS~~, ~~COMPLEXITYFILE~~, COVERAGE, ~~OVERALLCOVERAGEHITSBYLINE~~, ~~COVERAGEHITSBYLINE~~, ~~ITCOVERAGEHITSBYLINE~~, ~~COVERAGEONNEWCODE~~, ~~COVEREDCONDITIONSBYLINE~~. Strikethroughs were not delivered (but are optional metrics in SonarQube). There were no data about dependencies, even if this would be one of the most interesting aspects to visualize (changes of the system with respect to class or method dependencies).

Since dataset already forms possible ways of visualisation, the developed metaphors concern mainly the following: static visualisation of the size of classes (LOC, statements and methods), test coverage and code smells. The last mentioned, code smells, are automatically detected by SonarQube, as defined by its documentation as “A maintainability-related issue in the code. Leaving it as-is means that at best maintainers will have a harder time than they should making changes to the code. At worst, they’ll be so confused by the state of the code that they’ll introduce additional errors as they make changes.” (<http://docs.sonarqube.org/display/SONAR/Concepts>). According to Grandma Beck and Martin Fowler “a code smell is a surface indication that usually corresponds to a deeper problem in the system” so it is obvious that smell definition is context-dependent and it is disputable in many cases. Automatically detected smells in SonarQube are distinguished to “blockers”, “major” and “minor” issues most of which are small violations of good practices (unused private fields, missing methods in constructor, tab characters etc.). The broader understanding of code smells outside the realm of predefined metrics is part of outcome of the thesis. One of the few things the visualisation for beginners has shown is the potential of searching and identify code patterns and smells as described by Fowler (SOURCE).

2.2. Development // Chronicles of Failures or How I Worked With Data



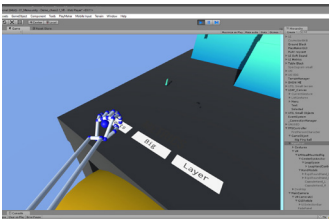
The Metaphors (as models of VR environment) are in their current version developed in Unity 5.5.0f3 and running on VIVE SteamVR platform which proved to be the most user-friendly and stable environment to work with and to test. The process of finding a solution could also be well described as a small chronicle of failures. First version of the test environment - so to say the first level and the first metaphor (working title "Dwarfs") - has been developed in Unity 5.3.5 as a desktop environment (all the interaction with a keyboard and mouse). All models were created manually since there was no database yet. I basically checked against the local server SonarQube statistics and copy/paste values I needed.



```
protected void FixedUpdate ()
{
    base.FixedUpdate ();
    updateBall ();
}

protected override bool checkConditionGesture ()
{
    Hand hand = GetCurrentHand ();
    if (hand != null) {
        if (isGrabHand (hand) && !isHoldingBall)
            return true;
    }
}
```

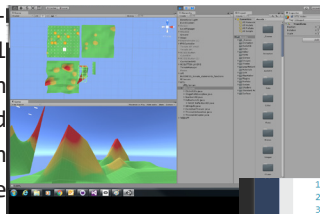
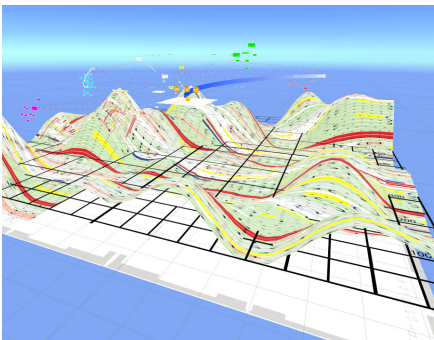
Although all the interaction worked properly on the desktop, the step to VR environment was painful. For working with the second developer kit from Oculus, it was necessary to downgrade Unity because the only version working with DK2 was 5.3.4p1. What did not brake, needed to be rewritten - basically all the interaction code since there is no mouse in the VR world-space. Even though there is a way around, I did not want to base all the interaction on the keyboard and mouse, VR would not make any sense in that case. DK2 also does not have controllers, so I needed a couple of days to get Orion update for LeapMotion SDK working with the old SDK for Oculus.



However great the newest upgrade for a small infrared controller for hand interaction in real time is (and it is!), it has worked just partly and, unfortunately, was not stable at all. No tests are possible if most of the time, the same gesture barely managed to work two times in a row. In any case, this was the setup which was used at the beginning of modelling the second metaphor (working title "Soft World" - yes, I started to watch the "West World" series at that time). I also got statistics in csv format for the current snapshot to play with (although there are possibilities to work with a broad range of data input within unity - json, xml, or even real-time data transfer between applications through OSClisten, my technical skills felt well



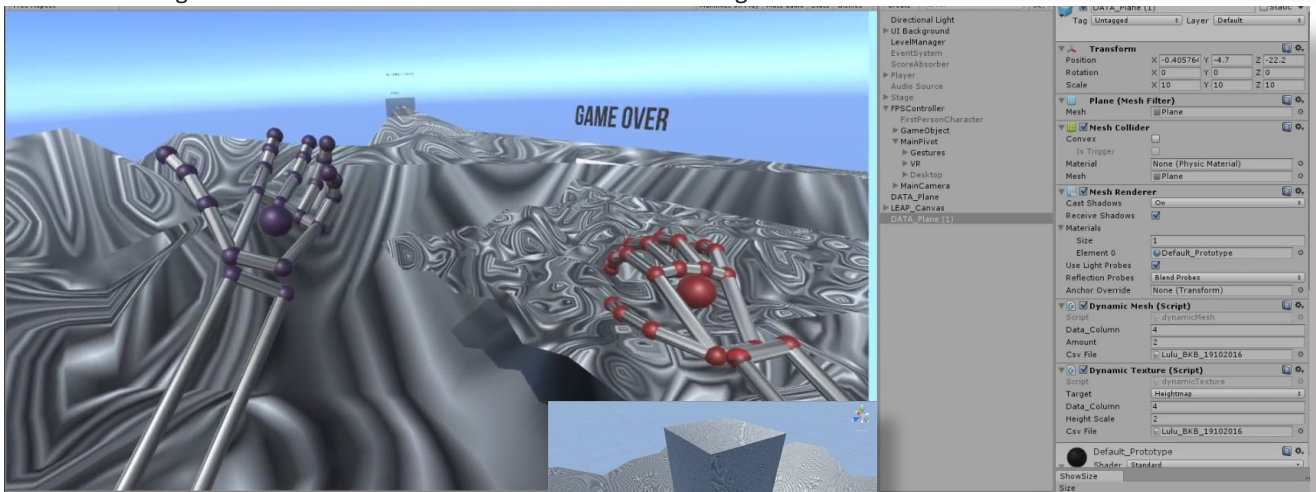
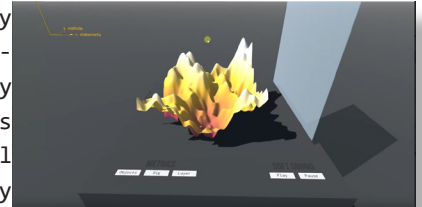
csv). Csv data tables were then translated to bitmaps (picked and presented in gray-scale) and those bitmaps again translated to terrains (I used heat-map code generating terrain, a free JavaScript snippet which does the translation for you after small cosmetic changes). In this way generated data



```
1 using UnityEngine;
2 using System.Collections;
3
4 public class ActivatingObjects : MonoBehaviour
5 {
6
7     public GameObject terrain1;
8     public GameObject terrain2;
9
10
11     void Start()
12     {
13         terrain1.SetActive(false);
14         terrain2.SetActive(false);
15     }
16
17     public void ActivatingTerrain1()
18     {
19         terrain1.SetActive(true);
20         // terrain2.SetActive(false);
21     }
22
23     public void ActivatingTerrain2()
24     {
25         terrain2.SetActive(true);
26         // terrain1.SetActive(false);
27     }
28
29 }
```

terrains look great, but are not very effective (too heavy for performance, too inflexible for interaction).

At that time, I started trying to create data objects procedurally, generate them directly out of the code with help of shaders. You could use one data set to generate a heat-map mesh (vertex shader, in Unity called "parallax map"), i.e. the geometry of objects and another data set for creating textures. The texture could not only bring colours generated out of code, but small 3D effects as well. The so called bump-map effect (in Unity called "normal map") creates optical illusion of 3rd dimension through the way how light reflects and diffracts on surfaces. Using shaders and



procedural meshes would actually be a proper way how to create virtual worlds out of huge data sets (even in real time). I imagine this like something between Git Extension and No Man's Sky for VR, multi-player, of course. Ha. In reality, I needed to ask for help by writing a shader (Unity uses mono development, a mono version of c# net, but shaders are written in a shading language, which I would not get running at all without the help of Lukas Seiler). On the 3rd level, which is not part of this report, but which you can test at the presentation, you can play with the surface and its texture created procedurally out of a data set.

2.3. Methodology

VRID Approach

VRID stands for Virtual Reality Interface Design and it refers to the methodology for developing VR interfaces suggested by (Tanriverdi&Jacob 2001). VR interfaces are more complex than the traditional one, there is a huge variety of possibilities and responsibilities and only a small range of user interface rules which are “hard-ened” as the “intuitive” standard. Most of the UI/UX best practices are still being developed on the go, depending on the input from the current boom of the consumer versions of VR devices. In this context, it can be quite helpful to check some of the older research articles (from the era of the first VR hype) to learn from what was once a topic of interest. (with the benefits of today’s GPU/CPU power).

VR interface design methodology identifies two main phases within VR interface architecture and in the corresponding software visualisation architecture.

1, High-Level Design Phase, whose main tasks are to:

a., Identify data elements

b., Identify objects

c., Formulate general specification in graphics, behaviours, interactions, internal and external communications for each object

2, Low-Level Design Phase

Detailed specification in graphics, behaviours, interactions, internal and external communications for each object. These factors together create the core of the approach, so called “VRID Model”.

The “VRID Model” is a set of key constructs that designers should think about while designing VR experiences.

1, The graphics component – specify graphical representation of interface objects. Or for the all objects /entities according to data input and functionality of the software visualisation which needs to have their representation in VR.

2, The behaviour component – The first task is to break the complex behaviours down into simple physical and/or magical behaviours. (Grabbing an object is a physical behaviour, highlighting when grabbed is a magical one.) Breaking the complex behaviours down into simple ones can help build a library of behaviours which can be reused (grabbing can be used on a different object with or without highlighting). Distinction between the physical and magical behaviours can also help communicate the design decision.

3, The composite behaviours – describing behaviours which consist of a series of simple physical and magical behaviours (can be seen as extension of the previous point which significantly helps with coding).

4, The interaction component – specifies the origin of inputs for interaction and their effects on the objects in VR. A conceptual distinction between interaction and their implications for the object behaviours (for grabbing, I need one script for an object and another one for the controller). When behaviours are de-coupled it also increases the reusability (highlighting an object can be called on trigger by pressing enter, whereby trigger can be a hand controller, another object or even an eye-ray-caster – it is important to specify and check necessary condition for each case).

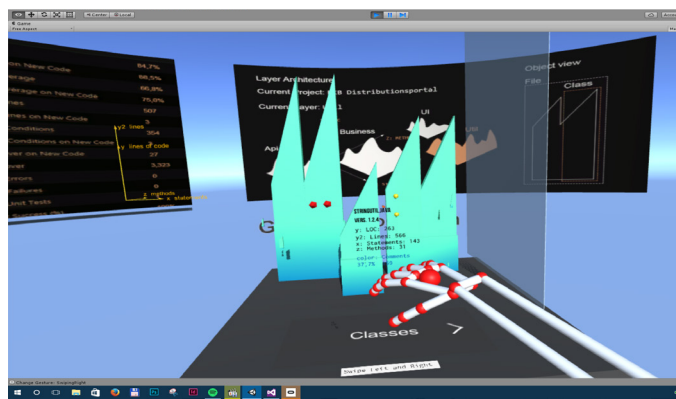
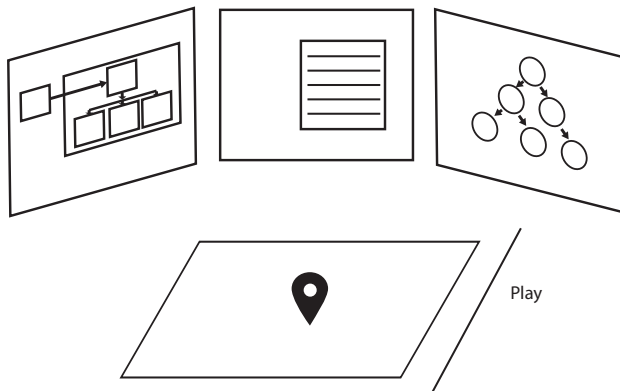
5, The mediator component – ensures loose coupling between the components, helps with controlling, communication and avoiding conflicts in the object behaviours. In most of cases, this is already built in the way how Unity scripting references are organised.

6, The communication component – is for the external communications of the object with other objects, data elements, or other applications.

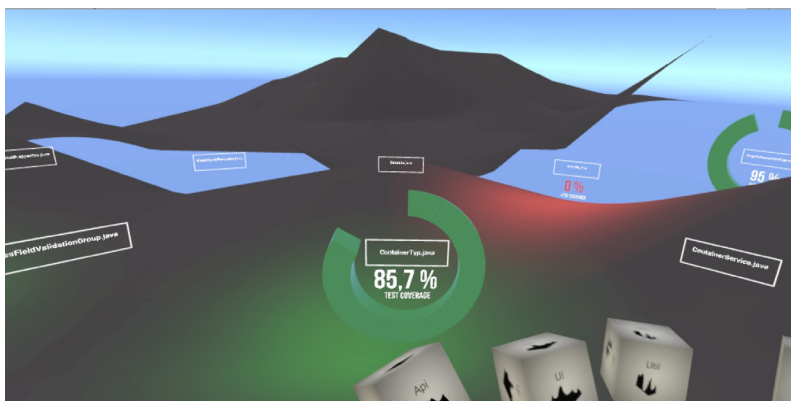
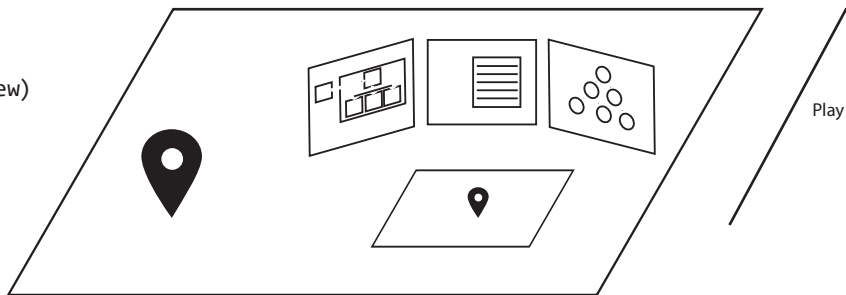
In order to decompose the design task into a smaller task, comprehensive planning is the “takeaway message” for designing VR experiences. However, it is hard to plan when you do not know what is possible and what you can built properly. What worked out for me was the “ping-pong style“. Plan something, decompose it and try to implement. By slipping up and searching for workarounds get other ideas about possible interaction, go back to VRID plan and adapt or extend it. Your skills and technology will shape your outcomes.

What follows is a short description of VR setup and a scenario for each of the metaphors (in this case they are built as separate levels between which user can move). Later on, a VRID model table list out data elements and interactions for each of them.

METAPHOR 1:
Giant Dwarfs
and Other
Soft Objects
(the micro view)



METAPHOR 2:
Soft World
(the macro view)



3.1. Giant Dwarfs and Other Soft Objects

Consider a virtual reality system developed for training software development and software architecture basics. It includes a virtual surface and virtual objects created from software statistics and a gesture interaction system. The user wears a head-mounted display, and uses two hand controllers in order to interact with the room-scale virtual environment. The HMD and controllers communicate 3D coordinates of the user's hands and head to the VR system. Coordinate data are used to interpret actions of the user. The 3D model of the software system is placed in front of the user, when she can interact with it.

Giant Dwarfs

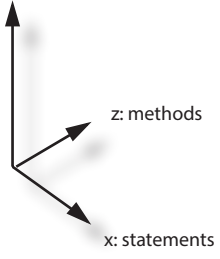
In front of the user, there is a table, above which floats a couple of green, flat shaped objects in different sizes. Some of them have red or yellow coloured diamond shapes on their surface. When the user points against the object, an information layer reveals that we are looking at the objects representing code files (filename, x, y, z values). The height of the objects (y) has two values, which creates sharp endings of otherwise rectangular shapes. It represents lines of code (y1) and lines (y2) which is more or less higher number because all the comments are counted in. The width (x value) maps the statements* used in Java file. The depth (z - value) represents methods**. The colour changes from green to blue gradient getting more intensive, as it is covering a base of the objects. The text layers tell us that the gradient colour represents the % amount of comments in the file. Pointing at the diamond-shaped objects reveals a label which shows information about "smells" (based on metrics from SonarQube). The text reveals when the "smel"1 was detected, by whom and in which code row it is to be found. It shows also the SonarQube icon used for labelling smells as minor or major (minor green, major red circle).

There are buttons on the table: Resize and Play. Pressing "resize" creates the same objects ten times bigger around the user, who can now walk around and examine also the small objects properly. Pressing "play" activates a Raycast which slowly goes through the objects from right to the left activating them. Each part of the objects play its melody.

* Definition of Statements according to SonarQube: "Number of statements as defined in the Java Language Specification but without block definitions. Statements counter gets incremented by one each time a following keyword is encountered: if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally. Statements counter is not incremented by a class, method, field, annotation definition, package declaration and import declaration." <http://docs.sonarqube.org/display/SONAR/Metrics+-+Statements>

** Methods are functions in SonarQube. Definition: "Methods represents number of functions. Depending on the language, a function is either a function or a method or a paragraph." <http://docs.sonarqube.org/display/SONAR/Metric+Definitions>

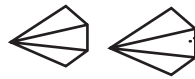
y1: lines
y2: lines of code



class 1

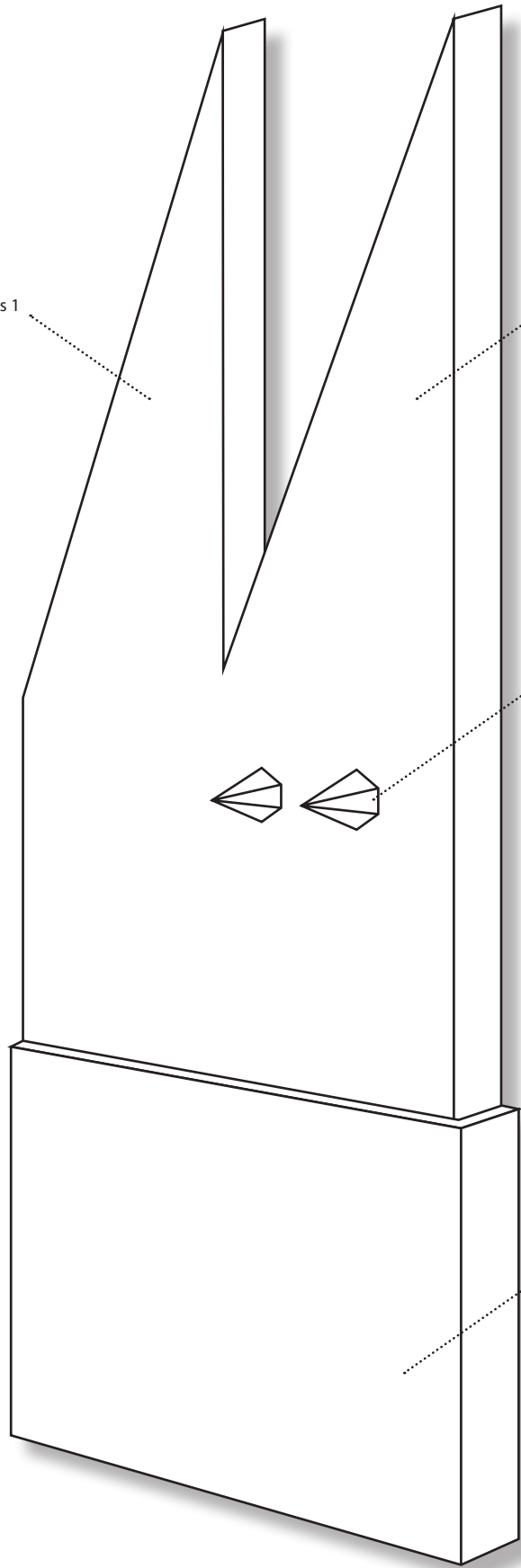
class 2

smells



Object:
one file with
two classes

comments



3.2. VR ID Model for the Metaphor

Data elements	Description	Data sources	Objects	Graphics
name; metrics or other data from sonarQube	GOAL: to identify software data inflows coming into the VR interface. INVOLVES: a, defining the data source; b, describing what the data represents; c, specifying the data set or other data parsing possibilities	GOAL: to identify data source coming into the VR interface. INVOLVES: The interface can receive data from three sources: user / physical devices / other VR systems /	GOAL: to identify objects that have well defined roles and identities in the interface. INVOLVES: a) identifying potential objects mentioned in the interface description; b) deciding on legitimate objects; describing aggregate objects and its parts; c) distinguishing between virtual and physical objects (the virtual ones are generated by computer, the physical ones are physical entities that interact with VR system – they may or may not require modelling).	GOAL: to model the virtual objects identified in previous steps; specify a description of graphical needs of virtual objects. INVOLVES: description of what kinds of graphical representations are needed for each object, and its parts, if any
File size and "smells" – issues on files (metrics from SonarQube).	Input: data from SonarCube about the file size, comment percentage and code smells. Visualizes the pre-defined issue reports out of SonarQube, but through the graphical representation of the file size also other "standard" smells (see Fowler) are identifiable. Metrics from onarQube size statistics; Size Metrics: File, number of classes, LOC, Lines, methods, statements (in JAVA: Number of statements as defined in the Java Language Specification but without block definitions. Statements counter gets incremented by one each time a following keyword is encountered: if, else, while, do, for, switch, break, continue, return, throw, synchronized, catch, finally. Statements counter is not incremented by a class, method, field, annotation definition, package declaration and import declaration; Methods are functions in SonarQube. Definition: "Methods represents number of functions. Depending on the language, a function is either a function or a method or a paragraph." http://docs.sonarqube.org/display/SONAR/Metric+Definitions Smells: issue amount, sing/icon, (label - since when (from whom), tag	The general source of data on the level of user and physical devices is HMD and controllers which give us the position of a user and the way she interacts with the environment. This will be specified in behaviours and interactions section. Focusing on data from SonarQube, in the current state we have mostly static visualisation where the data set is loaded once and not dynamically changed at runtime.	Potential objects for UI are: Data files as flat geometrical 3D shapes are virtual (computer-generated out of data) and aggregated (they have audio, collider, data layer showing textual information, potentially child objects – smells). Smells are also legitimate objects which are aggregate (have sounds, colliders, text-labels). "Play-raycast" is a 3D object, movable, has a collider and triggers the audio on other data objects. It has classical UI elements – buttons with functionality to activate objects. Physical objects are HMD (CameraRig) and hand controllers, which also have their virtual representation (not necessary to model, we are using prefabs in this case).	The data objects were modelled by hand, but ideally, they would be generated algorithmically out of data. The texture is also currently created manually, but could be generated data as well. The table is based on simple geometrical shapes, has no interaction, is created directly in Unity with a standard material. The Raycast is also a primary shape modelled in Unity with a transparent material. UI buttons are based on Unity GUI with own sprites and custom fonts. The small objects have box colliders, also created directly in unity with a custom sprite texture imported from Photoshop.

Behaviours	Interactions	Internal communication	External communication
<p>GOAL: to identify behaviours exhibited by objects; classify them into simple physical, simple magical, or composite behaviour categories; and to describe them in enough detail for designers to visualize the behaviours.</p> <p>INVOLVES: a) identifying the behaviours from the description; b) classifying the behaviours into simple and composite categories; c) classifying the simple behaviours into physical and magical behaviour categories; d) for composite behaviours, specifying the sequences in which simple behaviours are to be combined for producing the composite ones.</p>	<p>GOAL: to specify where inputs of interface objects come from and how they change object behaviours.</p> <p>INVOLVES: a) identifying interaction requests to objects; b) identifying the behavioural changes caused by these requests and which behavioural components will be notified about these changes.</p>	<p>GOAL: to specify control and coordination needs for internal communications among the components of objects in order to avoid potential conflicts in object behaviour</p> <p>INVOLVES: a) examining all communication requests and behavioural changes that are caused by these requests; b) identifying communication requests that may cause potential conflicts; c) deciding how to prioritize, sequence, hold or deny the communication requests to avoid potential conflicts.</p>	<p>GOAL: to specify control and coordination needs for external communications of the objects.</p> <p>INVOLVES: a) identifying communication inflows into the object, and their sources; b) communications outflows from the object, and their destinations; c) describing time and buffering semantics of external communications of the object.</p>
<p>Behaviours are: (1.) Activating the textual layer on each object (simple gesture, magical and physical – needs to listen for the hand controller pointing gesture); (2.) Showing/hiding the text label to smells (the same as previous) (3.) Play – activating and moving of the "Play-raycaster" object (composite, magical and physical – activates on UI button click provided by hand controller) (4.) Teleporting – is a simple magical as well as physical behaviour, based on the pointer of the left hand controller.</p>	<p>Showing Text information on object calls on the box colliders and needs a trigger (hand controller pointer which also has a rigid body). This collision notifies the "text-node" object, which will then be activated. "Playing" works similar, but uses the graphical user interface button element as a mediator. Hand controller collides with the button, on click activates the UI event system, which then calls the code on the ray-caster object. This is a simple c# script which performs the movement based on iTween library (an open source animation system based on a simple hash-table rule set of transition which are called within script when needed).</p>	<p>Potential conflicts are on "playing" function of the ray-cast object: originally, there should be also an option to pause and rewind (It would be interesting if we had much more objects and heard something which caught our attention and needed to find the source of the sound). In the current version, this option is not working due to improper storing of the last played position which causes conflicts with a new call.</p>	<p>Communication inflow are mostly the 3D coordinates of the user's head and hands in the room-scale VR, most of the technological requirements (buffer semantics, collision detection) are regulated by the provider of the HMD software and its development environment (VIVE, SteamVR and Unity). However it is important to program carefully, plan and test the user interaction so that there are no glitches. To paraphrase a VR filmmaker Eran Amir, the risk of creating bad VR experiences is not just that people would not like them or be bored, but they can get sick by motion sickness. And this is not to underestimate.</p>

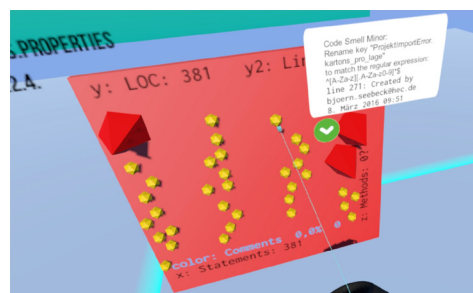
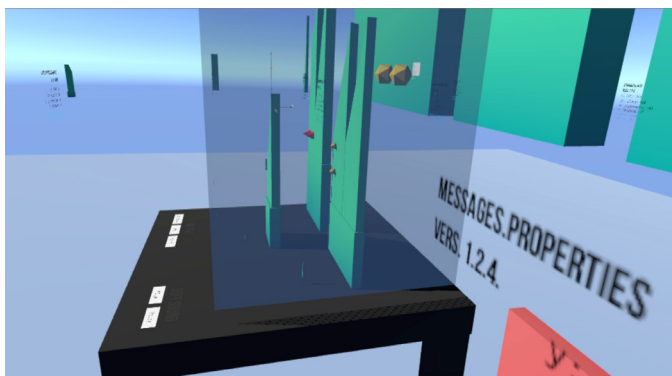
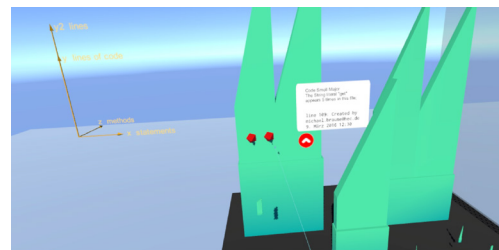
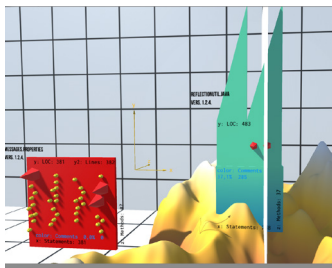
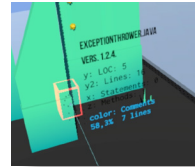
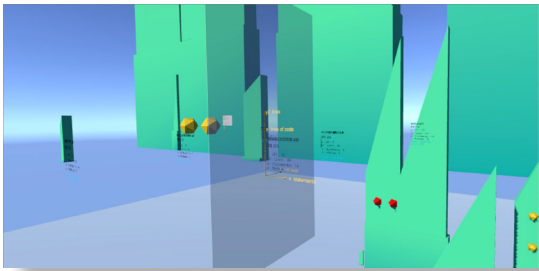
3.3. What the Giant Dwarfs Could Be Good For

// The Little Big Helpers

1, Large scale problem comprehension – surrounded by a hundred of giant dwarfs, we could start to recognize patterns in packages and classes. Visually quickly recognisable differences in the size of files can be traced even in our small example. Giant files meet small classes of Util.test – could be a trace of refactoring attempt – Unit-tests can help deal with huge classes; it is classical advice to write tests for each method before extracting it to ensure you don't break functionality. After asking for the purpose of small Util.test files, it has been found that they were actually misplaced. With the next snapshot, they were moved in a separate directory.

2, Displaying software defects or problems – we could even build a categorisation of some patterns. Take the two big files in our example, let us call them Giant Dwarfs. Even if in Java projects each class is usually its own file, here we have two examples of large double-class files. Both of them have a lot of lines of code and many statements. They are potential god classes in the system – the object that knows too much or does too much. The god objects are examples of an anti-pattern, something which should be avoided. A common programming technique is to separate a large problem into several smaller problems (a divide and conquer strategy) and create solutions for each of them. Once the smaller problems are solved, the big problem as a whole has been solved as well. Our Giant Dwarfs are also quite flat – having not so many methods, but a lot of attributes which they define – they could also be just data classes (data containers – for the data used by other classes). Such classes do not contain any additional functionality and cannot independently operate on the data they own. A closer look to the source code or more exact visualized data could help specify the case. If we had statistics about lines of code per method, we could use colour blocks to identify all the methods bigger than 20 loc as those are potentially hard to understand and maintain (so-called brain methods). We could even look at the huge amount of statements, pull out the if ones and identify the “Most Promising” classes, through which their creators too often asked “Why should I create a new method for each process if I can only add an if?”

From our small example we could also sort out Dummies – an overly commented code (>40%) which is not necessarily a blessing, as it can be considered “an insult to the intelligence of the reader” (Spoida 2002, online). Such classes could have a potential for refactoring: “whenever we feel the need to comment something, we write a method instead” writes Fowler when he calls for more decomposing of methods (Fowler, p. 64). On the other hand, the opposite of them, very poorly commented files (where the percentage of comment is lower than 20%) could also be the “Hugger-muggers” – a “secret” code with unclear functionality. The less commented code can also be the exact opposite; those brilliantly named files, which use small methods and names them so clearly they do not need comments. Only as a side note: when I started learning programming, what I found most impressive was that there is good and bad – good means it runs, bad – it does not. I still cannot really program, but I know, that “to run” or “not to run” are quite far from the criteria for a good and bad code. Without additional information on the context and content, or more experience, we can see patterns without being sure about their interpretation.

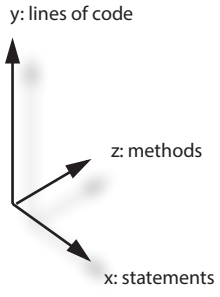


3.4. Soft World

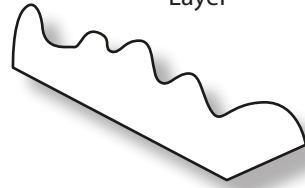
Consider a virtual reality system developed for training software development and software architecture basics. It includes a virtual surface and virtual objects created from software statistics and a gesture interaction system. The user wears a head-mounted display, and uses two hand controllers in order to interact with the room-scale virtual environment. The HMD and controllers communicate 3D coordinates of the user's hands and head to the VR system. Coordinate data are used to interpret actions of the user. The 3D model of the software system is placed in front of the user, where she can interact with it.

In front of the user is a terrain which represents a specific software directory, its files are peaks, where height (y) represents lines of code, the width (x) are statements and depth (z) are methods. Data set used for creating terrains is the same as in case of objects in level one, but is easily replaceable. The user stays on a plateau and around her there are more terrains which are in the size of islands, one for each directory: there is Api, Business, UI, Util, Webapp. One terrain (in this case a Utility directory) is displayed as a small model in the table size in front of the user and all five models are around the plateau as big walkable terrains. With the right controller she points to the small terrain and the outlines of a square prism are highlighted (on the small as well as on the big model). A text label appears on the top of the prism, which shows information about the file (the file name and the values of x, y and z). Simultaneously, a screen is shown where she can scroll up and down through the source code of the highlighted file.

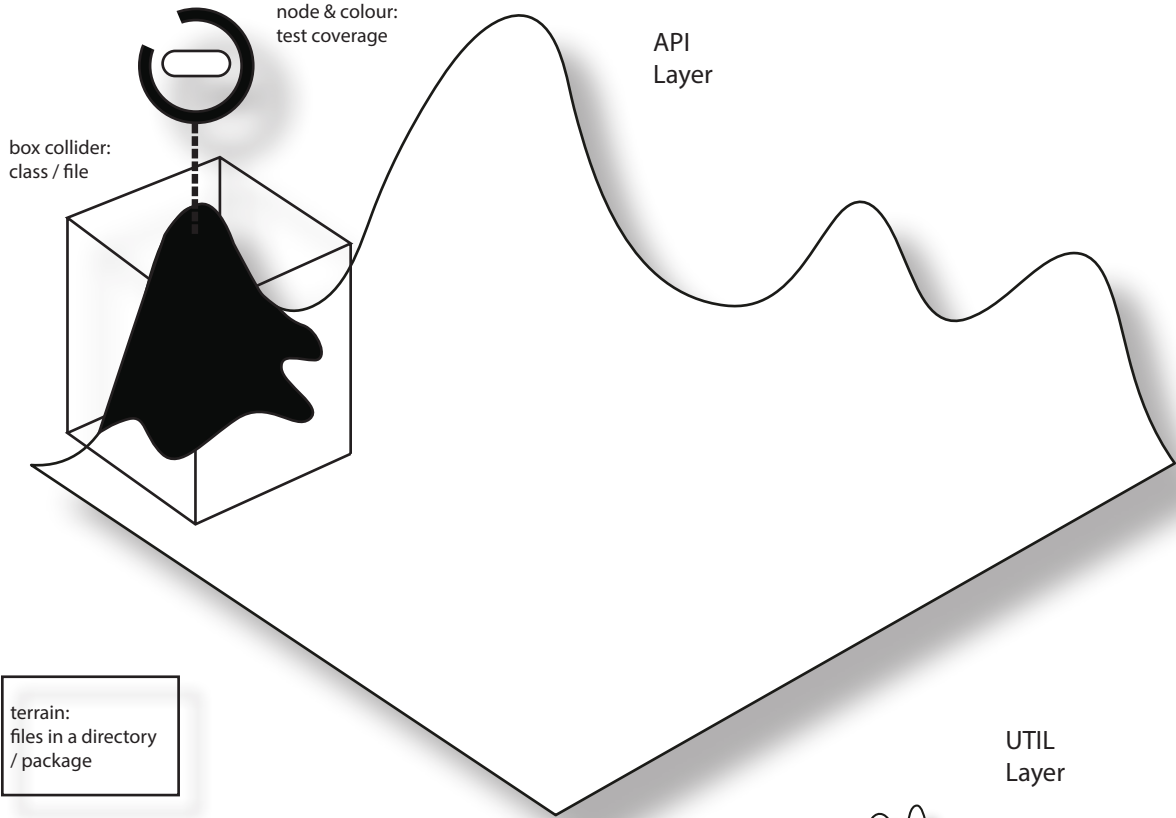
She can walk around the small model. With the left hand controller, she can also teleport in the further locations. On her right side, there are cubes with icons and labels of all five terrains which work on click as direct teleports to selected terrains. The terrains have different colours, according to the test coverage data. Where there was no unit tests, the terrain is black. Where it has been tested, the colour and additional objects show the results of testing. Green slot means tested, a text node shows the result of testing and a floating circle (similar to a 3D pie chart diagram) represents the % value of unit test. Where there was a unit test written but covered 0% of code, the area is red. Thus, she can walk around and explore further followed by all the data labels facing her, so she can read them anytime if they were a bit bigger (Ooh, we need the resize function here, too).



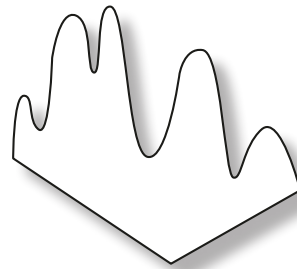
WEBAPP
Layer



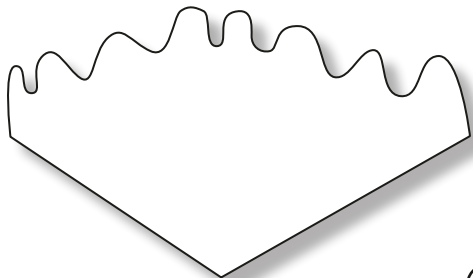
API
Layer



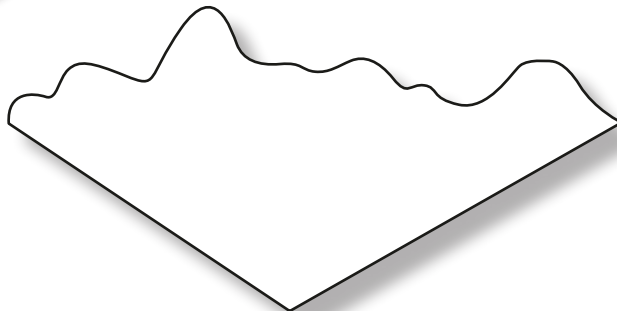
UTIL
Layer



UI
Layer



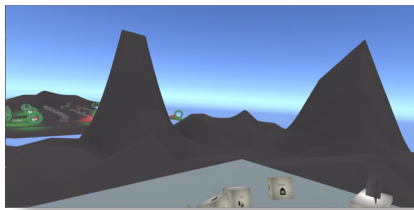
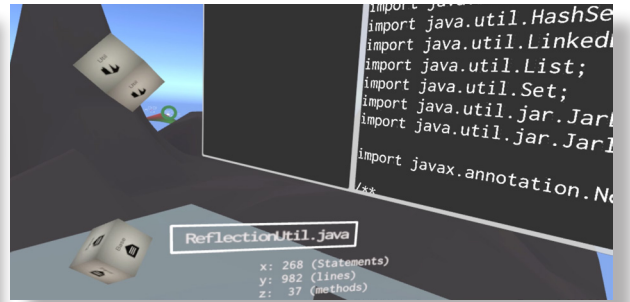
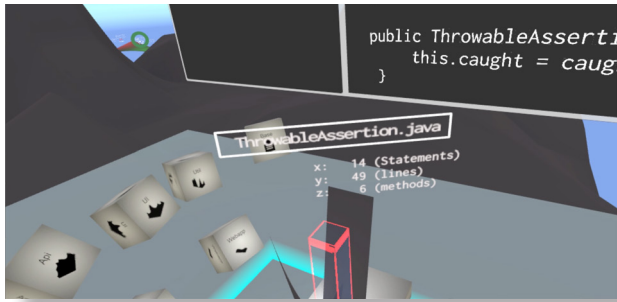
BUSINESS
Layer



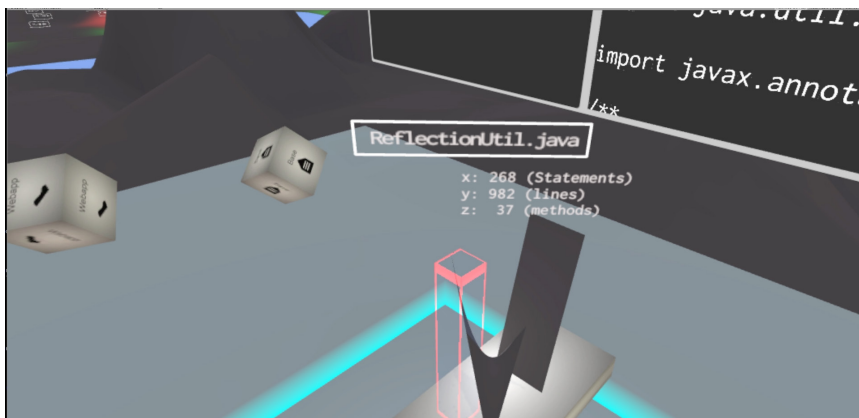
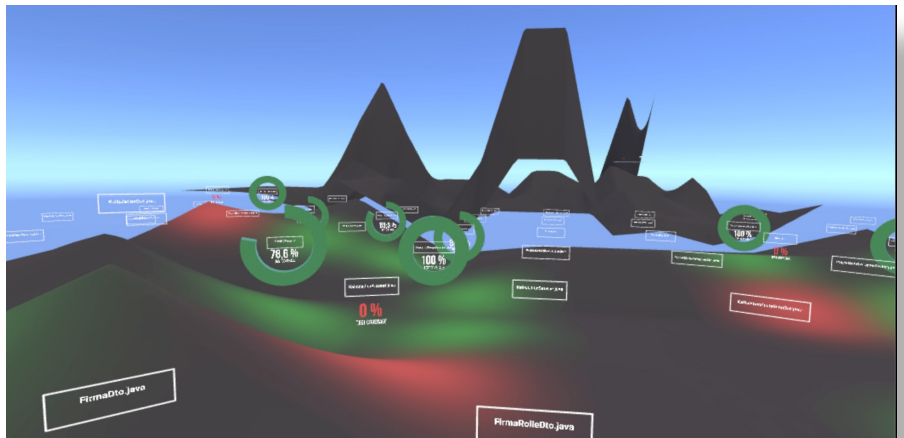
3.5. VR ID Model for the Metaphor

Data elements	Description	Data sources	Objects	Graphics
name; metrics or other data from sonarQube	GOAL: to identify software data inflows coming into the VR interface. INVOLVES: a, defining the data source; b, describing what the data represents; c, specifying the data set or other data parsing possibilities	GOAL: to identify data source coming into the VR interface. INVOLVES: The interface can receive data from three sources: user / physical devices / other VR systems /	GOAL: to identify objects that have well defined roles and identities in the interface. INVOLVES: a) identifying potential objects mentioned in the interface description; b) deciding on legitimate objects; describing aggregate objects and its parts; c) distinguishing between virtual and physical objects (the virtual ones are generated by computer, the physical ones are physical entities that interact with VR system – they may or may not require modelling).	GOAL: to model the virtual objects identified in previous steps; specify a description of graphical needs of virtual objects. INVOLVES: description of what kinds of graphical representations are needed for each object, and its parts, if any
Coverage on files (% value of unit test on each file on which test has been written) on top of the surfaces created out of code size statistics (metrics from SonarQube).	Input: data from SonarCube about test coverage of the files. Visualises where the test coverage is missing. Data set is currently in csv format and parsed with HeidiSQL. The coverage test reports are an additional layer of information on top of the terrain (additional models and colours).	As in the previous metaphor. The only actual interactive data are the source code, which are projected on the screen when a file is indicated.	Potential objects for UI are: "Terrains" created out of the data directories: Api, Business, Util, UI, Webapp. All five directories are present as (1.) Small models – usable for interaction (2.) Big models – explorable and walkable, (3.) Part of UI menu. The "Home-base" is a base where small models are grouped in front of the user; the "Screen" to show a source code of the files. The "Keyboard" for simple writing task at the code screen. The "Teleport-menu" a 3D UI menu for changing location. "PieChart-Rings" for showing data coverage. Aggregate objects are: "Terrains" – consists of the surface mesh; "intractable box collider" for each file, "text node"- provides additional info; "Screens" – shows source code, is scrollable, is writeable (has two sections); "Keyboard" with t9 can be used to write short notices to the code file (ideally possible to export); "Pie-chart" showing test coverage have a text-node with info about file name, % of coverage and a pie-chart object representing the same % coverage but as an object so you can see it from further way. Physical and Virtual objects are all objects in VR environment except the hand controllers, they are physical ones, for which the standard predefined virtual models are being used. Their functions are based on VR tool kit (teleporting, grabbing, pointing).	Data terrain modelled out of the bitmap image of data. Since the image creates grayscale pixel out of values in a specific data column, we can trace all files back (x, y, z values). Each file gets its own cubical box collider with a middle-centered text node which shows the file name. The terrain needs a texture which represents the data coverage (black is no test, green is tested, red is test with 0% coverage). The texture is currently created manually, but could also be based on data. Pie-Chart rings are 3D objects, they are also representing the % of coverage and floating over the terrain at the place of tested files. The text node is additional information about the exact % of test coverage (could be enhanced of other information e.g. the amount of uncovered lines or complexity grade of file: "VerteilungExcellImport.java; Coverage: 78.8%; Uncovered Lines: 21; Complexity: 52). The screen is a standard canvas element with a text field where the original source code is loaded. It could actually be made as an original SonarQube browser window open in VR (possible enhancement for the next version). The current version does not support a scripting reference which is kind of impractical for coders. The canvas element has two parts, left smaller one for writing, right for showing the text. Both of them have a title field in the upper part of the canvas which shows name and path of the file as well as a snapshot number. Possible enhancement: curved screen for better viewing. Keyboard: standard keyboard from SteamVR plugin. UI menu: Boxes with icons for each of the teleport destination. Possible enhancement – menu on the left collider in form of "world in hand" or in "tilt-brush" style.

Behaviours	Interactions	Internal communication	External communication
<p>GOAL: to identify behaviours exhibited by objects; classify them into simple physical, simple magical, or composite behaviour categories; and to describe them in enough detail for designers to visualize the behaviours.</p> <p>INVOLVES: a) identifying the behaviours from the description; b) classifying the behaviours into simple and composite categories; c) classifying the simple behaviours into physical and magical behaviour categories; d) for composite behaviours, specifying the sequences in which simple behaviours are to be combined for producing the composite ones.</p>	<p>GOAL: to specify where inputs of interface objects come from and how they change object behaviours.</p> <p>INVOLVES: a) identifying interaction requests to objects; b) identifying the behavioural changes caused by these requests and which behavioural components will be notified about these changes.</p>	<p>GOAL: to specify control and coordination needs for internal communications among the components of objects in order to avoid potential conflicts in object behaviour</p> <p>INVOLVES: a) examining all communication requests and behavioural changes that are caused by these requests; b) identifying communication requests that may cause potential conflicts; c) deciding how to prioritize, sequence, hold or deny the communication requests to avoid potential conflicts.</p>	<p>GOAL: to specify control and coordination needs for external communications of the objects.</p> <p>INVOLVES: a) identifying communication inflows into the object, and their sources; b) communications outflows from the object, and their destinations; c) describing time and buffering semantics of external communications of the object.</p>
<p>Behaviours are: (1.) Highlighting the box colliders of files upon pointing (highlighting is a magical behaviour, pointing physical, composite); (2.) Showing/hiding the screen with the source code (magical, simple or part of composite b.); (3.) Scrolling of the source code text (physical); (4.) Showing/hiding the keyboard (magical, simple); (5.) Writing (combined); (6.) Teleporting (magical, simple as well as composite – see further down); (7.) Showing/hiding the additional text info on files (same as highlighting); (8.) All the text nodes are facing camera, so they are readable (magical and physical – needs to listen for the head position). Composite behaviours are: (1-2-7, with possibility of additional 3 and 4-5) on pointing to the peak at terrain a box collider is highlighted (ideally on both of the models at the same time – in front of the user both on the small and big terrain – to show the two are the same data, though on a different scale) and additional data (file name and x, y,z axes are displayed on the top of the box collider). Simultaneously, the screen with the source code appears where the user can scroll over the source code. Teleporting is a simple behaviour, but has two options. The user can teleport between terrains with help of UI buttons (go directly to a selected location) and he or she can also walk around in the room-scale VR through walking in a physical environment or teleporting with help of a pointer bound to the left hand controller (a bezier pointer which shows a reachable destination for local teleporting).</p>	<p>UI teleport menu boxes also have composite behaviours: they are clickable (the user can teleport upon the click of the controller trigger) and are graspable (user can grab and move them), too. Grabbing gets input from the hand controller (side buttons are triggered) and informs the box object on which the interaction code is written. Highlighting: call on box colliders, needs a trigger (hand controller pointer) which notifies the “text-node” object (it will be activated), the Screen scrolling also needs the hand pointer and react to the Touch-pad scroll. The text-node objects listen to the head position (camera-rig) when facing the camera and turn according to its position (quaternion is always set to have 90-degree angle).</p>	<p>Potential conflicts are on highlighting of the box-colliders and simultaneous activation of the screen (too fast pointing on different files can cause blinking (fast changing) of the screen, so there can be an optional condition of activating screen on secondary click or not calling the screen if the trigger input is too short (say under 0.5sec).</p>	<p>The same as with previous model, since we are not getting any real time data out of other programs, external communication inflow are the 3D coordinates of the user head and hands in the room-scale VR, for which the technological requirements are regulated by the provider of the HMD software and its development environment (VIVE, SteamVR and Unity).</p>

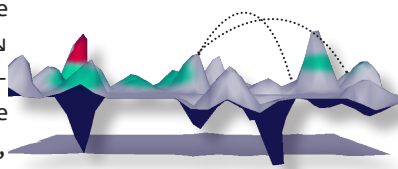


3.5. "SOFT WORLD"



Discussion & What I Wanted and Never Managed

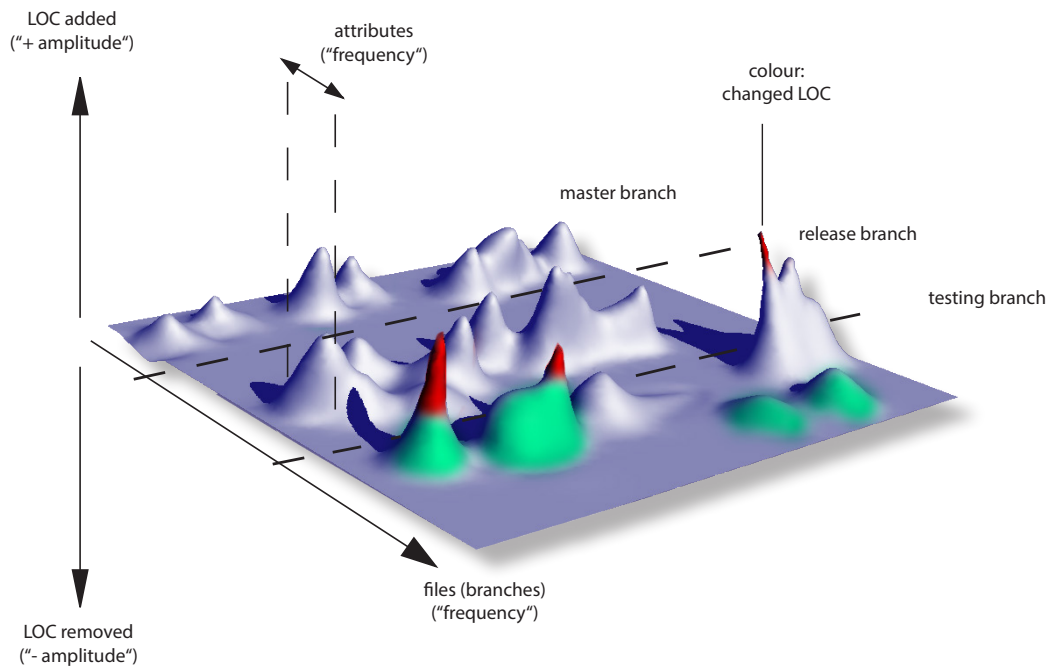
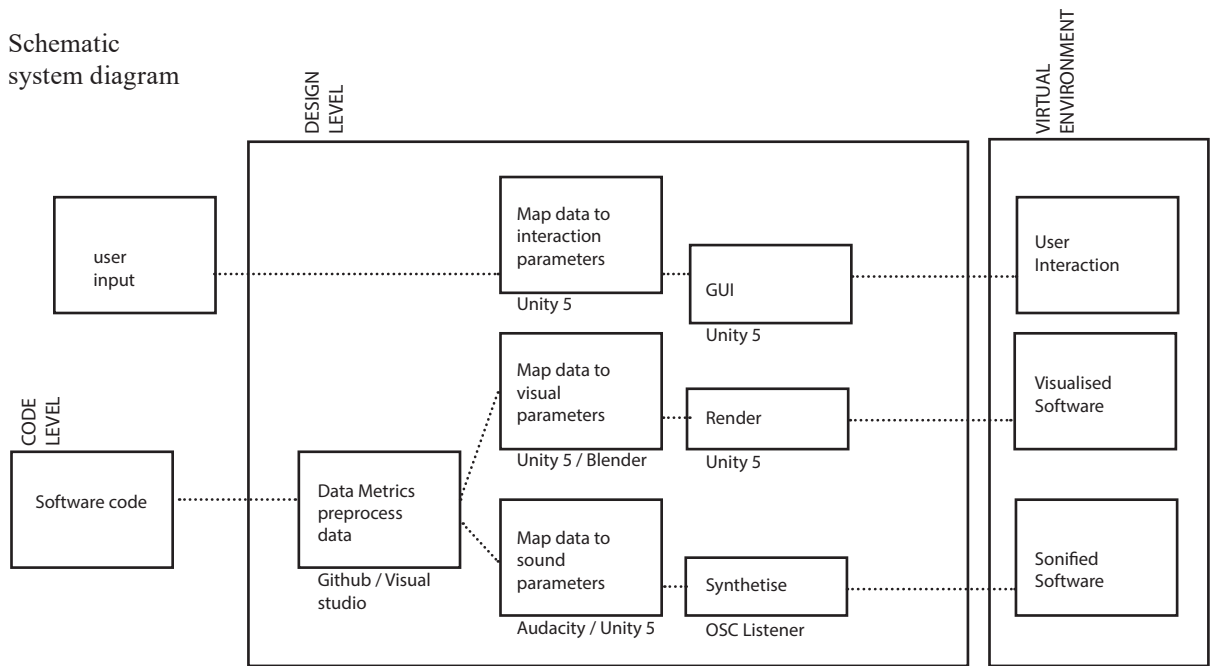
Visualize instability

One problem of the static code visualisation I made is that it takes just the status of the code at the moment it was committed (not even that, in our case when the snapshot of the system has been created). We lose a constant review of important changes. The amount of changes in the code are important for measuring the instability of the software. Parts with many changes, or classes showing constant overwriting are the areas of interest for software developers. As a potential “smell”, regions need to be identified and monitored. Dynamically generated terrains (similar to what we see in Soft World) could be used for mapping changes within the source code as well. An example of such data mapping can be seen at image  which represents the y-axis of the changes in the source code. The upper part represents the addition of LOC, the lower part represents the removal of LOC, the colour refers to the amount of changed LOC. The x-axis refers to the branches of the code (e.g. master, testing, release). It also shows which the current “head” branch is. The z-axis refers to the time and its changes.

To see what effect the changes had on the other parts of the software would be possible through visualizing the dependencies, e.g. on methods (would be probably easier to check on c# code). The dependencies could be shown as bezier curved connections between the classes (peaks of the terrain). Through the force-directed layout of those connection, we would see bridges between classes and even between the layers or directories, (e.g. Api to Business or to UI).

When you are in VR, you want to do stuff, not just watch. Even simple task as grasping and moving cubes have much stronger effects than long produced surroundings with which you cannot interact. Out of such small findings, I could start a litany of all ideas I had but never managed to create properly: a VR software architecture simulator with built elements (sort of Lego version just with different front-end application possibilities, services, business-logic and hosting services) which users can freely connect and play with and plan their software architecture; to create an augmented reality app for playing with a data shader; write own fictive version of software development ISO norm; create dynamic visualisation for cardboard – a call graph as a roller-coaster; a hierarchical code-tree model shaking simulator – to get rid of dead/unused code in project files; and “Oh, I heard a bug!” a.k.a. “Silent Code Disco” a sonification of code where sound would be generated procedurally together with the data mesh.

Schematic system diagram



Epilogue

As usually by finishing, there is the fascinating mix of simultaneous relief and frustration. Happy that it is almost over, but sad that you cannot just start now – to work on what you wanted with all the knowledge about mistakes you made. My motivation was optimistically naive: not just to understand the complex systems of software development, but also to describe them, and not just to describe and visualize them but to question them, even to question their visual representation back again (similarly to quoted Benjamin Bratton at the beginning of the thesis). I wanted to learn C# and software architecture basics, get more experience in creating 3D content and interaction. To summarize, even if I am quite sceptical about the outcomes of this thesis, the ideas I got now and the knowledge I have gained makes me satisfied. I believe there will also be a place for playing around further, this time together with other people – we have the playground now and I got pretty good in creating buttons and the like in VR.

Further information

<http://vrsoftwareviz.tumblr.com/>

<http://virtualmaterialism.com/>

Bibliography

- Ball, T., Eik, S.(1996): Software Visualization in the large. In: *Computer*, vol. 29, no. 4, IEEE Computer Society Press. pp. 33-34.
- Berrigan, R., Grundy, J., Panas, T. (2003): A 3D Metaphor for Software Production Visualization [online:] https://www.researchgate.net/publication/4026411_A_3D_metaphor_for_software_production_visualization accessed: 11.5.2016
- Berry, D.M., Pawlik, J. (2005): What is code? A conversation with Deleuze, Guattari and code* In: *Kritikos: an international and interdisciplinary journal of postmodern cultural sound, text and image*. Volume 2, December 2005, ISSN 1552-5112 [online:] <http://intertheory.org/berry.htm> accessed: 10.12.2016
- Diehl, S. (2007): *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of Software*. Springer.
- Donalek, Djorgovski, Davidoff, Cioc, Wang, Longo, Norris, Zhang, Lawler, Yeh, Mahabal, Graham, Drake (2014): Immersive and collaborative data visualization using virtual reality platforms. In: *IEEE Xplore: 08 January 2015* [online:] <https://arxiv.org/ftp/arxiv/papers/1410/1410.7670.pdf> accessed: 2.7.2016
- Enri, D., S. (2010): Codemap: Improving the Mental Model of Software Developers through Cartographic Visualization [online:] <http://scg.unibe.ch/archive/masters/Erni10a.pdf> accessed: 20.7.2016
- Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D. (2014): *Refactoring: Improving the Design of Existing Code*. MA: Addison-Wesley, 455p.
- Gabriel, R.P. (1996): *Patterns of Software: Tales from the Software Community*. NY: Oxford University Press, 235p.
- Johnson, M., Lakoff, G. (1980): The Metaphorical Structure of the Human Conceptual System. In: *COGNITIVE SCIENCE* 4, 195-208 p. [online:] <http://www.fflch.usp.br/df/opessoa/Lakoff-Johnson-Metaphorical-Structure.pdf> accessed: 4.7.2016
- Kosara, R., Ziemkiewicz, C., (2008): The Shaping of Information by Visual Metaphors [online:] https://research.tableau.com/sites/default/files/Ziemkiewicz_InfoVis_2008.pdf accessed: 10.5.2016
- Lanza, M., Greevy, O., Christoph Wyseier, Ch. (2006): Visualizing Live Software Systems in 3D [online:] <http://www.inf.usi.ch/faculty/lanza/Downloads/Gree06a.pdf> accessed: 8.8.2016
- Riva, C. (2004): View-Based Software Architecture Reconstruction. PhD thesis, Vienna University of Technology, 2004. In: *Corum ii. In Proceedings of the 5th Working Conference on Reverse Engineering*, pages 154-163, IEEE Computer Society Press.
- Souza, R., Silva, B., Mendes, T., Mendonça, M. (2012): SkyscrapAR: An Augmented Reality: Visualization for Software Evolution [online:] <http://reuse.cos.ufrj.br/wbvs2012/papers/wbvs03.pdf> accessed: 15.8.2016
- Spuida, B. (2002): The fine Art of Commenting [online:] <http://www.icsharpcode.net/TechNotes/Commenting20020413.pdf> accessed: 5.12.2016
- Tinnell, J. (2016): From WIMP to ATLAS: Rhetorical Figures of Ubiquitous Computing. In: *Computational Culturea journal of software studies* [online:] <http://computationalculture.net/article/from-wimp-to-atlas-rhetorical-figures-of-ubiquitous-computing> accessed: 1.9.2016
- Teyseyre, a. R., Campo, M. R. (2009): An Overview of 3D Software Visualization, In: *Ieee Transactions On Visualization And Computer Graphics* Jan-Feb 2009, pp. 87- 105 [online]: http://ieeexplore.ieee.org/ieee_pilot/articles/01/ttg2009010087/article.html accessed: 9.4. 2016
- Tanriverdi, V., Jacob, R.K. (2001): VRID: A Design Model and Methodology for Developing Virtual Reality Interfaces. [online:] <https://www.cs.tufts.edu/~jacob/papers/vrst01.tanriverdi.pdf> accessed: 14.10.2016
- Tufte, E. D. (1997): *Visual Explanations: Images and Quantities, Evidence and Narrative*. Cheshire, Connecticut: Graphic Press, 157p.
- Wettel, R.(2010): Software Systems as Cities [online:] <http://wettel.github.io/download/Wette108a-icse-tooldemo.pdf>

<i>software (definition)</i>	<i>software as an animal</i>	<i>software as a food</i>	<i>software as a person</i>	<i>favourite music</i>	<i>hated music</i>	<i>role</i>
Abbildung von Anforderungen und Funktionsweisen in einem System	Elefant	 Pizza	-	Black Music, Charts	Folksmusic	Entwickler
Software ist automatisierte Kopfarbeit	Ein Nashorn; groß und schwer. Manchmal wirkt es langsam, aber es kann auch schnell rennen. Und es ist durchsetzungsfähig.	Scharfes Chili mit viel Brot: Schmeckt gut, bringt dich zum Schwitzen. Und wenn man mit Brot "löschen" möchte, kann selbst das zu viel sein.	-	Alternative, Country	 Italo-pop	Projektleitung
-	A camel or elephant because they can carry people	-	-	Fast music (hardcoretechno or metal)	Slow music (electronic)	Entwickler
Schnittstelle zwischen Benutzer und PC um die Funktionen, Operationen, etc., zu vereinfachen.	Kamel - kann mehrere Personen transportieren	Baguette - man kann es belegen	Otto (Simpsons - the bus driver)	Hardcore, Electro	Deutscher "Gangster" Rap	Entwickler
Helping making things easier	Eierlegendewollmilchsau	-	Chuck Norris	Rock	Gabber, Techno	Entwickler
Set of instruction to solve a problem or doing a task	Snake, cause the logic is windy and curvy and not a straight like	Old bread, since it is dry and hard		Classic Rock	Techno	Entwickler
Ein Programm, das Erleichterung schafft und komplexe Anforderungen lösen kann.	Ohrwürmer, da sie gerne in solchen Bereichen "arbeiten".	Mais - ein Maiskolben hat viele Maiskörner, also viele kleine Einzelteile und sehr viele Informationen	-	Dubstep, Hardcore, Rock	Gangster-Rap, Volksmusik, Schlager, Blues, Jazz	Entwickler
-	schön und einsam	 Knobi-Spaghetti (Lecker + Einsam)	-	Ba	Techno	-
Nicht alles Neue ist gut	Dino	Kalter Kaffee	Forrest Gump	Indie	Volksmusik	Applicationmanagement
-	Esel - langsam und störrisch, aber beharrlich	ein zähes Stück Fleisch: schwer verdaulich	Winston Churchill: alt, schwerfällig, clever	Rock, Blues, Jazz, Pop	Hip Hop, Klezmer	Berater