INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Evaluation of Virtual Reality based Mesh Saliency Maps

Richard Metzler

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Markus Wiedemann
Abgabetermin:	29. September 2017

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 29. September 2017

.....
(Unterschrift des Kandidaten)

Kurzzusammenfassung

Im Rahmen dieser Arbeit wird untersucht, welche Teile von 3D Objekten als visuell interessanter empfunden werden als andere und wie sich die *wahrgenommene Wichtigkeit* auf der Oberfläche solcher Modelle verteilt. Zwei grundsätzlich verschiedene Lösungsansätze zu dieser Frage werden zu Hilfe gezogen. Es werden automatisch, anhand eines mathematisch begründeten Prozesses errechnete Verteilungen von Wichtigkeit betrachtet. Mit diesen Ergebnissen werden Verteilungen wahrgenommener Wichtigkeit verglichen, welche durch direkte Nutzer-Interaktion mit einer Auswahl-Applikation entstanden sind, die im Rahmen dieser Arbeit erstellt wurde. Das Ziel dieser Arbeit ist das Bereitstellen einer Grundlage von Daten, welche den Anfang einer Evaluation von Unterschieden dieser beiden Vorgehensweisen ermöglicht.

32 Nutzer nahmen bei der Studie teil, welche durchgeführt wurde um die Grundlage für diese Diskussion zu schaffen. Die Teilnehmer nutzten die im Rahmen dieser Arbeit erstellte Software um Punkt-weise Teile von 3D Objekten, die sie als interessant empfanden, zu markieren. Gleichzeitig wurde ein grundlegendes Differenz-Maß entwickelt, welches etwaige Unterschiede beschreiben soll. Dieses hat zum Ziel, eine Prozentzahl als Ergebnis zu liefern, welche beschreibt wie stark die durchschnittliche Nutzer-Auswahl von den vorab errechneten Ergebnissen abweicht.

Während eindeutige Ähnlichkeiten erkennbar sind und diese sich auch in den Kennzahlen widerspiegeln, besteht diese Evaluation mehr auf qualitative Beschreibungen von Tendenzen und Mustern, da eine fundierte, umfassende Analyse von Unterschieden solcher Datentypen den Rahmen dieser Arbeit gesprengt hätte.

Abstract

The subject of this work is to inspect which parts of 3D objects are visually interesting, which are not, and how *perceived importance* is distributed on the surface of such models. Two fundamentally different approaches to this question are compared in this work. First, *importance maps* computed by an automated, mathematically founded procedure are considered. Second, distributions of perceived importance, based on direct user input, gathered from an application based on virtual reality technology, are considered. The goal of this work is to collect enough data to start an evaluation of differences between the results of these two approaches.

In order to establish a foundation of this discussion, a user study was conducted with 32 participants. They would each use the software developed in the scope of this work to select parts of objects they deemed *interesting*. Beforehand, a basic measure of difference was conceptualised in order to describe differences that were speculated to be found. The goal was to compute one percentage-ratio which, based on vertex-wise comparison of importance values provided by user input and computation, indicates how much the user selection differs from the results of computation.

While there are clear similarities, this evaluation os more based on observation of tendencies and patterns since a sound, exhaustive analysis of differences in such types of data would have gone beyond the scope of this work.

Contents

1 Introduction **1**

2 Related work **5**
2.1 Mesh saliency and human perception 5
2.2 Human attention in Virtual Reality 8

3 Concept **11**
3.1 Implementing a selection application for an immersive VR environment . . . 11
3.1.1 Spatial indexing of 3D data 11
3.1.2 Selection process 14
3.2 Conduct user study with the selection application 15
3.3 Measure of differences 16
3.3.1 Terms and abbreviations 16
3.3.2 unweighted difference ratio 17
3.3.3 weighted difference ratio 18
3.3.4 step-wise summery 19

4 Selection Application **23**
4.1 Additional third party libraries 23
4.1.1 OpenGL 23
4.1.2 GLUT 24
4.1.3 GLEW 24
4.1.4 ASSIMP 24
4.2 Relevant class files 25
4.2.1 Object 25
4.2.2 Mesh 25
4.2.3 Octree 25
4.3 Key Features 25
4.3.1 Spatial indexing via Octree 26
4.3.1.1 Root constructor `ocTree()` 26
4.3.1.2 Subtree constructor `ocTree()` 27
4.3.1.3 `getRootDimensions()` 28
4.3.1.4 `setDimensions()` 28
4.3.1.5 `split()` 29
4.3.1.6 `getNodeByIdentifierArray()` 30
4.3.1.7 `buildTreeRecursively()` 31
4.3.2 User selection 33
4.3.2.1 `addVerticesToSelectionByCoordinates()` 33
4.3.2.2 `removeVerticesFromSelectionByCoordinates()` . . 38
4.3.3 Tracking selection 38

4.3.4 Testing setup . . . 39
4.3.4.1 Test files . . . 40
4.3.4.2 Helper and debugging functions . . . 40
4.3.4.3 Testing at the V2C . . . 42

5 User study **45**
5.1 Hardware used . . . 45
5.2 Models used . . . 49
5.3 Tasks given . . . 50
5.4 Questionnaire . . . 50
5.5 Shortcomings of the study design . . . 51
5.5.1 Task, instructions and questions . . . 51
5.5.2 Technical issues . . . 52

6 Results and discussion **53**
6.1 Participant demographics . . . 53
6.2 Feedback from the questionnaire . . . 54
6.3 Individual feedback and comments . . . 55
6.4 Saliency and difference maps . . . 56
6.4.1 Bunny . . . 56
6.4.2 Cow . . . 59
6.4.3 P51 . . . 62
6.5 Evaluation of differences . . . 65
6.5.1 Measure of difference . . . 65
6.5.2 Observation of saliency maps . . . 66
6.5.3 Mesh saliency with mechanical objects . . . 67

7 Conclusion and future work **69**
7.1 Insights and conclusions . . . 69
7.2 Future work . . . 70
7.2.1 Improving measure of difference . . . 71
7.2.2 Altering variables . . . 71
7.2.3 More extensive user study . . . 72
7.2.4 Unused data . . . 72

List of Figures **75**

Bibliography **77**

1 Introduction

Beginning in the 1950's, virtual reality technology [Ste92] has been continuously researched and improved and its professional relevance is becoming ever more present today. There is a plenitude of recent works showing that it bears great potential and positive possible contributions to architecture and construction [SM14], [LPP15], [SJRT13], health care and psychotherapy [BB14], [MF14], [dRKH+14], engineering and industrial design [MEC14], [WCAH+16], gaming and home entertainment [VCSF16], [Zyd05] and education [MGC+14], [OF+15]. One must also consider that this technology can help gaining new insights and open up new perspectives into greater, more abstract matters of social, environmental and economic manner [OHH+15], [NNVL+16].

With resources such as memory and computing power becoming more and more available at ever-increasing rates, 3D objects and their mesh representations are constantly growing in complexity and size, in terms of shaders, texture maps as well as the sheer number of vertices. Still, many professional applications revolving around interaction with such models require means of displaying them in real-time without significant perceived loss of quality to ensure a smooth and fast workflow. This is where mesh simplification and segmentation plays an important role [WL10], [SG01], [ZLSZ12].

This issue becomes even more pressing in a professional, commercial context where access to state of the art, high-performance graphic processing units or render farms is not a given for everybody. With less computing power available, means of user-oriented, real-time rendering are of vital importance to a fast and unimpeded way of working on 3D assets. *Mesh saliency* is an automatic, mathematically founded procedure proposed in 2005 by Lee *et al.* [LVJ05]. It describes vertex-wise perceived importance a given 3D model, based on differences in local curvature and aims at providing a basis for so-called saliency-based complexity reduction of such objects. Here, the number of vertices and triangles that make up a 3D geometry are automatically reduced, often resulting in gross losses of quality. Concepts such as *mesh saliency* are designed to uphold important details to a certain extent, even after reduction has taking place. Put simply, the idea is to merge more vertices in regions that are deemed to be of low visual interest while keeping the number of vertices high in the important areas.

Lee *et al.* state that their procedure is based on characteristics of human perception and cognition. However, a scientific evaluation as to whether distributions of perceived importance that result from using it are congruent with real impressions of users or not, is yet to be conducted. To tackle this problem, direct user input describing what they perceive as *important*, is needed. This work includes a description of a virtual reality based *selection application* which allows vertex-wise selection of *important* regions.

For this selection process, the Virtual Reality and Visualisation Centre's five-sided projection installation at the Leibniz Supercomputing Centre in Munich was available [v2c]. This installation creates interactive, immersive virtual reality environments via multiple projectors and tracking sensors. Users only need to wear a lightweight pair of stereo shutter glasses that are synchronised with the projectors and thus can seperate two images for the specta-

tor, one for each eye. The glasses are equipped with a tracking system so that their exact position, orientation and tilt can be captured in real-time, allowing the computation of the users perspective in 3D scene at any time. Based on this perspective, the projectors use the walls as projection surfaces and throw live images resembling what the user would see if he/she were physically in the virtual scene onto the walls. The projection installation grants a virtual reality experience which is enhanced by the fact that the user only needs to wear a pair of glasses instead of a fully sized headset. From a user perspective, another advantage of such a setup is the fact that there are no cables connected to the glasses which can evoke a feeling of inhibition or the constant worry of stumbling over and accidentally damaging them.

This work is segmented into three major tasks. First, a piece of software that allows real-time interaction in an immersive virtual reality environment is implemented. This so-called *selection application* has to let users select and deselect vertices of 3D objects in an easy, fast way and provide clear visual feedback on what is currently selected and what is not. Then, a user study is conducted. Participants are asked to use the *selection application* in the five-sided projection installation of the V2C [v2c] and select parts of three 3D objects that they deem *important*. With this data, a comparison between automatically calculated *importance* (*mesh saliency*) maps and *user saliency* maps can begin. The last step is the to start such an evaluation. A measure of difference is conceptualised for this work, trying to describe how much user selections differ from what parts of 3D objects are deem *interesting* by automatic computation. Based on this measure as well as structured observations of the resulting maps, this evaluation and discussion can begin.

The measure of difference shows that, for the three objects used during the user study, user selections differ from computed *importance maps* by roughly 23%, 27% and 38%. However, the validity of these ratios can be doubted because the measure of difference, being a first quick suggestion, lacks expressiveness as discussed in section 6.5.1. Furthermore, observations shows that, while there are clear similarities in what is deemed *important* by computation and users, individual, highly object-dependent differences are to be expected.

Figure 1.1 depicts interaction with the application implemented for this work. Details on its implementation can be found in chapter 4, the general structure and conceptual part of this work is described in chapter 3. For more related work on the topic, see chapter 2.

Details on the user study can be found in chapter 5, its results are presented and discussed in chapter 6. Lastly, for a quick summery of this work as well as an outlook on possible future work, see chapter 7.

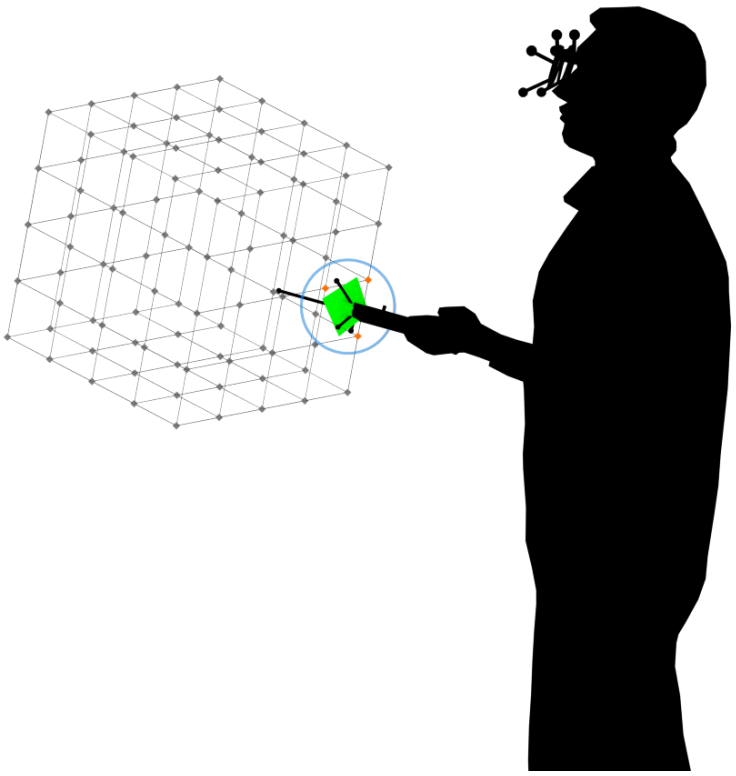

Figure 1.1: Schematic depiction of how the *selection application* works from a user perspective

2 Related work

Since this work focuses on the impact that being immersed in an interactive virtual reality scene has on human attention when both focusing and performing tasks on 3D objects, this section will be subdivided into two parts. First, publications discussing *mesh saliency*, a technique used term to predict regional percieved importance of digital representations of real-world (or synthetic) objects as well as 2D images, will be presented and briefly described. The second section will provide a look into human behavior, based on cognition and outside stimuli in virtual reality environments. These two sets of scientific works will provide a solid knowledge of terms and methods commonly used in this field and describe the current state of the art.

2.1 Mesh saliency and human perception

Research on what human perception guides us to focus our attention on when presented a 3D representation of an object was begun just past the year 2000 and has been a continuous effort ever since. One commonly cited publication in this field is Lee *et al.* [LVJ05]. Based on low-level human visual attention [KU87], it introduces the term *mesh saliency*, a measure for regional based importance of 3D meshes and also presents a way to compute it. This fully automatic process successfully predicts what would be classified by most observers as prominent, visually interesting regions on a mesh, thus allowing mesh operations such as simplification [CMS98] and segmentation [Sha08] to produce results that are more appealing to the beholder.

The model for computing *mesh saliency* is based on a center-surround comparison of local curvature. It is scale-dependent on a *saliency factor* ε, which is based on the diagonal of the objects bounding box, and is able to identify salient features of a mesh, depending on their surrounding area. Geometrically complex regions, for example a large patch containing lots of bumps of similar size, will be rightfully dismissed as, in most cases, regions that are not interesting from a human perceptional stance.

Taking a closer look at the basic formula through which saliency for any vertex of a mesh can be computed according to Lee *et al.* helps understanding the underlying concept. As a first step, the mean curvature map for a mesh, describing mean local curvature values on a point-level for each of its vertices, needs to be calculated via commonly known approaches such as [Tau95]. The resulting mean curvature map $\mathscr{C}$ defines a mapping from each vertex of a mesh to its mean curvature $\mathscr{C}(v)$. Using a distance measure such as the Euclidean or geodesic method, one can compute the neighbourhood $N(v,\sigma)$ of a vertex v which then defines a set of points within a distance σ. The Euclidean appraoch was used in Lee *et al.* and subsequently in the formula below. Using these definitions, the authors denote the Gaussian-weighted average of the mean curvature by $G(\mathscr{C}(v),\sigma)$ and present the following way of computing it.

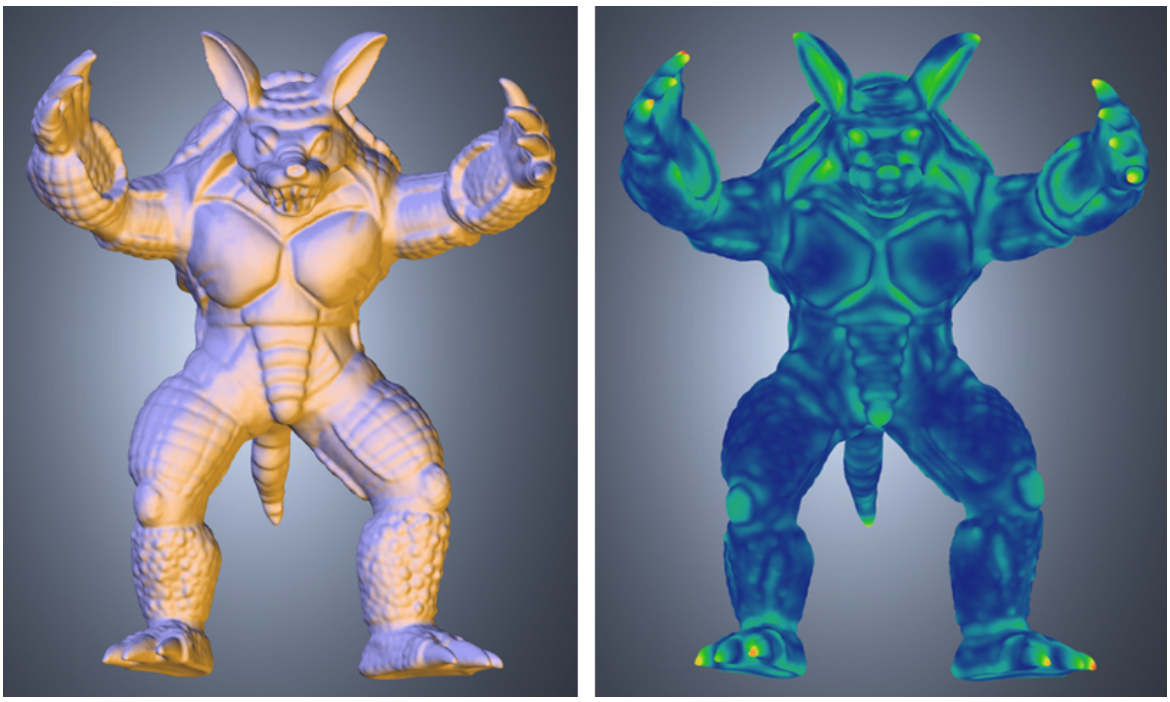

Figure 2.1: A model and its computes mesh saliency map. Published by Lee *et al.* [LVJ05]. Bright colors (yellow and red) indicate high saliency values for their respective vertices, dark colors (shades of blue) indicate low values.

$$G(\mathscr{C}(v),\sigma) = \frac{\sum_{x \in N(v,2\sigma)} \mathscr{C}(x) exp - ||x-v||^2 \backslash (2\sigma^2)}{\sum_{x \in N(v,2\sigma)} exp - ||x-v||^2 \backslash (2\sigma^2)}$$

For computation of the Gaussian-weighted average, a cut-off factor for the filter is assumed at a distance of 2σ, in other words twice the distance that a vertex can have to another vertex to still be considered in its neighbourhood. Based on these definitions, the saliency $\mathscr{S}(v)$ of a vertex v is defined as the abslute difference between the Gaussian-weighted averages, as seen in the formula below.

$$\mathscr{S}(v) = |G(\mathscr{C}(v),\sigma) - G(\mathscr{C}(v),2\sigma)|$$

In order to get more refined results, one can conduct multiple computations of *mesh saliency* with different values for σ. Lee *et al.* use the previously mentioned *saliency factor* ε with $\varepsilon \in 2,6$ in their paper, to generate multiple values for σ.

The concept of *mesh saliency* has since been refined, augmented and adapted to serve as a basis for a multitude of specific use-cases and applications. When processing single vertex saliency, Wu *et al.* [WSZL13] took into consideration not only the curvature of the region surrounding the vertex, but also the global context of it. In other words, for each vertex to be attributed a value describing its saliency, its *global rarity*, derived from comparing its features to those of every other vertex of the object, is computed. They performed a user

study in which they had participants choose one out of two saliency maps for a set of objects, presented in a random order. One map was generated using their approach, the other one with the model presented in Lee *et al.* Participants were asked to pick the one that was a closer representation of what they would have considered interesting regions and features. Since their method got picked in almost 58 per cent of cases, while the results produced by the model presented in Lee *et al.* were favored in about 42 per cent, this can be considered a true improvement of the way *mesh saliency* can be computed.

One approach to improve the method of finding salient elements in 2D images relied on paying extra attention to depth-information in [CHR13]. In this work, Ciptadi *et al.* found that better results in terms of automatic identification of objects and surfaces could indeed be achieved this way. Transferring these insights into a 3D context is easy since visually complex models often base on multiple image-maps describing, among other information, depth values on the surfaces of the model. In [PZV11], the authors took a more task-driven approach to contribute to the concept of *saliency*. Gathering colour- and depth information about real-world scenes using a Kinect sensor, they extracted semantic cues about surface heights, relative surface orientations and occluded edges. Based on that data, they computed combined saliency maps which allowed them to assign real-world objects to four different categories, enhancing ways a robotic system can interact with them, providing the best possible points where the objects can be grasped and whether they are in reach at all or not (due to occlusion by other objects).

Another recent work aimed at identifying single, distinct elements and objects of 3D models was presented in [Kos03] by Koschan *et al.* The authors propose a segmentation algorithm that utilises a human perception phenomenon known as the *minima rule* which suggests that contours of negative curvature minima can serve as boundaries of disjunct visual parts or elements. Another detailed comparison between automatically detected points of interests and what participants in a study actually declared as visually interesting points was drawn by Dutagaci *et al.* in [DCG12].

To verify the practical relevance of identifying salient regions and features on 3D meshes, Howlett *et al.* [HHO05] conducted a user study on whether it is possible to determine such features in advance. Based on observations gathered from eye-tracking device based user studies, they concluded that, especially with natural objects (animals, humans etc.), this was indeed the case. On top of that, they also reported a significant increase in visual fidelity on objects which were simplified based on saliency weight-maps, according to reports of study participants. Furthermore, in [KG03] the authors used user-guided simplification to preserve higher levels of detail in areas of 3D objects that people deemed important to the recognisability of the object. After performing mesh simplification according to [GH97], enhanced by taking user-derived weight maps into account, the authors observed what they described as perceptually improved approximations of input objects.

In another highly noteworthy work by Munaretti [Mun07], the concept of *mesh saliency* was extended to deformable, in other words animated, objects. The author presented a way to generate so-called *multi-pose saliency*, a combination of multiple saliency maps compoted for static poses of a mesh. These static meshes can also be interpreted as keyframed poses for dynamic deformation, which makes this work a potentially outstanding contribution to any field where 3D objects are being animated.

The author found a remarkable improvement of the original way of computing *mesh saliency* as presented by Lee *et al.* by using geodesic distance [SSK$^+$05] instead of euclidean distance when comparing local curvature values and implemented a way to compute

multiple saliency maps for different levels of detail.

2.2 Human attention in Virtual Reality

While navigation in virtual reality space via a traditional desktop setup with input devices such as a mouse and a keyboard still seems to allow users to perform better in basic tasks such as navigation, they generally perceived interaction via a head-mounted display more natural and intuitive [SDP+09]. It is worth noting though, that this work evaluated a series of user studies described in their respective papers which were published between 1997 and 2006. Thus, it is safe to assume that recent VR technology would get much better results in comparison. This was hinted at in the paper multiple times, mentioning the idiosyncrasies of the equipment used in the studies. The main tasks in the studies described in this paper included navigation (both in small and large-scale virtual environments), searching for certain objects, physically replicating simple virtual sculptures as well as generic volume visualisation tasks (identification, judgement of size, shape and connectivity). Regarding navigation, the authors concluded that during the six considered studies, slightly faster or equal completion times between VR and desktop setup users could be observed. Results for search tasks were found to be more varied. One study reportedly concluded that desktop users were faster, another one stated the opposite. Visualisation tasks such as size estimations were fulfilled with better results by users in a fully immersive virtual environment compared to participants using a head mounted VR display [QTIHM06]. This is an interesting find for this work since this user study was conducted with the help of so-called fish tank VR [WAB93]. This setup, due to the lightweight stereoscopic glasses and almost unrestricted freedom of movement, resembles a very basic variation of the kind of immersive experience that can be achieved with the use of a multi-wall projection installation which I had access to for this work.

Another, in the context of this work, very relevant study described in the paper above is [MJSS02]. This work, aiming at finding measurable advantages of immersive virtual reality (IVR) over conventional display methods within the context of complex 3D geometry, had users closely observe sculptures consisting of more or less randomly bent rods of equal thickness. The users - grouped into IVR and desktop users - were asked to physically replicate the fairly complex object with real, easily deformable leaden rods while looking at the virtual object. The paper describes two studies, in one of wich the IVR setup was a multi-sided projection installation, allowing the users to view the geometric data from effectively every possible position and angle while the desktop user group had to use a joystick or control pad for navigation. It reports that IVR users performed consistently superior regarding both time and error-rate. This suggests that immersive virtual technology might be able to offer a more precise understanding of complex geometric data which is a compelling assumption regarding this work.

With the ambition to develop a predictive model for the positive outcome using a VR setup can have compared to its expenses, Pausch *et al.* [PPW97] found that, while not being being able to help users perform search tasks in virtual space faster than with a desktop setup, users with a head-mounted display were able to complete the tasks with more certainty. They spent significantly less time re-examining areas, which they commonly did with the desktop setup - up to 41% more time. The task users were given in this work was to find a specific letter hidden on the walls of a virtual room which were textured with evenly spread

sets of letters, or confidently declare that the letter wasn't present. The target letter was not actually present in the room in 50% of the tests. Based on the observation that VR users in this study barely spent any time rescanning parts of the virtual room, the authors assumed that a VR setup can have a greatly beneficial impact on systematic search tasks. They based this on the fact that spatial understanding and navigation skills are naturally very well developed parts of human cognition and proposed that the immersive experience did such a sufficiently well job at mimicking a real life environment that these skills could be used to a greater extent than in a desktop setup. Again, this is further reason to be optimistic about finding interesting patterns in what users in a VR setup find to be highly significant regions of 3D objects.

Taking a step back towards the basics of human attention in 3D space, in 1998, Atchley *et al.* [AK] conducted four experiments addressing attention in 3D scenes on a very basic level. Participants were shown simple scenes, each containing sets of six short lines. The scenes were arranged on four different depth planes, one behind the other, and displayed on a stereographic display. The basic task given to participants in all of the experiments was to focus a briefly visible colour singleton on a specific, previously cued depth plane. One of the lines on the indicated plane would change its colour for 100 milliseconds and participants had to correctly say whether it was tilted to the left or the right. To determine the time it takes to shift attention from one plane to another, for some user groups, the colour singleton would appear in a plane other than the previously hinted one. To further track speed and accuracy of attention focus, a distraction element (one additional line changing its colour simultaneously to the target line) was shown to some participants, sometimes in the *target plane*, sometimes in a different one. From their observations, the authors gathered that depth-plane attention can be successfully guided and that distraction elements appearing on the *target plane* significantly interfered with the users' ability to give correct answers, while such elements appearing on other planes had virtually no impact on results.

Taking the effort of tracking attention a step further into virtual reality context, Lee *et al.* [LKC07] accomplished just that on an object basis. They presented a framework capable of such a task, both bottom-up (stimulus-driven) and top-down (goal-directed). Based on pixel-level saliency maps, computed via known methods, similar maps for multiple objects in the scene are generated, allowing predictions on which objects will more likely to be focused first by users. Using the method presented in this paper, object-level saliency maps can be computed in real-time, depending on the users dynamic position and orientation within the scene. The authors exploited knowledge of human cognition which suggests that attention is object-based [ODK99] and, using a monocular eye tracking device, compared the results of estimated object-level saliency maps to the behavior of participants in their study. They dynamically assigned saliency values to each object and, depending on how many of the objects with the highest values (first 1, 2 or 3) were taken into consideration, observed estimation accuracies ranging from about 50 per cent to up to nearly 95 per cent. As one can imagine, accuracy values were the highest when users were given a task, for example finding a certain object within the scene. This shows that attention in virtual space can be tracked and accurately predicted on object level.

In 2012, Y. Kim *et al.* [CSPF12] conducted a study that relied on an eye-tracking system to compare previously computed saliency maps to eye movements of participants, aiming at comparing the computer generated saliency map with user input describing regional importance of 3D objects. The authors based the computation of mesh saliency maps on the work by Lee *et al.* [LVJ05]. They then introduced a normalized, chance-adjusted saliency in

order to evaluate the correlation between point-wise mesh saliency values and the users' eye fixations on 2D rendered images of 3D objects, each visible for five seconds. They concluded that the computational model of mesh saliency has better correlation with human eye fixation points than both a randomly model as well as another curvature-based model during this specific timespan. This suggests a basic correlation between mesh saliency maps and input gathered from tracking human vision. However, since 2D images were used during this study and there was no form of interaction whatsoever, lots of ways to validate the relevance of mesh saliency are still untouched.

3 Concept

One of the main goals of this work is to describe and implement a framework that allows for quantitive statements about differences between user selected regions of importance and computer generated mesh saliency maps to be made. It also includes such a comparison on a basic, conceptual level. Based on these abstract tasks given, the workload and how it is approached, can be described by the following milestone-like, high level requirements.

1. implement a selection application for an immersive virtual reality environment
   a) spatial indexing of 3D data
   b) selection process
2. conduct a user study to acquire data for a comparison
3. conceptualise a measure of differences between the data sets

In this section, I will go through these requirements and describe in more detail what specific challenges they entailed and how I went about implementing solutions to them. I will also describe the underlaying concepts I chose to use and how I amended them to better fit this work where needed.

3.1 Implementing a selection application for an immersive VR environment

One of the main challenges of this work is developing a piece of software that allows users in an immersive VR environment to select vertices of 3D objects. Designing, implementing and adjusting this software to being executable as a multi-threaded client-server application is another challenging aspect of this work. For the rest of this work, this piece of software will be refferend to as *selection application*. For details on its implementation, see section 4.

3.1.1 Spatial indexing of 3D data

3D objects and data in general are read as lists of coordinates by computers. Common 3D file formats such as .OBJ, .FBX and .STL contain the same data in similar structures, using multiple lists of different kinds of geometric information. While they these formats vary in the range of information they can hold, they all represent at least the following types of essential data. *Vertices*, or a three tuple of float values describing x, y and zu coordinates and *faces*, or basic triangles consisting of three vertices. Other optinal information that can be represented include vertex normals, texture coordinates and more complex features such as assigned materials, animations and armature objects. Figure 3.1 shows an excerpt of a 3D file in .OBJ format, viewed in a simple text editor (gedit).

```
# Blender v2.76 (sub 0) OBJ File: 'teapot.blend'
# www.blender.org
mtllib teapot.mtl
o teapot
v 24.623280 49.265629 -15.350850
v 22.454641 51.212872 -16.810532
v 24.681118 52.984787 -16.030649
v 26.758923 49.646870 -14.111491
# [...]

vt -0.576792 1.397005
vt -0.565282 1.372590
vt -0.546440 1.389664
vt -0.570761 1.415686
# [...]

f 1/1/1 2/2/2 3/3/3
f 4/4/4 3/3/3 5/5/5
f 6/6/6 4/4/4 5/5/5
f 4/4/4 7/7/7 8/8/8
```

Figure 3.1: Notation in .OBJ format

All these types of information share the following properties which were of relevance for this work. They are contiguous lists of lines, each line representing one instance of the data type indicated in the beginning of the line. Figure 3.1 depicts an excerpt of an .OBJ file ocntaining vertices (lines starting with v), vertex texture coordinates (lines starting wit vt) and faces (lines starting with f).

These lists are not ordered. Depending on the modelling process, the vertices, faces and all the other attributes can be presented in an completely arbitrary order which hold very little information about the actual, geometric features of the object. The spatial position of any given vertex in relation to the entire object can not be retrieved from this type of notation. This created the demand for spatially indexing of 3D data in the scope this work.

I decided to implement the concept of Octrees [Octb] because of its convenient characteristics as well as prior, personal working experience with *quadtrees*. In an octree, the geometric size of the smallest possible leaf node can be determined in direct relation to the object to be indexed and, at any level, all leafs and nodes will be of the same size. This is highly useful in radius-based proximity-requests for multipl reasons as described in section 4.3.2.1.

One of the most common queries performed on 3D data are proximity queries. This means that vertices located within a given radius around an input query-coordinate are to be retrieved. The trivial and obviously highly inefficient solution to such a problem would be to iterate through the entire list of vertices and their coordinates and check for those who fulfill this query condition. This is where spatial indexing structures such ad octree come into play. The concept of octrees is highly recursive and can realize all the most common types of queries (including search queries) in logarithmic time.

An octree is a set of nodes that store references to one another. The highest-level node is called the root node, bottom level nodes are usually referred to as leaf nodes or leafs. Every node that is not a leaf node has eight children-nodes which, in a spatial sense, make up the entire space of their parent node. Such non-leaf nodes can also be described as roots of subtrees.

Figure 3.2 taken from [octa] shows a simple octree structure indexing five vertices, both

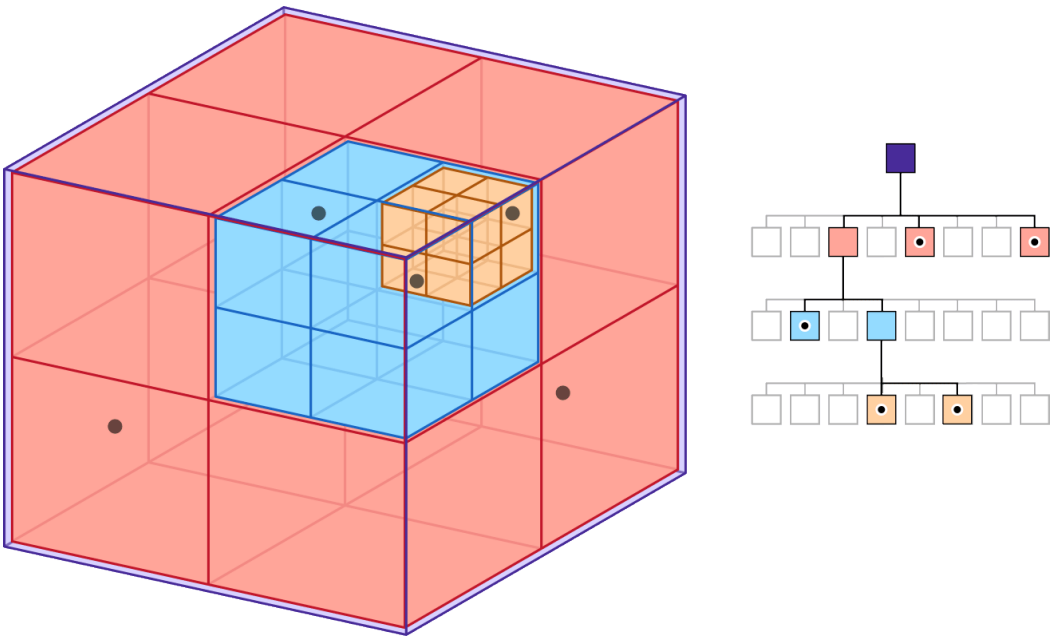

Figure 3.2: An octree structure indexing five vertices. Left: 3D view, right: tree structure; 2017, https://developer.apple.com/documentation/

in 3D and tree view. Note how the root node on level 0, shaded purple on the right, is represented as a large purple cube on the left, encompassing the entire set of vertices. The figure visualizes how an non-balanced octree structure that allows one vertex per leaf node at most would look like. On levels 1, 2 and 3, there are eight nodes each. On level 1 (shaded red), there are two leaf nodes that hold one vertex each. The eighth subtree node contains three vertices and is thus split into eight subtree nodes (shaded blue), each on level 2. Out of these level 2 subtree nodes, one is split again into eight level 3 leaf nodes, two of which store one vertex each.

The most important parameter for an octree is the maximum allowed number of vertices per leaf node. Once the dimensions in x-,y - and z-direction of the 3D object to be indexed have been determined, the root node of the octree will store every vertex until said maximum allowed number of vertices is reached. Up until that point, the root node is a leaf. It is from then on no longer a root and creates eight new nodes (its child nodes) and store references to them. This process is repeated recursively for every new node until every leaf node holds less than the maximum allowed number of vertices per leaf node. It is worth mentioning that in many implementations of the octree concept, every leaf node needs to be at the same level. For this work, the non-balanced version of an octree where that is not the case is used. The reason for this is that in 3D objects, the data is not evenly and very sparsely distributed inside the entire space defined by its bounding box that. The vertices make up the surface of the model, inside and outside of it, there is no spatial information to be indexed at all. Queries to an octree structure can be processed with logarithmic costs which, in combination with sparsely distributed data, allows for fluent, real-time interaction with objects up until a certain magnitude of vertices total. By carefully choosing suitable values for the maximum allowed number of vertices per leaf node and maximum allowed split depth, acces times can be further optimsed.

3.1.2 Selection process

To gather data to compare with computer generated mesh saliency maps, tracking user input is needed. In an appropriate immersive virtual reality setup, 3D objects get loaded and spatially indexed using octrees as described in the section above. Now, the selection function is comes into play. Using a hand-held input device, users can interact with the scene in two essential ways - navigation and performing arbitrary operations mapped to the device's buttons. Each such operation can, among other information, use the current position and rotation of the wand as parameters. Figure 5.1 in section 5.1 shows an example of such an input device.

Figure **??** depicts the *wand* used in the V2C. The yellow joystick in the middle of its top side, in combination with the current rotation of the device, can be used to navigate the user's view onto the scene. Furthermore, the two essential functions of the application, selecting and deselecting vertices, are mapped to two buttons of the device. From a user perspective, both these functions are executed in a radius around the tracked position of the device. Put simply, vertices that are rendered near the device at the time the user presses buttons, become either part of the current selection or get removed from it. In accordance with the simplistic structure of the user study, the interaction is designed to be as little complex as possible, hence only these two buttons having actual, distinct purposes.

With this setup in place, everything needed to implement the user selection part of the application is at hand. In order to provide as much visual accuracy as possible, a bright green diamond shaped object at the tracked position of the *wand* is projected into the scene in addition to the loaded model. To additionally provide clear visual feedback, vertices selected are painted in bright red, clearly seperating them from non-selected parts of the object, displayed in plain grey. For more details on the visual presentation of the application, see section 3.2.

Whenever a user presses the button the *add* function is mapped to, a proximity query is performed on the octree structure indexing the object with its current position as the input. The position is passed as a three-dimensional coordiante. For a detailed description of how such a query is handled in this application, see section 4.3.2.1.

In addition to these input coordinates, the following two parameters are passed to calls to this function in the context of a user adding or removing vertices to the current selection: The pre-computed selection radius and a reference to a temporary set vertices, holding unique vertex IDs. After a query is terminated, the set of vertices found will be added to this temporary set. Due to the design of the synchronisation routine between server and client threads of this application, this set will be emptied before the result of each such query is returend and written to it. If the *add* button is pressed, the content of the intermdiate selection will be added to another set holding the list of currently selected vertices total. If the *remove* button is pressed, each of the temporarily selected vertices is looked up in this second set and, in case it is found, removed from it. This approach prevents the total set of selected vertices to be sent from the server to all client threads at every frame during runtime. Only newly selected vertices (or those that are to be removed from the current selection) are sent across the application. The time for each action (in seconds since the application was started) is logged as well. For more details on the implementation of the *selection application*, see chapter 4.

3.2 Conduct user study with the selection application

Another essential part of this work is conducting a user study where all of the aspects described in this chapter so far are be put to use. The goal is to collect data on what parts of 3D objects users find visually interesting or important. Users put on the stereoscopic, trackable glasses, step inside a suitable immersive VR environment, for example a multi-sided projection installation, and are asked to mark regions of the object currently displayed that they consider to be intersting or important.

After taking a few minutes to get familiar with navigating and interaction within the projection installation, as well as selecting and deselecting vertices with the hand-held input device, the main part of the user study begins. For a detailed documentation of how the user study was designed and executed, see chapter 5.

Selections of the users are gathered to compute user-weighted importance maps for the displayed 3D objects. These are compared to previously computed mesh saliency maps. For the rest of this work, they will be referred to as *user saliency* values. For the results and discussion of this comparison, see chapter

In this selection application, selection and deselection operations are executed with the current tracked position of the hand-held device as input coordinates. To give users a clear, live feedback on that tracked position, for every frame, a ten-sided, diamond-shaped object is also rendered in the scene on that very position. The object is shaded in a bright green, to be easily distinguishable within the 3D scene. However, it did not offer a visual representation of the pre-computed selection radius. This radius r_{max} is defined as d_{min} / $2^{l_{max}}$ * 0.95, in other words 95% the size of the smallest possible leaf node, so it is slightly different for every object loaded in the application. The size of the ten-sided object representing the tracked position of the device within the scene is chosen to stay the same for each object and not to adapt to their sizes. This is done with the intention of providing more consistency within the scene, independent of the currently loaded object. This design decision goes in accordance with the minimalistic general visual presentation of the application, as described in the following brief summary.

- background-color of the application: (almost) black
- shader used for the loaded object: light grey, no textures, normals used
- shader used for selected vertices: bright red
- shader used for diamond-shaped object indicating the tracked position of the *wand*: bright green

Two steps are taken to eliminate the possibility of user selections being influenced by them first having to adapt to and getting familiar with the selection application. First, Every user is given as much time as desired to get familiar with the navigation and selection workflow, accompanied by hints and instructions. Selections made here are not considered for the results of the userstudy. After that - when the user feels ready - the three objects are presented in a randomly chosen order for five minutes each. The users are given the tasks stated at the beginning of this section and are free to use as much of the five minutes as they want to perform vertex selection and deselection operations.

The five minute time limit is an absolute upper limit for each selection process. Users are free to end selection prematurely whenever they state they are satisfied with the selection

as it is. See figure 5.2 in section 5.1 for a basic depiction of what the the interaction looks like in the five-sided projection installation used for this work.

Besides the five minute *tutorial phase* - the time users can take to get familiar with the application - and the actual tasks to be fulfilled (as described at the beginning of this section), a few more instructions are given and requests asked for. Users are asked to consider symmetry in their selection, meaning that, in cases where it is possible, parts of objects users select on one side of the object are also selected on the opposite side. Users are told that there is no *correct* way to fulfill the selection task. They are encouraged to select whatever tehy deem relevant according to the tasks based on their personal judgement and subjective interpretation of *interesting*. The users are left alone during the selection process. After four of the maximum five minutes have passed, users are told that they have one minute remaining. After five minutes, the current selection process is stopped, the result written to a logfile and the next object gets loaded into the VR environment.

3.3 Measure of differences

Since large portions of the workload of this work is the implementation of the selection application as well as conducting the user study, the comparison of user generated and pre-computed mesh saliency maps is designed to be quick and easy. The main goal is to have a ratio that expresses whether there are major differences to be further examined or not.

This chapter describes a basic way to compute a *difference ratio*, normalised to a decimal value between 0 and 1, describing how much user saliency maps differ from mesh saliency maps, is conceptualised. It is based on a simple vertex-wise comparison of saliency-values and makes use of the proximity queries provided by the octree structure, which is also the basis for the selection application itself as described in section 4.3.1.

In chapter I consider both *raw* and *weighted difference* values, for a detailed explanation of these terms, see the following subsections. For a brief summery and step-wise explanation, see subsection 3.3.4.

3.3.1 Terms and abbreviations

The following terms, symbols and abbreviations will be used in the subsequent sections to explain the measure of difference used in this work.

unweighted difference ratio - a float number, normalised to a value between 0.0 and 1.0 indicating how much the generated *user saliency map* differs from the pre-computed *mesh saliency map*, based on unweighted (raw) difference values

weighted difference ratio - a float number, normalised to a value between 0.0 and 1.0 indicating how much the generated *user saliency map* differs from the pre-computed *mesh saliency map*, based on weighted difference values

v_i - the currently considered vertex $i \in$ set of all vertices V belonging to an object

$S_M(v_i)$ - the computed mesh saliency value for v_i as described in [LVJ05]

$S_U(v_i)$ - the user generated saliency value for v_i, describing its *perceived importance* value based on how many users selected it in relation to the most-selected vertices.

vertId	timestamp
27	64
59	62
63	62
64	62
65	62
67	62
68	62
69	62

Table 3.1: Sample content of a user selection log file

$\Delta_{raw}(v_i)$ - the unweighted (raw) difference between the user and mesh saliency values $S_U(v_i)$ - $S_M(vi_i)$. The absolute value of the result is normalised to a value between 0.0 and 1.0

$\Delta_{weighted}(v_i)$ - the weighted difference between the user and mesh saliency values $S_U(v_i)$ - $S_M(vi_i)$. The absolute value of the result is normalised to a value between 0.0 and 1.0

$V_{proximty}(v_i)$ - a *proximity set* of vertices v_1, ... v_j that are located within radius r around the coordinates of given vertex v_i

ω - a threshold value for *mesh saliency* values $S_M(v_1)$, ... $S_M(v_j)$ for all vertices v_1, ... $v_j \in$ a *proximity set* $V_{proximty}(v_i)$ of vi. 0.8 is used in this work.

3.3.2 unweighted difference ratio

As stated above, $\Delta_{raw}(v_i)$, or the unweighted (raw) difference between user generated and computed mesh saliency value can easily be obtained by taking the absolute value of $S_M(v_i) - S_U(v_i)$. For each of the objects used in the user study a mesh saliency map using `climberpi's` implementation of *mesh saliency* [clm] is computed, providing $S_M(v_i)$ for every vertex v in V_{bunny}, V_{cow} and V_{P-51} where V_l is the set of all vertices that make up the object with label l.

Values $S_U(v_i)$ are computed via a simple, average-based comparison of how many users selected each vertex. As mentioned in 4.3.2, user selections are written to simple structured text files containing one unique vertex ID and the time they are added to or removed from the selection (in seconds since the application was started) in every line, seperated by a tab symbol as depicted in table 3.1.

In the simple evaluation process used for this work, before each logfile is read, a container object *user selections* is created with size equivalent to the number of vertices of the currently considered object. After the last log file is processed, *user selections* contains a mapping for each vertex v_i, describing how many users selected it at offset i, which equals the vertex ID. In other words, let $userselections[i] = t$ for each vertex v_i of an object where t is the amount of users that selected v_i.

Next, the maximum number of times one or multiple vertices were selected t_{max} is computed. Based on this maximum value, *user saliency* values are $S_U(v_i)$ generated as follows.

$S_U(v_i) = userselections[i]/t_{max}$

User saliency values for each of most often selected vertices - those selected by t_{max} users - will be 1.0, and 0.0 for those which were not selected by a single user. Every other *user saliency* value will be somewhere in that range, describing what one could informally call the *popularity* of the vertices.

With both *mesh* and *mesh saliency* values for each vertex v_i, we can use the absolute value of their difference as the *unweighted (raw) difference*. The final step to generate the overall *unweighted difference ratio* is a simple addition of $S_U(v_i)$ values for every $v_i \in V$, where V is the total number of one object and dividing the result by V. This ratio gives a quick, first impression of how much the average user selection differs from a *mesh saliency map* computed according to [LVJ05].

3.3.3 weighted difference ratio

The unweighted *difference ratio*, being very basic and average based, lacks expressiveness. During research and preparation for this work, it became clear that *mesh saliency maps* offer a solid basis for prediction models of what parts of 3D objects human attention tends to gravitate towards. By further developing and refining the concept, these predictions became precice and reliable, see chapter [**?**]. I based my idea for the *weighted difference ratio* on the assumption that differences in saliency values for some vertices are more interesting than others.

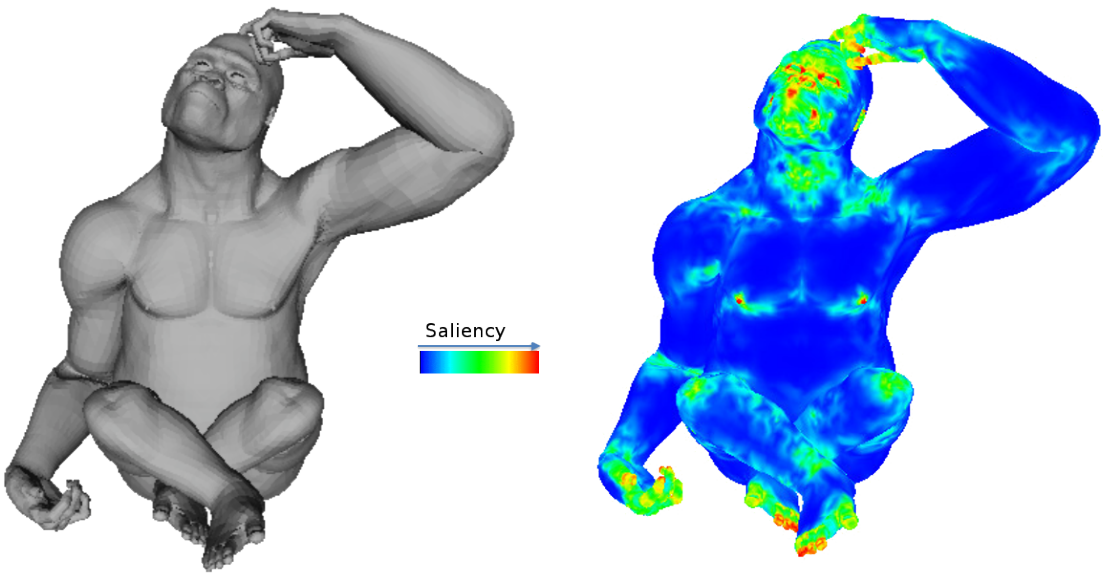

Figure 3.3: A multi-scale mesh saliency map for an object including color scale. Published by Nouri *et al.* [NCL15]

Consider figure **??**. It depicts an object and its multi-scale mesh saliency map as well as a scale for reference. The scale describes how yellow and red colored vertices indicate high mesh saliency values for those vertices while green and blue colored ones suggest lower values. Note that this map is not computed via the *standard* mesh saliency model but an enhanced, multi-scale based one. The figure is suitable to explain the idea behind which

point-wise difference value in user and mesh saliency values are *weighted* in this work.

Based on the assumption that a difference betwen *mesh* and *user saliency* values of a vertex is more interesting if it is surrounded by vertices which have high *mesh saliency* values than it would be if all the surrounding vertices are deemed not highly interesting by *mesh saliency*, such differences get weighted. Consider a vertex v_i in the following cases.

case 1: v_i has a *user saliency* value of 0.2 or lower and vertices in $V_{proximty}(v_i)$ have an average *mesh saliency* value of ω or higher.

case 2: v_i has a *user saliency* value of 0.2 or lower and vertices in $V_{proximty}(v_i)$ have an average *mesh saliency* lower than ω

case 3: v_i has a *user saliency* value of 0.8 or higher and vertices in $V_{proximty}(v_i)$ have an average *mesh saliency* value of ω or higher

case 4: v_i has a *user saliency* value of 0.8 or lower and vertices in $V_{proximty}(v_i)$ have an average *mesh saliency* lower than ω

This simple evaluation process is based on the assumption that realistic, reliable results are achieved by the model of *mesh saliency*. So it focuses on finding deviations within parts and regions of objects that have high average mesh saliency values. Thus, only in cases 1 and 3, vertex-wise differences get weighted. High vertex-wise differences in regions with a low overall mesh saliency value still are considered for the final *weighted difference ratio*, but not as much as those at hand in cases 1 and 3.

Regarding how the actual weighting is done, a simple function that increases the vertex-wise *raw difference* values, normalising them between 0.0 and 1.0 without getting greater than 1.0 is used. An altered, restricted function describing a circle that has its centre at $x = 1.0, y = 0.0$ and radius $r = 1.0$ is suitable for this demand. Figure 3.4 depicts this *weighting function*. It shows how vertex-wise differences between *mesh saliency* and *user saliency* values are used for computation of the final, overall difference ratios - both *unweighted* (black line) and *weighted* (blue curve).

Consider figure 3.4. The straight, black line, being a steady identity function of float values, shows normalised *unweighted* vertex-wise difference values. For such values, both in regards to *unweighted* and *weighted* overall *difference ratios*, their unaltered, absolute value $\Delta_{raw}(v_i)$ is used for computation. However, should vertices in $V_{proximty}(v_i)$ have an average *mesh saliency* value of ω or greater, the difference between *mesh saliency* and *user saliency* values at v_i is weighted according to the blue curve function shown in the figure.

In conclusion, if v_i is surrounded by vertices whose average of computed *mesh saliency* values surpasses a certain threshold ω, it is assumed that v_i is located in a greater, contiguous patch of the object with high importance as determined by the processing according to *mesh saliency*. Hence, $\Delta_{raw}(v_i)$ is used for generating the overall *unweighted difference ratio* and $\Delta_{weighted}(v_i)$ is used used for generating the overall *weighted difference ratio*.

3.3.4 step-wise summery

This subsection provides a brief summary of the steps to computing overall *unweighted* and *weighted difference ratios* described in greater detail in the precedig subsections. It provides more insight into how the data is organised and prepared for evaluation. It describes the workflow for a single 3D object.

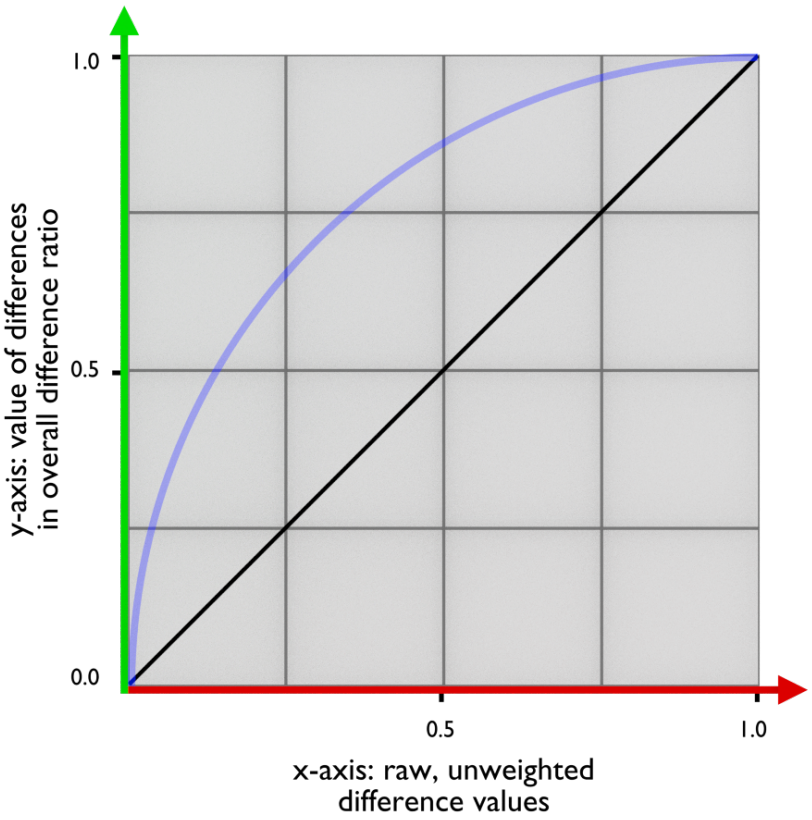

Figure 3.4: Graphic representation of the weighting function in relation to the *unweighted* difference values (straight, black line)

1. Compute the *mesh saliency* map for a given object using using `climberpi's` implementation [clm] and store the result in an external textfile `computed_saliency` for later use.

2. Read n textfiles `u_00`, ... `u_n`, each containing one user selection, compute the *user saliency* map based on how many times each vertex was selected in relation to the most-selected vertices and store the result in an external textfile `user_saliency` for later use.

3. Read `computed_saliency` and `user_saliency`, store the contents in two containers of type `std::vector` called *computed_saliency_values* and *user_saliency_values*.

4. calculate vertex-wise saliency differences and store them in an a `std::vector` *differences_unweighted*.

5. Iterate through the list of vertices of the currently considered object. For every such vertex v_i, find vertices v_1, ... $v_j \in V_{proximty}(v_i)$ via a range-query to the `ocTree` instance indexing the object (see section 4.3.2.1).

6. For each vertex v_i, look up *mesh saliency* values for vertices v_1, ... v_j in the *mesh saliency* map. Compute the average of those values.

   a) if average is ω or greater, compute $\Delta_{weighted}(v_i)$ and storte it in a `std::vector differences_weighted`.
   b) if average is less than ω, keep $\Delta_{raw}(v_i)$ as it is in `differences_unweighted`.

7. Iterate through `differences_unweighted`, calculate the average and return it as the overall *unweighted difference ratio*.

8. Iterate through `differences_weighted`, calculate the average and return it as the overall *weighted difference ratio*

4 Selection Application

This chapter describes the of the selection application used for this work. After a rundown of third party requirements and a summary of relevant C++ classes, the description is further subdivided according to its abstract, key requirements. The goal of this chapter is to describe how the application, especially the Octree[Octb], is designed and implemented. Accordingly, key lines of source code as well as plenty of explanatory comments are provided.

4.1 Additional third party libraries

To ensure a scalable, platform independent implementation of the application, the following third party libraries, frameworks and APIs are used.

4.1.1 OpenGL

The Open Graphics Library OpenGL [Ope] is a powerful industry standard API for rendering 2D and 3D graphics, independently of programming language and operating system. One of its most outstanding features is its ability to directly perform operations on the graphics processing unit of a computer, allowing fast, hardware-accelerated display of graphic elements. For this work, OpenGL is used for displaying the 3D objects both in the user study and throughout development of the selection application. The task of displaying rendered images across multiple projection surfaces on a 360°panorama view is handled by software developed at the Zentrum für Virtuelle Realität und Visualisierung (V2C) of the Leibniz-Rechenzentrum [v2c].

OpenGL is based on the following basic structures and concepts:

Vertex Buffer Objects (VBOs) contain actual vertex data. Coordinates, normal and color information, texture mapping and any other kind of data that is desired can be saved in these kinds of objects. They are designed as buffer objects to be stored directly within the memory of the video card, ensuring extremely fast access times.

Vertex Array Objects (VAOs) are objects which can contain one or more Vertex Buffer Objects and store information for complete, rendered objects. In other words, VAOs store descriptions of vertex data stored in VBOs. For example, the number of coordinates the vertices are made of, in which order etc. From a performance aware point of view, they are a great improvement over older, deprectecated concepts in OpenGL since multiple calls to bind and upload distinct sets of data belonging to the same object to the graphics processing unit can be bundled in one call to a VAO.

Vertex Shaders are small pieces of C-like code, executed by the graphics card, which can perform extremely fast, basic operations on every vertex of a vertex data input stream. Vertex shaders are one of two mandatory types of shaders. As soon as an openGL `draw` command is called, they process every incoming vertex and output two things for each

one: Its position (untouched or altered in any way desired) within the output window on screen, and a set of arbitrary defined custom data such as normal information, color and texture mapping. These attributes are calculated on a per-vertex basis.

Fragment Shaders complete the absolute minimum for an openGL-based rendering pipeline to be able to display anything, together with vertex shaders. They are executed per pixel and, therefore, considerably more often than vertex shaders. Their purpose is to interpolate, for each pixel in between vertices, its RGB value, based on the per-pixel values calculated by the vertex shader. The term *fragment* is often informally described as a *potential pixel*, as the final RGB value of a pixel can still be changed after it is processed by the fragment shader.

4.1.2 GLUT

As stated on its official webpage [GLU], GLUT is an official OpenGL Utility Toolkit which provides, among other features, support for multiple windows, control of such windows and handling input from devices such as keyboards and mouses. It is commonly used to achieve interactive windows with cross-platform compatibility displaying rendered images produced by OpenGL. Handling input via the hand-held controller in the user study is achieved with the help of GLUT during this work.

4.1.3 GLEW

The OpenGL Extension Wrangler Library (GLEW)[GLE] is a cross-platform extension loading library, specifically designed to be used by C/C++ applications. It provides run-time mechanisms for OpenGL extensions supported on the target platform, allowing to faster query and load those extensions.

4.1.4 ASSIMP

Available across multiple operating systems including Android and iOS, The Open Asset Import Library [ASP], is a powerful open source library that offers import, export and post-processing functions for most commonly used 3D data formats. In this work, its easy to use import function for OBJ files is used for loading the 3D objects to be displayed in the user study. ASSIMP implements a set of hierarchically organised data structures or so-called nodes. Two of the most relevant ones for this work are briefly described below.

aiScene is the root of all the imported data returned from a successful call to one of ASSIMP's import functions. Global information such as the direction of the coordinate system, its origin location as well as references to all the other data in the scene are stored here.

aiMeshes represent imported meshes within the scene. Each aiMesh has its own local coordinate system with an origin point and all the vertices belonging to it. Multiple sets of data describing one imported mesh can be stored in these mesh objects but sets of vertices and faces are always guaranteed to be present, thus enabling a basic graphic representation of the mesh.

4.2 Relevant class files

This section covers all the relevant C++ classes used to implement the selection application. Note that these descriptions only cover the general structure and purpose of these classes within the context of the applicatoin. For a more detailed description of the most crucial functions, see section 4.3.

4.2.1 Object

The object class is used to represent a 3D object within the project. It uses import functions from ASSIMP to load a file via a given source path. An object can contain multiple mesh objects, segmentation happens automatically based on a threshold number of vertices that can be stored in one mesh. This class is used to work with potentially very large 3D files in a uniform and quick way, mostly by implementing wrapper functions that have each mesh object associated to an object call their upload and draw functions, their destructors etc.

4.2.2 Mesh

One object can consist of multiple meshes. These meshes are coherent with instances of aiMesh (see subsection 4.1.4) and all the important attributes such as vertices, faces, normals, texture coordinates and IDs are stored here. OpenGL functions such as uploading vertex buffer data to the graphics proccessing unit and drawing are implemented here. Some of the application's most crucial functionalities such as adding to and removing vertices from the global selection of vertices to be highlighted are implemented in this class, see 4.3.

4.2.3 Octree

Spatial indexing of loaded objects in the application is entirely handled in this class. This has been one of the most labour-intensive parts of the application because formal guides to implementing it, independent of coding language, are scarce and working with the data that is stored in the object and mesh classes above required an extensive amount of customisation.

4.3 Key Features

This section describes the following features and functionalities which are most crucial to the selection application.

- Spatial indexing via Octree
- User selection
- Tracking selection
- Testing setup

Id[O_l, O_l+1, O_l+2]	X min	X max	Y min	Y max	Z min	Z max
000	p.X min	p.X mean	p.Y mean	p.Y max	p.Z mean	p.Z max
001	p.X mean	p.X max	p.Y mean	p.Y max	p.Z mean	p.Z max
010	p.X min	p.X mean	p.Y min	p.Y mean	p.Z mean	p.Z max
011	p.X mean	p.X max	p.Y min	p.Y mean	p.Z mean	p.Z max
100	p.X min	p.X mean	p.Y mean	p.Y max	p.Z min	p.Z mean
101	p.X mean	p.X max	p.Y mean	p.Y max	p.Z min	p.Z mean
110	p.X min	p.X mean	p.Y min	p.Y mean	p.Z min	p.Z mean
111	p.X mean	p.X max	p.Y min	p.Y mean	p.Z min	p.Z mean

Table 4.1: Child node bounding values

4.3.1 Spatial indexing via Octree

As mentioned above, the `ocTree` class handles spatial indexing and, therefor, provides quick access to every vertex of an imported 3D object via a set of integer-like (`size_t`) indices. The general approach to this implementation of the concept of Octree is designed with a heavy emphasis on its recursive features. Instances of it can be created from everywhere in the application by the call of its public root constructor function. Nodes can be leafs or not, which is indicated by a boolean flag for every instance of an `ocTree` object. Leaf nodes do not have subtree-nodes that refer to them as parents, they solely store vertices within their bounds. Non-leaf nodes have eight children nodes, in other words, eight more `ocTree` objects which refer to them as their parent node.

For better understanding during development and clearer, human-readable log messages, unique binary identifiers were implemented with care. Each node of the tree has a private `std::vector` which serves as a unique combination of boolean values describing its identifier. It can be used for directly accessing any desired `ocTree` (subtree) object within a tree through its root node.

Starting from the root node (level $l = 0$), such an identifier *Id* with a length of n boolean values can be used for locating the respective node within 3D space by considering three of its consecutive values at a time. At any level l, those values of `Id` can be found at positions O_l, O_l*+1* and O_l*+2* within it, where O_l is the *level offset* $l * 3 - 3$. If O_l*+2* equals its length `n`, the search ends and the resulting node can be queried for the vertices within its bounds. Every non-leaf node has eight subtree nodes on level *l+1* where l is the level of that node. Their bounds can be derived directly from the parent nodes' maximum and minimum values as table 4.1 depicts. The suffix p for new values refers to the parent node, `[n]` is the `n`th element of identifer `ID`. O_l describes the *level offset* $l * 3 - 3$. Note that $O_l + 3$ determines the child node's minimum and maximum values in x, $O_l + 2$ in y and O_l in z dimension.

The most important functions of the `ocTree` class, as implemented in this work, are described below.

4.3.1.1 Root constructor `ocTree()`

This public constructor creates a new instance of the class `ocTree`. Parameters required are 1. a sequence container, such as an array (`std::vector` is used in this work) holding the `mesh` objects to be spatially indexed, 2. an integer determing the maximum amount of vertices that one leaf node can store, and 3. an integer determing the maximum split

depth, in other words the maximum depth of the tree. As an optional fourth parameter, a boolean flag can be passed as well. Its default value is set to be `false`, if it is set to `true`, additional information regarding the recursive construction of the tree, including identifiers, level, dimensions and number of vertices held by each subtree, are printed to the console via `std::cout`. During subsequent creation of subtrees, this parameter are used for each new object.

From the main class, for example, creating a new instance of an `ocTree` object is handled by the following short command:

```
1 myOcTree = new ocTree(meshes, 100, 4, true);
```

This creates a spatial indexing structure for the 3D data stored in `meshes` where each node can hold up to 100 vertices and the maximum level of nodes is 4. Additional information is printed to the console because the last parameter is set to `true`.

4.3.1.2 Subtree constructor `ocTree()`

This somewhat more complex, private constructor is used for every `ocTree` object that is not a node. In addition to the parameters that were used for the root node, the following parameters are required: 1. A set of vertices to be searched through for those located within the bounds of this particular subtree (`std::vector` of `glm::vec3` objects is used in this work), 2. An integer determing the level of the parent node, the level of this new subtree is set to that level plus one, 3. An array of nine float values describing its parents dimensions (together wit the *split directions*, this is used to determine the bounds, or dimensions, of this new subtree), 4. A reference to the root node of this subtree, 5. A vector of boolean values describing its parents unique identifier, and 6. A vector of three boolean values describing the *split directions* passed by the parent node. Again, an additional boolean flag determing whether, during the recursive building proccess of the subtree, information is printed to the console or not, is also passed with the value of the respective member variable of the parent node.

The crucial task of setting the right unique identifier of a newly created subtree node is also handled in this constructor. The following code-snippet shows how that is implemented in this work.

```
1  ocTree::ocTree(...) {
2     int identifierSize = m_parentIdentifier.size();
3     int levelOffset = m_level*3-3;
4     std::vector<bool> id(identifierSize);
5
6     for (int i = 0; i < levelOffset; i++) {
7        id[i] = parId[i];
8     }
9
10    id[levelOffset] = splitDirections[0];
11    id[levelOffset+1] = splitDirections[1];
12    id[levelOffset+2] = splitDirections[2];
13
14    for (int j = levelOffset+3; j < identifierSize; j++) {
15       id[j] = false;
16    }
17    m_identifier = id;
18 }
```

After setting up the essential variables, lines 6 - 8 copy the parts of the parent's identifier up until the current level offset. Every new subtree node is a child of a lower-level node and this step implicitly entails that relationship. If an `ocTree` object at a level higher than 0 has child nodes - which makes it the root of a subtree within the Octree - has the unique identifier 010000, the first three values of the identifiers of its eight children nodes is 010.

Lines 10 - 11 set the crucial values at the level offset O_l, $O_l + 1$ and $O_l + 2$ according to the values passed via `splitDirections`. In the context of a parent node calling `split()`, these passed values are eight sets of three boolean values each that can be represented as 000, 001, 010, 011, 100, 101, 110 and 111.

Since every identifier of every node within an Octree, in this implementation, has to have the same length (that is $3 * maximumLevel$ where *maximumLevel* is the maximum allowed level of subtree nodes), lines 14 - 16 take care of assigning `false` as placeholders to every position that is not relevant due to the node's level. In line 17, the final identifier of the current node is set as its private member variable.

4.3.1.3 `getRootDimensions()`

This is called by a newly created root `ocTree`. In this first basic step, all vertices of every passed `mesh` object are iterated through to find maximum values which are used as its general bounds in x, y and z direction. For convenience, a margin value of 0.0001 is added to maximum values and subtracted from minimum values to enable one common rule of unambiguously assigning any given vertex (expecially the ones that are located on boundaries of subtrees) to exactly one subtree - for the root node as well as all subtree nodes.

4.3.1.4 `setDimensions()`

This simple private function takes care of setting up correct minimum, mean and maximum values in x, y and z dimension for a newly created subtree node. The following parameters are required: 1. An array of nine `float` values containing the parent nodes' bounding and mean values, and 2. A reference to an `std::vector` of three boolean values containing the *split directions*.

Based on the *split directions* given via the second parameter, setting up the bounding values for the new subtree is a matter of assigning the correct value of the first parameter. Said second parameter - a vector of `float` values - contains its values in the following order: 0. minimum X, 1. maximum X, 2. mean X, 3. minimum Y, 4. maximum Y, 5. mean Y, 6. minimum Z, 7. maximum Z, 8. mean Z.

To convey the idea of what this function does more clearly, consider figure 4.1. Keeping in mind the order in which the three boolean values that make up *split directions* are handled in the code-snippet below, it is clear that, say for newly created subtree nodes with identifiers 000 and 101 N_0 and N_5, its bounding and mean values are directly derived from the values of their common parent node p as shown in Table 4.2.

child node	minX	maxX	minY	maxY	minZ	maxZ
N_0	p.minX	p.meanX	p.meanY	p.maxY	p.meanZ	p.Z max
N_5	p.meanX	p.maxX	p.meanY	p.maxY	p.minZ	p.meanZ

Table 4.2: bounding values of subtree nodes according to `setDimensions()`

```
1  void ocTree::setDimensions(...) {
2     if (splitDirections.at(0) == false) {
3        m_minZ = parentDimensions[8]; // minZ = p.meanZ
4        m_maxZ = parentDimensions[7]; // maxZ = p.maxZ
5     } else {
6        m_minZ = parentDimensions[6]; // minZ = p.minZ
7        m_maxZ = parentDimensions[8]; // maxZ = p.meanZ
8     }
9
10    if (splitDirections.at(1) == false) {
11       m_minY = parentDimensions[5]; // minY = p.meanY
12       m_maxY = parentDimensions[4]; // maxY = p.maxY
13    } else {
14       m_minY = parentDimensions[3]; // minY = p.minY
15       m_maxY = parentDimensions[5]; // maxY = p.meanY
16    }
17
18    if (splitDirections.at(2) == false) {
19       m_minX = parentDimensions[0]; // minX = p.minX
20       m_maxX = parentDimensions[2]; // maxX = p.meanX
21    } else {
22       m_minX = parentDimensions[2]; // minX = p.meanX
23       m_maxX = parentDimensions[1]; // maxX = p.maxX
24    }
25
26    m_meanZ = (m_minZ+m_maxZ)/2;
27    m_meanY = (m_minY+m_maxY)/2;
28    m_meanX = (m_minX+m_maxX)/2;
29 }
```

4.3.1.5 `split()`

This private function, called by a leaf node in case there are more vertices within its bounds than the maximum number of allowed vertices per leaf, takes care of turning a leaf node into a intermediate node, the root of a subtree in other words. The vertices that are stored by the calling node when this function is called, make up the set of vertices to check by its eight children nodes. These eight subtree `ocTree` objects are created via a call to the private constructor of the `ocTree` class. To illustrate the purpose this function serves, the following simplified C++ code-snippet shows the necessary steps for the creation of two of the eight new subtree nodes that are to be constructed.

```
1  bool ocTree::split() {
2     float parDimensions[9] = {
3        m_minX,  m_maxX,  m_meanX,
4        m_minY,  m_maxY, m_meanY,
5        m_minZ, m_maxZ, m_meanZ
6     };
7
8     std::vector<bool> split0 = {false, false, false};  // 000
9     std::vector<bool> split1 = {false, false, true};   // 001
10    // <repeat for six remaining bool vectors>
11
12    ocTree* chidlLeaf0 = new ocTree(m_verticesInBounds, m_level,
13       m_maxVerticesPerNode, m_maxSplitDepth, parDimensions,
14       m_root, m_identifier, split0, m_debugInfo);
15    ocTree* chidlLeaf1 = new ocTree(m_verticesInBounds, m_level,
16       m_maxVerticesPerNode, m_maxSplitDepth, parDimensions,
17       m_root, m_identifier, split1, m_debugInfo);
18    // <repeat for six remaining ocTree objects
19
20    m_myChildren[0] = chidlLeaf0;
21    m_myChildren[1] = chidlLeaf1;
22    // <assign six remaining children to private array of ocTree nodes>
23
24    m_isLeaf = false;
25 }
```

Lines 2 - 5 define an array of `float` values which contains minimum, mean and maximum values in x, y and z dimension for this node. Depending on the so-called *split directions* given via `split0`, `split1` and so on, the children nodes of this node is able to retrieve their spatial bounding values directly from `parDimensions`.

Lines 8 and 9 show the first two vectors of *split directions*. The remaining six (not shown here) go on to describe values 010, 011, 100, 101, 110 and 111.

Lines 12 - 14 and 15 - 17 show the initialisations of two new `ocTree` objects using the class' private subtree constructor. Note that the two shown calls differ only in one parameter, `split`*n*. This is also true for the remaining six (not shown) objects to be constructed.

Lines 20 and 21 show the assignments of newly created subtree nodes to their place in the current node's private array of pointers to `ocTree` objects - its children nodes. This provides fast and direct access to them for later queries.

4.3.1.6 `getNodeByIdentifierArray()`

This recursive, public function returns a pointer to a leaf node via a boolean input vector representing its identifier. Starting at the root node, it traverses through tree and returns the node with given identifier at the lowest level. The only parameter required is a `std::vector` *Id* of $n = L*3$ boolean values where L is the maximum level of the `ocTree` object calling this function.

As described above, a node at level l that is recursively calling this function, first calculates the current level offset $O_l = l * 3 - 3$ and then considers elements $O_l, O_l + 1$ and $O_l + 2$ of passed identifier vector *Id*. Depending on these values, the matching child node performs the next recursive call of `getNodeByIdentifierArray()`. Every `ocTree` object has a private array as a member variable that holds eight other `ocTree` objects, its children nodes. Given the bounds of a parent node, which are defined as minimum and maximum

values in x, y and z direction, calculating the three mean values in all three dimensions is trivial. The resulting set of minimum, maximum and mean values can be combined in multiple ways and used for minimum and maximum values of all eight subtree nodes, see table 4.1. To geometrically locate the children nodes, one must check the three relevant boolean values and search either before or beyond the median values in x, y and z direction. A `false` value of x means the respective child node lies within the parent node's minimum and mean values in x direction, `true` means it lies within mean and maximum x values. This pattern is inverted in directions y and z. In both cases, `false` means the child node starts at the mean value of the parent node (its minimum value equals the mean value of the parent) and ends at its maximum value, whereas `true` indicates the opposite. Figure 4.1 depicts two exemplary parent nodes at level l (checkered) and their bounds with one of their eight subtree nodes, the ones indicated by $Id[O_l,\ O_l + 1,\ O_l + 2]$, highlighted in red. Again, note that $O_l + 3$ determines the bounds in x, $O_l + 2$ in y and O_l in z direction.

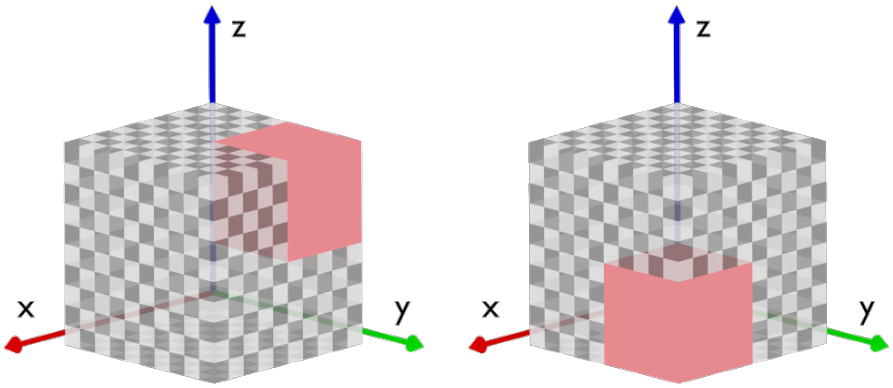

Figure 4.1: Parent node (checkered) highlighted child nodes. Left: 000, right: 101

`getNodeByIdentifierArray()` returns a reference to an `ocTree` object whose identifier matches the series of boolean values passed during the call. It mainly served testing and debugging purposes during development of the application, making sure the spatial indexing structure would be computed correctly for any given set of input 3D data. It is a utility function that provides fast, direct acces to any one desired node within an `ocTree` structure.

4.3.1.7 `buildTreeRecursively()`

A call to this function causes an entire set of given vertices to be indexed and assigned leaf nodes in the tree. Its only parameter required is an indexed list of vertices. A `std::vector<std::pair<size_t, glm::vec3>>`, with the `size_t` parts of the `pairs` providing ordered indexes and the `vec3` parts representing the vertices with three coordinates each, is used in this work.

Most of what this function does, happens in a `for`-loop which iterates thorugh the entirety of the set of passed vertices. Its basic procedure is depicted in the following simplified C++ code-snippet.

```
1  void ocTree:: buildTreeRecursively(...) {
2     for(int t = 0; t != vertices.size(); ++t) {
3        it = &vertices[t];
4        if (it->second.x >= m_minX && it->second.x < m_maxX) {
5           if (it->second.y >= m_minY && it->second.y < m_maxY) {
6              if (it->second.z >= m_minZ && it->second.z < m_maxZ) {
7                 m_verticesInBounds.push_back(*it);
8              }
9           }
10       }
11    }
12
13    if (m_verticesInBounds.size() > m_maxVerticesPerNode) {
14          if (m_level < m_maxSplitDepth) {
15             split();
16          } else {
17             std::cout << "EXCEPTION" << std::endl;
18             return false;
19       }
20    }
21 }
```

Lines 2 - 11 check whether the current vertex lies within the bounds of the calling node. Note that each node calling this function considers every vertex its parent node hold in their member variable `m_verticesInBounds`. In turn, if the calling node is to call `split()` later on, creating eight new subtree nodes, those nodes consider each of the vertices that, via this loop have been determined to be located within their parent node. This stems from the trivial observation that a subtree can only contain vertices that are also contained by their parent node. So the root node of an `ocTree` always checks every single vertex of the loaded 3D object. However, the higher the level of its subtree nodes, the fewer vertices have to be checked by those nodes.

Line 13 shows the crucial check wheter the number of vertices within the bounds of the calling node exceeds the maximum allowed number of vertices per node. If this is the case and the maximum allowed level (`m_maxSplitDepth`) of subtrees is not reached yet (line 13), a call of `split()` by this node follows.

Figure 4.2 depicts a simple `ocTree` structure that could result from indexing a small set of 3D data. This particular tree has a root node at level zero, represented as a basic cube in the upper left part of the figure. As the number of vertices within the bounds of the root exceeds the maximum numbers of vertices a node may hold in this particular tree, the root calls `split()` so that eight new subtrees are created and the root switches its boolean flag `isLeaf` to `false`, indicating that it is no longer a leaf node but the root of an actual subtree within the entire `ocTree`. Given that the maximum level of subtrees visible in the figure is 2, we assume that this is also the maximum allowed level for subtrees. This would mean that the identifier `vectors` of every node within this tree has a length of $3 * 2$. The level 1 subtrees 000100 and 000111, as shown in the figure, also have more vertices within their bounds than what is the maximum number of vertices per node so they, too, split and creat a total of 16 new child nodes, each at level 2.

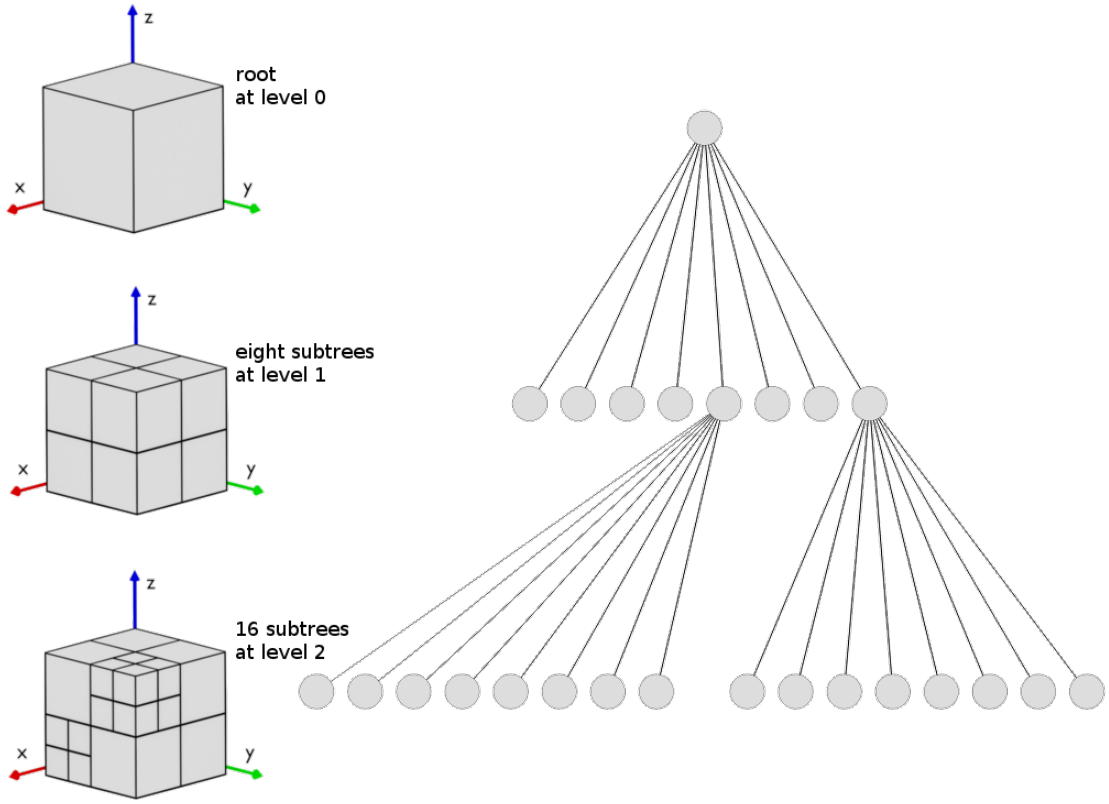

Figure 4.2: depiction of a simple `ocTree`. Left: 3D view, step-wise representation of the splitting process (downwards from the top). right: final 2D representation of the tree's structure

4.3.2 User selection

This section covers the elementary functions that handle the selection of vertices through user input. The two crucial functions explained in the subsequent subsections implement means to add vertices to an initially empty set of vertices and remove them later if desired. Selected vertices are be visually highlighted in the application and, at any time, the current set of selected vertices can be saved to an external text file.

4.3.2.1 `addVerticesToSelectionByCoordinates()`

This private, recursive function adds vertices to a set of selected vertices by reference, based on three-dimensional coordinates and a search radius. The following parameters are required: 1. An input coordinate depicting the point in real space where a user is using the input device (a `glm::vec3` is used in this work), 2. A `float` value describing the radius around thee input coordinate in which vertices are to be added to the selection, and 3. A reference to the set of indexes of vertices (`std::set<size_t>` is used in this work).

An additional boolean flag determing whether, during the recursive search for the correct node, information is printed to the console or not, is passed with a default value of `false`. If set to `true`, it causes information to be displayed. The general course of events within this function proceeds as follows:

1. Check if any of the given input point's coordinates are located outside of the bounds of the root of the `ocTree` indexing the 3D object. Throws exception if this is the case.
2. Check if the current node is a leaf (i.e. has no child nodes). This terminates the recursive search and proceeds with finding vertices around the given input coordinates, as it is safe to assume that the correct node is found.
3. Add all vertices within this node to the set of vertices to be checked.
4. Determine if the radius around the query point crosses the bounds of this node in any direction. If so, add all vertices of the neighbouring node to the set of vertices to be checked.
5. Iterate over the entire set of vertices to be checked to find which ones lie in radius around the query point and add those to the set of selected vertices.
6. If this node is not a leaf node - steps 3, 4 and 5 are skipped in this case - determine the correct child node in which the query point is located and have that node recursively call `addVerticesToSelectionByCoordinates()`.

Consider figure 4.3 for a clear depiction of the process. For the sake of a simplified, clear presentation, an orthographic top view which only considers axes x and y is shown here. Four of an implicitly shown parent node's eight subtree nodes are shown. The figure illustrates, from left to right, three vital steps towards finding vertices that lie within a search radius r around an input query coordinate P_q. Throughout these steps, P_q is depicted as an orange point, r is shaded blue. Vertices within this radius compose the result of the query V_{result} and get added to the set of currently selected vertices. Vertices which are considered to get checked if they lay within r, V_{check}, are highlighted blue, the ones where this is not the case are plain grey.

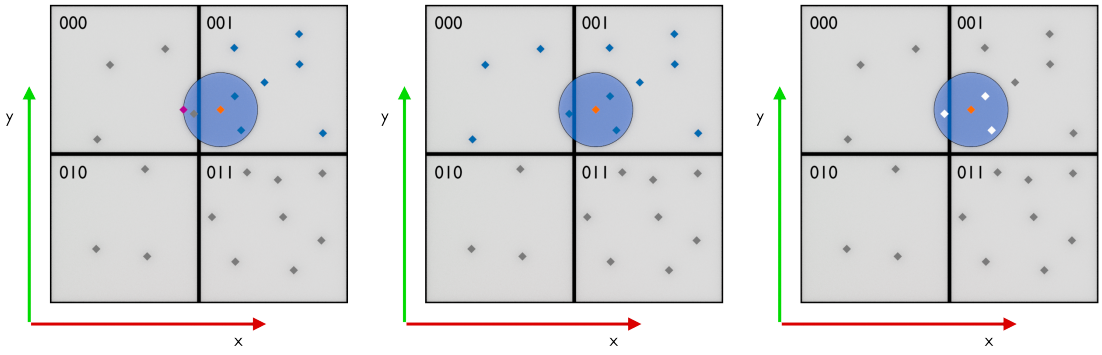

Figure 4.3: Determing vertices to check and final set of selected vertices

The leftmost part of figure 4.3 shows the state after two crucial steps have already been taken: Leaf node N_q, i.e. the node that contains P_q and does not have child nodes (001 in the figure) is found and all the vertices within it added to V_{check} (highlighted blue). Furthermore, since the search radius around P_q crosses the bounds of N_q in negative x direction, an additional call to `addVerticesToSelectionByCoordinates()` with parameters $P_q^{'}$ and radius $r^{'} = 0$ is executed, with $P_q^{'} = [P_q.x - r,\ P_q.y,\ P_q.z]$ (highlighted purple). This

additional query found that $P_q^{'}$ is located in leaf node 000. Thus, as shown in the middle part of the figure, vertices in 000 are added to V_{check}. Note that one vertex that is located within r - but not the bounds of N_q is added to V_{check} this way. The righthand part of the figure shows the result of the original query. After checking all vertices in V_{check} whether they lie within r around P_q, a set of vertices which fulfill this very condition V_{result}, containing three vertices (highlighted white) is returned.

The following code-snippets show how the process described above is implemented in this work.

```
1  ocTree* ocTree::addVerticesToSelectionByCoordinates(...) {
2     if (target.x < m_minX || target.x > m_maxX || target.y < m_minY
3        || target.y > m_maxY || target.z < m_minZ || target.z > m_maxZ) {
4        std::cout << "EXCEPTION" << std::endl;
5        return NULL;
6     }
7
8     if (this->getisLeaf() == true) {
9        result = this;
10       std::vector<std::pair<size_t, glm::vec3> > verticesToCheck;
11       for(size_t t = 0; t < result->m_verticesInBounds.size();++t){
12          verticesToCheck.push_back(result->m_verticesInBounds[t]);
13       }
14       if (radius != 0) {
15          glm::vec3 neighbourCo;
16          if (target.x - radius < m_minX) {
17             neighbourCo = {target.x-radius, target.y, target.z};
18             ocTree * neighbour = m_root->addVerticesToSelectionByCoordinates
19             (neighbourCo,0,intermediateSelection,debugInfo);
20             verticesToCheck.insert
21             (verticesToCheck.end(), neighbour->m_verticesInBounds.begin(),
22             neighbour->m_verticesInBounds.end());
23          }
24          if (target.x + radius > m_maxX) {
25             neighbourCo = {target.x+radius,target.y,target.z};
26             ocTree * neighbour = m_root->addVerticesToSelectionByCoordinates
27             (neighbourCo,0,intermediateSelection,debugInfo);
28             verticesToCheck.insert
29             (verticesToCheck.end(), neighbour->m_verticesInBounds.begin(),
30             neighbour->m_verticesInBounds.end() );
31          }
32          // <repeat for y and z direction>
33       }
34
35       std::vector<std::pair<size_t, glm::vec3>>::const_iterator it;
36       for (it = verticesToCheck.begin(); it!=verticesToCheck.end(); ++it) {
37          if (radiusContainsCoordinates(r, it->second) {
38             intermediateSelection.insert(it->first);
39          }
40       }
41       return result;
42    }
43 }
```

In lines 2 - 6, it is validated whether the input coordinates of P_q given via `target` is located within the bounds of the calling node - that is usually going to be the root node of an `ocTree` structure. If this is not the case, `null` is returned. If line 8 evaluates to `true`,

this means that the current node is a leaf node and since given input coordinates do not lie outside its bounds, this has to be target node of the query N_q.

In line 10 - 13, a new `vector` of `pairs` consisting of `size_t` values and `glm::vec3` vectors, representing the set of vertices to be checked is created. The `size_t` parts of `pairs` in this vector provide unique indexing, ensuring fast and direct access to each element. However, at this stage, this set only contains vertices within initial target node N_q. Without the rest of the code, vertices within one or multiple *neighbour nodes* that contain $P_q^{'}$ vertices which sould get selected, would be ignored.

Line 14 - 36 show two out of six possible types of *neighbour node queries* - both possible ways in which r can cross the bounds of N_q in x direction and how they are handled are illustrated. Note how parameter `radius` is used here (line 14). If it is set to 0, this request (which is also handled by `addVerticesToSelectionByCoordinates` can be classified as a *neighbour query* which means that it cannot possibly envoke further *neighbour queries*. In lines 17 and 28, varying intermediate versions of $P_q^{'}$ are derived by adding and subtracting `radius` to initial x value of P_q after it is found that they are located within a neighbouring node of N_q (lines 16 and 24). After this, assitional queries are processed via `addVerticesToSelectionByCoordinates` with crucial input parameters $P_q^{'}$ and `radius` = 0 (lines 18, 19 and 26, 27). Finally, the vertices within the found *neighbour node* are added to the set of vertices to be checked (lines 20 - 22 and 28 - 30). As hinted by the comment in line 32, this process is repeated for y and z directions.

Before recursively returning the calling node in line 41, the most important part of this function - that is actually determining which vertices are located in raidus r around input coordinates N_q - takes place in lines 36 - 40. Here, we see a `for` loop through V_{check}, the set of vertices that lie within the bounds of N_q as well as those within one or more *neighbour nodes* $N_q^{'}$. The simple utility function `radiusContainsCoordinates` takes care of this. Note that, in case it evaluates to `true`, the `first` element of a pair within `verticesToCheck` (or V_{check}) is added to `intermediateSelection` (or V_{result}). This first element is a simple `size_t` number which serves as a unique identifier for every vertex of the entire loaded object. This approach ensures that almost negligibly small amounts of data are actually transferred within the application - between server to client instances (see 3.1.2)). Depending on what system architecture the application is ran on, these `size_t` values only take 32 or 64 bits of space in memory, whereas `glm::vec3` take up to 12 times as much space.

Note that the code-snippet above only considers cases in which `radius` r around N_q crosses one set of bounds of `ocTree` nodes. However, it is a trivial observation of the conceptual structure of octrees that, depending on combinations of a large `radius`, a low maximum allowed number of vertices per node and a high maximum allowed `level`, critical situations could emerge. Such a situation could take place in a fine-grained octree structure with a large number of high level leaf nodes. For any node N on a level l, the following is true.

$N_s(d) = N_R(d)$ / 2^l where $N_s(d)$ is the size of N in dimension d and $N_R(d)$ is the size of root node N_R in dimension d.

So in this application, after the object is loaded and spatially indexed, we consider the maximum allowed level for leaf nodes l_{max} as well as the bounding values of its root node. The minimum difference between the maximum and mininum values in a dimension indicate the shortest dimension d_{min} of the root node. Based on these values, we can set a maximum allowed value r_{max} for the search radius as follows.

$r_{max} = d_{min}$ / $2^{l_{max}}$ * 0.95. In other words, the maximum allowed search radius is 95% the size of the smallest possible leaf node. Thus, no search query can evoke more than three additional queries for *neighbour nodes*, one in each dimension.

Finally, the following simplified code-snippet shows how, making use of recursion, child nodes perform further calls to `addVerticesToSelectionByCoordinates` until N_q (or $N_q^{'}$) is found, based on the directions in which $P_q^{'}$ crosses the bounds of the calling node.

```
1  ocTree* ocTree::addVerticesToSelectionByCoordinates(...) {
2     // <cont.>
3     if (target.x < m_meanX) {
4        if (target.y < m_meanY) {
5           if (target.z < m_meanZ) {
6              // 110
7              result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
8           } else {
9              // 010
10             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
11          }
12       } else {
13          if (target.z < m_meanZ) {
14             // 100
15             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
16          } else {
17             // 000
18             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
19          }
20       }
21    } else {
22       if (target.y < m_meanY) {
23          if (target.z < m_meanZ) {
24             // 111
25             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
26          } else {
27             // 011
28             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
29          }
30       } else {
31          if (target.z < m_meanZ) {
32             // 101
33             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
34          } else {
35             // 001
36             result = m_myChildren[6]->addVerticesToSelectionByCoordinates();
37          }
38       }
39    }
40 }
```

Finding which child node N_{l+1} is to perform the next recursive call happens in three simple `if - else` statements, the first one consisting of blocks of lines 3 - 20 and 21 - 39. In case the first `if` in line 3 evaluates to `true`, P_q lies between the calling subtree node's minimum and mean values in x direction, thus the relevant child node's identifier must be `false` (0) at its *level offset* O_{l+1}+2. If the statement in line 4 is `true`, the second O_{l+1}+1 can only be `true` (1). Whether the last value at O_{l+1} is `true` or `false` is determined in line 5. We see exemplary, shortened recursive calls to `addVerticesToSelectionByCoordinates`

in lines 7, 10, 13, 15, 18 and so on which are returned at the end of the function.

4.3.2.2 `removeVerticesFromSelectionByCoordinates()`

This is the counterpart to `addVerticesToSelectionByCoordinates` and the second of two elementary methods that allow users to create and modifiy vertex selections in this application. Using the exact same parameters in identical order, it is a private, recursive function that allows for vertices added to the current selection to be removed again. In fact, the two functions vary so little, we can use the same list of general tasks as shown in 4.3.2.2. The crucial differences are highlighted in bold text.

1. check if any of the given input point's coordinates are located outside of the bounds of the root of the `ocTree` indexing the 3D object. Throws exception if this is the case.
2. check if the current node is a leaf (i.e. has no child nodes). This terminates the recursive search and proceeds with finding vertices around the given input coordinates, as it is save to assume that the correct node is found.
3. add all vertices within this node to the set of vertices to be checked.
4. determine if the radius around the query point crosses the bounds of this node in any direction. If so, add all the vertices of the neighbouring node to the set of vertices to be checked.
5. iterate over the entire set of vertices to be checked to find which ones lie in radius around the query point and **remove those from** the set of selected vertices.
6. if this node is not a leaf node - steps 3, 4 and 5 were skipped in this case - determine the correct child node in which the query point is located and have that node recursively call **`removeVerticesFromSelectionByCoordinates()`**.

At this point it is worth noting that these two fundamental functions were mapped to two different buttons on the hand-held controller used at the V2C. Figure 4.4 shows a simple 3D object at various stages during consecutively performing these fundamental operations on it. The leftmost part of the figure shows a simple sphere with no vertices selected. The middle part shows the same sphere after eight vertices have been added to the selection and how they are highlighted. The righthand part of the figure shows the object after four vertices have been removed from the selection via `addVerticesToSelectionByCoordinates` with other coordinates.

4.3.3 Tracking selection

The set of user selected vertices is being kept track of as described in this section. Using the two fundamental functions described in 4.3.2.2 and 4.3.2.1, every vertex of a spatially indexed 3D object can be selected and deselected at will. At each frame, every currently selected vertex is highlighted in an easily distinguishable bright red shade. Additionally, at any point during runtime, the server program of the application can print the current set of selected vertices to an external textfile.

Vertices in this application can either be selected or not selected. Only selected vertices are tracked. This is implemented using the following two variables in the main class.

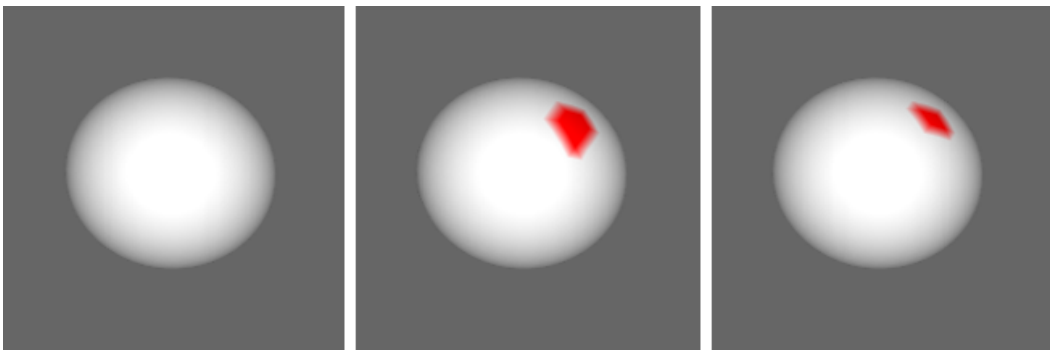

Figure 4.4: A basic vertex selection process

offset (`size_t`)	coordinate (`glm::vec3`)
0	[0.481, 0.461, 0.458]
1	[0.351, 0.335, 0.024]
2	[0.385, 0.324, 0.021]
3	[0.452, 0.342, 0.125]

Table 4.3: content of `verticesList` at startup

`std::set<size_t> vertexSelection` - a set of unsigned integers. A `set` is a C++ specific type of container with two handy properties for this use case. It guarantees the uniqueness of each element, as the value of an element is also the key used to identify it, and it is allocator-aware, which means it uses an allocator object to dynamically handle its storage needs (see [set]).

`std::vector <std::pair <size_t, glm::vec3> > verticesList` - A `vector` containing `pairs` of unique `size_t` values and `glm::vec3` objects. `vectors` are contiguous storage containers that can store any type of objects, provide access to elements by using offsets (just like arrays) but also change their size dynamically (see [vec]). Since the actual content of this variable is created and stored during spatially indexing a loaded 3D object during the recursive building process of an octree structure (see 4.3.1.7), its final required size is not know at startup of the application, hence an ordinary `array` is not a suitable datatype here.

After the setup process of the application, `verticesList` essentially holds an ordered list of pairs of integer numbers and three-dimensional coordinates. Table 4.3 illustrates an excerpt of its possible content after startup. Note that the order of coordinates is arbitrary and, in no way, represents the structure or order of the loaded 3D file.

4.3.4 Testing setup

This section describes the steps taken to ensure that the selection application meets all its requirements and ensures its key features described above are implemented correclty. While it is not formally verified that the application works correctly in every use case conceivable, it was tested up until a point where fluent, real-time and comfortable interaction was possible from a user perspective.

4.3.4.1 Test files

While developing the selection application, I used 3D modeling software to create a set of simple test files that would be exported to .obj format so they could be used for testing purposes. These objects were designed to test whether common queries are handled quickly and correctly as well as what happens in unfortunate edge cases. Such a fairly complex edge case is depicted in figure 4.5.

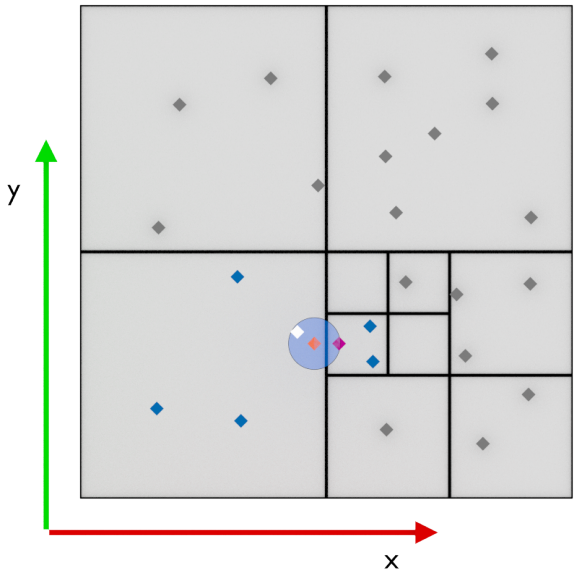

Figure 4.5: A fairly complex query to the octree structure of the selection application

For the sake of readabilty, figure 4.5 depicts an orthographic view on the structure of a test file used, showing only x and y axes. In such a case, all vertices shaded blue must be considered but only the one vertex shaded white is a correct result to the query. An additional query with input coordinates depicted by the point shaded in purple and radius 0 must be invoked and correctly handled as well, since the righthand border of the target subtree is crossed by the search radius. All vertices shaded grey must not be considered. The behavior of the application must be congruent to what is described in section 4.1.

Figure 4.6 shows a test file used during all stages of implementing the selection application. It is designed for the Octree structure to be built in a clearly predictable way. For each combination of the maximum allowed number of vertices per node and the maximum allowed level, construction of the structure can be precicely predicted because all parts of the object are positioned in eighths of the space encapsulated by its bounding box. This file was used to a great extent for many testing queries as well as debugging.

The debugging process encompassed a multitude of test calls to the applications core functions using arbitrary coordinates as well as extensive use of the helper and debugging functions described in the subsequent subsection.

4.3.4.2 Helper and debugging functions

The following functions which are non-essential to the functionality of the application are included. They were used during implementation for documentation, debugging and con-

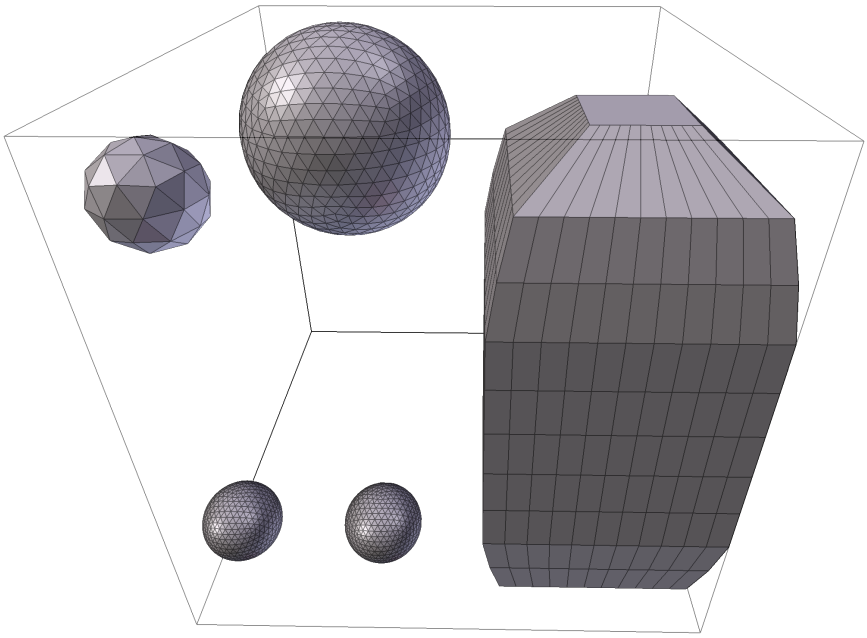

Figure 4.6: A test file used during implementation, perspective view. Its bounding box and edges are depicted

trolling the correctness of the core features. Parameters for these functions are omitted.

`ocTree* getNodeByIdentifierArray()` - gets a leaf node via a boolean input array representing its identifier. Traverses through an entire tree and returns the node with given identifier at the lowest level. The input boolean array has to contain 3∗ the maximum allowed split depth boolean values representing the identifier of a node. For example, at a maximum allowed split depth of 3, `false, false, false, true, true, false, false, true, true` is interpreted as 000110011 and a pointer to the according leaf node is returned.

`bool getIsLeaf()` - determines whether this is a leaf or not. Returns private bool variable `m_isLeaf`.

`void debugTreeInfo()` - prints information about the finished `ocTree` object to the console, including total number of vertices stored by it, maximum allowed number of vertices per leaf, maximum allowed split depth and dimensions in x, y and z direction.

`void debugFirstVertex()` - prints the coordinates of vertex at index 0 `m_vertices` to the console. This is a quick way to verify that import via ASSIMP worked.

`debugAllVertices()` - prints the coordinates of all vertices to the console. This is a way to verify that import via ASSIMP worked correctly.

`void debugIdentifierasString()` - prints the boolean identify array of the calling subtree to the console in an easily readable form, i.e. 001101 for example. If the calling node is the root, "I am root" is printed instead. Additionally, `m_level` as well as the identifer array of the parent node of the calling subtree node are printed.

In addition, all essential functions of the `ocTree` class have an optional parameter `debugInfo`, which is set to `false` by default. If set to `true`, information is printed to the console during execution of the functions. Depending on what function is called, this information varies strongly obviously. In general, all subtree nodes that recursively call any relevant functions as a result of the initially called function, call `debugIdentifierasString()`.

4.3.4.3 Testing at the V2C

I spent multiple hours in the projection installation, trying the application myself. By experiencing the application from a user perspective, I wanted to make sure the feedback is quick, clearly noticable and easiyl understandable. I tried to find bugs and wrong behavior by using the hardware in ways that were not planned on. Formally describing the procedure does not make sense, so consider the following brief descriptions of things I tried and what insights I gathered from them.

Finding limitations to the tracking setup - The projection installation of the V2C is 2.7 meters in width, height and depth. I tried to find out where the tracking system was working reliably and precisely and where the tracked space ended. I did not take exact measures but I found out that at a distance of about 30 centimeters away from the walls and the floor, the tracking started to get inaccurate. The positions of the tracked devices would freeze at the last position it was tracked correctly, thus making further selection or deselection operations impossible as well as leading to wrong projection of the scene.

Finding limitations to response time - At a slow pace of movements, the projection thrown onto the walls surrounding the user, the visual feedback on where selection operations take place as well as indication of their results are in perfect synchronisation with what a user would expect. However, when moving the hand-held input device rapidly, its tracked position would get corrupted and the bright green object indicating its position within the 3D scene would only be rendered every dozen or so frames, being frozen in place in the meantime.

Finding limitations to synchronisation - During the selection process and during the user study, the application froze multiple times. One or more projection programs would not react anymore, leaving no option but force-quitting and restarting the application. I was not able to reproduce this behavior. I can only speculate, based on various times trying this myself, that if a button is pressed for an ongoing timespan while rapidly moving the hand-held input device, this behavior seemed to occur mor frequently. Again, a formal description of what happened as well as the reasons why it happened can not be provided. A more extended testing and bugfixing process would be needed and this goes beyond the scope of this work.

Evaluating immersiveness and projection quality - As described in section 5.5.2, the framerate of the projection onto the the top wall was noticeably higher than those of the other projectors. During the user study, no remarks regarding this were made but from a developer stance, this can potentially impair immersion. Upon taking a look as close as possible to details of projected objects, the overlay of the two projections became visible, causing an incorrect spatial effect and blurriness. However, this only

was noticeable when the tracked stereoscopic glasses got really close to the surface of a projected model.

In conclusion, it is fair to say that the *selection application* provides clear feedback on what is happening based on user interaction and interaction is possible at a not too fast movement speed. Users should keep in mind that the tracking system has limitations. Not getting close to walls and not moving the hand-held input device rapidly while simultaneously keeping buttons pressed ensures a convenient, fluent interaction.

5 User study

From August 21th to September 12th 2017, I conducted a user study that aimed at gathering data to compare with the results of processing 3D objects according to *mesh saliency* [LVJ05]. However, due to the fact that the projection installation at the V2C was not available for me at all times, I could not use every day in that timespan. The installation was assigned to me for a total seven days. I was able to collect a reasonable amount of data from a sufficiently sized group of participants. This chapter will go over details of how I organised and conducted the user study.

As briefly described before in section 3.2, I asked participants to use my selection application and the hardware available at the V2C to select regions of 3D objects they found *interesting*. In general, participants got familiar with the devices and interaction mechanics quickly and were able to make selections according to their perception of what was of importance. In general, the ratio of overall differences between *mesh saliency* and *user saliency* is fairly low for the three objects used in this work, ranging from about 0.23 to 0.38. However, whether this can be used as basis for any type of statement regarding how differently important parts of 3D data are recognised by humans in VR and computational, automatic procedures, is doubtful as discussed at the end of this chapter.

5.1 Hardware used

The five-sided projection installation at the V2C is equipped with several devices which can be tracked inside of it. Users taking part in the user study were asked to put on a pair of lightweight stereoscopic glasses, shown in figure 5.1. Note the three white, spherical parts on each side of the glasses. These are used for tracking their position, orientation, tilt and yaw within the projection installation. According to these parameters, two images are projected onto the walls of the installation from behind, each for one eye. The glasses are in synchronisation with the projectors and actively switch between left and right 120 times per second, only showing one of the projected images at a time. This way, an immersive 3D effect for one user wearing the glasses within the installation is created.

For interaction, a hand-held device called *wand* see figure 5.1, is used within the projection installation. It allows for navigation within the scene via a little joystick as well as interaction with four buttons. The joystick moves the scene in the opposite way the user pushes it. This has been described as counter-intuitive by some users.

As described in 3.1.2, two of the three buttons surrounding the joystick are enough for the selection application's needs. The two essential functions of the application are mapped to them in the following way. The button on top of the *wand*, on the left side of the joystick is used for making selections on the currently loaded 3D object, the button right to it can be used to clear already marked selections. Figure 5.1 shows the device. Again, note the white, spherical parts used for tracking position, orientation, tilt and yaw of the device attached to it.

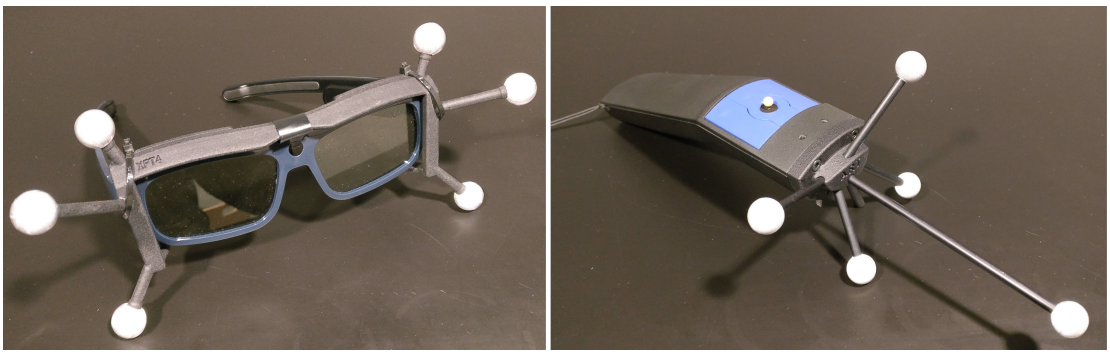

Figure 5.1: The hardware components available at the V2C. Left: A pair of active, stereoscopic shutter glasses. Right: The main input device, referred to as *wand*

Figure 5.2 depicts what the selection process inside the projection installation at the V2C is like form a user perspective. Users utilise the *wand* for navigation as well as selecting and deselecting parts of the presented objects they deem *interesting*. The stereoscopic glasses provide a spatially convincing, three-dimensional immersion of the object. In the figure, the object is a simple cube with two iterative subdivisions. Its vertices are shaded gray, the edges connecting them are shaded black. The diamond-shaped object indicating the position of the *wand* within the 3D scene is shaded bright green. The blue circle around it represents the spherical query radius for selection and deselection operations. Vertices within that lie within that circle are shaded orange. Figure 5.3 shows a real selection process within the projection installation at the V2C.

Consider figure 5.3. Starting with the top photograph of the figure, it shows a loaded object where with nothing selected yet. The middle photograph shows the same object after a selection has been made. The bottom photograph shows the object after vertices have been deselected again. Note the green object that is also projected onto the walls. For the user, through the tracked glasses, it is visible at the tracked location of the input device at all times.

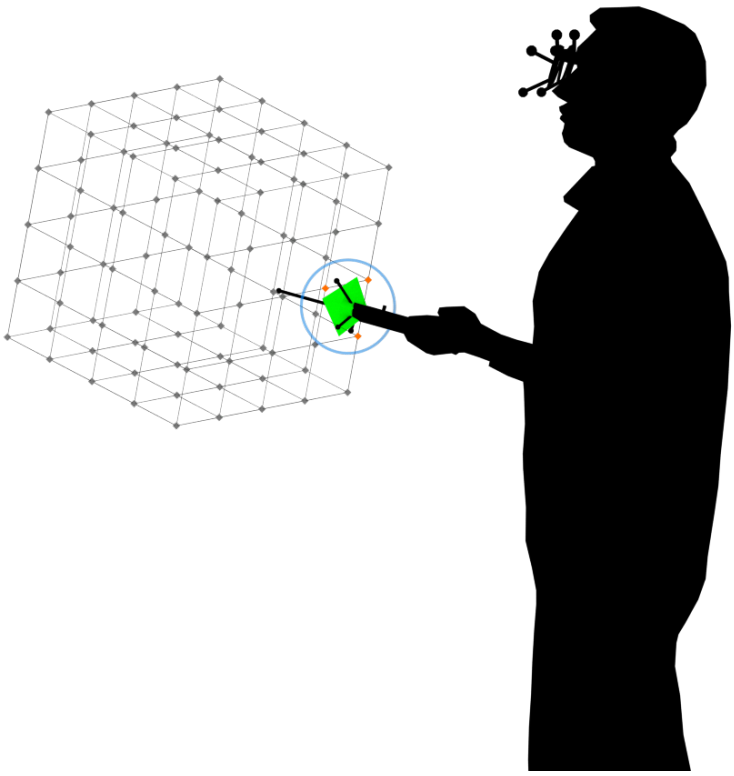

Figure 5.2: The selection process using the available hardware at the V2C

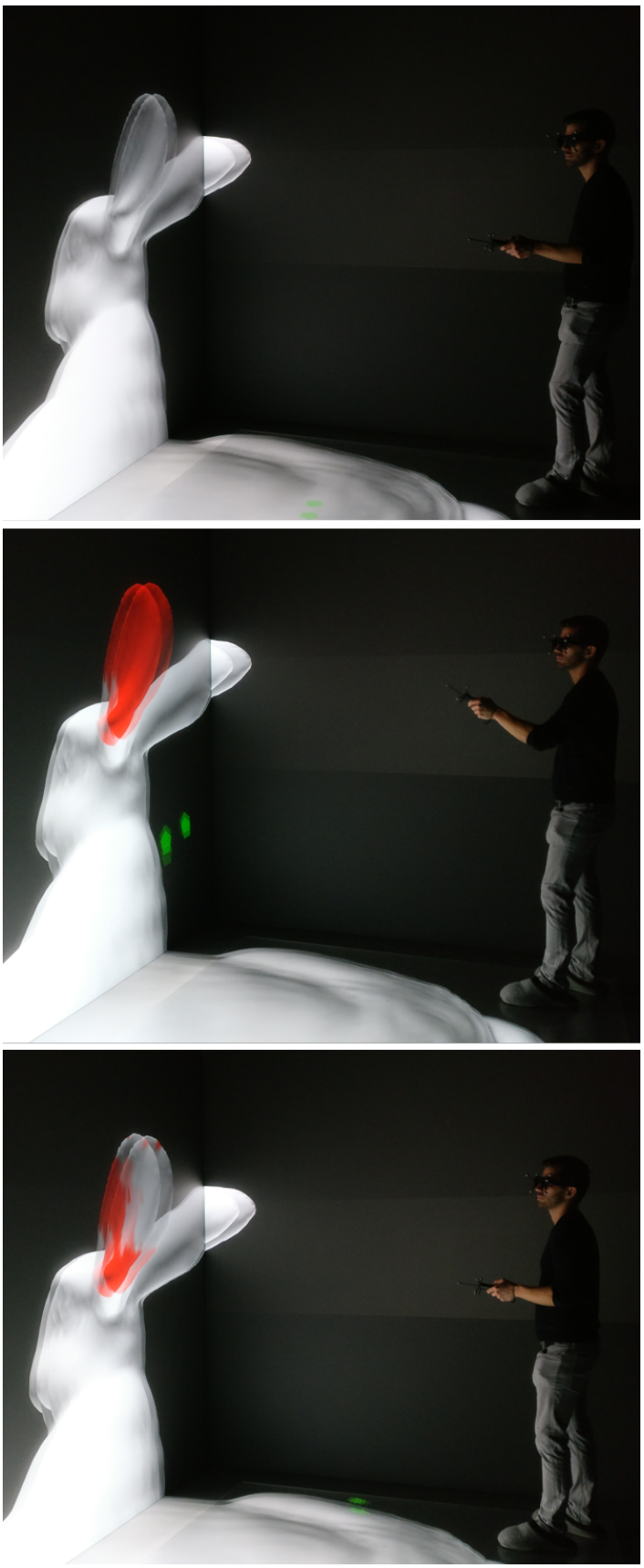

Figure 5.3: A real selection process

5.2 Models used

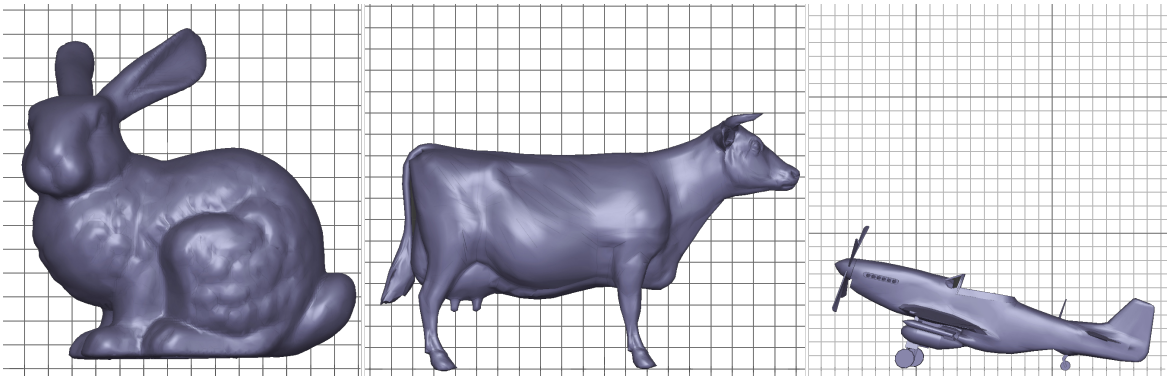

Figure 5.4: The 3D objects used for the user study

Regarding objects used for the study, I aimed at offering as much variety of types of objects among as few objects as possible. The motivation for this was to achieve results that are generally applicable for any type of geometric data, at least to an basic extent. Also, to avoid the users losing motivation due to repeating tasks during the study, keeping the number of objects to a minimum was a constraint to be considered. I decided to use three objects described table in 5.1 and depicted in figure 5.4. Note that the models are from third-party vendors. I made sure they are all free to use for educational purposes and weblinks to where they have been published originally are provided in the bibliography.

Figure 5.4 shows the objects used in the user study. From left to right, the 3D scanned bunny, the modelled cow and the low-poly, modelled aircraft are shown. Consider the grid to get a sense of their proportions. I did not scale them according to their real world sizes and instead made them take up approximately equal space in the virtual scene. This was done with the intention to provide a similar level of visual detail, independent of the actual geometric level of detail, for each object. Furthermore, I wanted to prevent the size of the objects from having any sort of influence on how users perceived them.

object	created through	source	vertex count	class of objects
cow	3D modelling	[cow]	69,648	purely natural
P51 Mustang	3D modelling	[P51]	51,708	purely mechanical
bunny sculpture	3D scan	[bun]	68,754	natural, man-made

Table 5.1: 3D objects used in the study

5.3 Tasks given

Users were asked to use the hard- and software available to select regions and parts of the objects they deemed *interesting* since *mesh saliency* claims to be able to reliably predict such parts [LVJ05]. I told the users that this would be the task at hand only after they stated they felt ready to interact with the scene presented in the projection installation. They were given as much time as they wanted to get somewhat accustomed to navigating and the selection process it but in general, nobody took longer than a few minutes.

I then explained the course of the user study, going over the following instructions:

- The task is to select parts and regions of the objects users consider *important*.
- Some suggestions as to what could be considered *important*:
  - *what do you consider visually interesting or important?*
  - *what do you assume are natural focus points of attention?*
  - *what parts do you consider vital for identifying the object?*
- However these are just ideas, in general users were encouraged to select whatever they, personally, deemed *important* according to their own understanding. I made sure to emphasise this because I aimed for natural results.
- There is no *correct* way to make the selection. Users were, again, encouraged to make selection according to their perception.
- How precise the selections would be was up to users. If they were satisfied, that was the desired level of precision
- I asked to consider symmetry to some extent in the selection where possible.
- The time limit per object was five minutes. I made clear that the selection process would be stopped after that period of time.
- However, if users would be satisfied with the selection earlier, I encouraged them to say so. In this case, if there were still objects left to work with, the next one would be presented. Otherwise, the practical part of the user study was finished.

5.4 Questionnaire

After users completed the *practical* part of the study, they were asked to fill out a short questionnaire, determined to acquire demographic data about the participants as well as feedback as to how they perceived interacting with the selection application. The data gathered through this are presented in section 6.1.

The questionnaire was two pages total. After stating that user data would be used anonymously and not be provided to any third parties, I also made sure to leave my contact information on the questionnaire. After this, the first page contained single-choice questions about gender, age, profession, prior experience with VR technology and whether the user was strongly visually impaired. Page two contained statements to which participants were asked to respond to in terms of how strongly they agree or disagree to with them. The (translated) statements read as follows:

- *I am interested in technology in general.*
- *I am interested in VR/AR technology.*
- *I easily got familiar with the selection application.*
- *The spatial impression within the projection installation was convincing.*
- *Navigating within the selection application felt comfortable and / or intuitive.*
- *The selection process felt comfortable and / or intuitive*
- *The selection process felt precise*

Users were asked to rate each of these statements on a Likert-scale. I offered five options to chose from: 1. *I strongly disagree*, 2. *I disagree*, 3. *No comment*, 4. *I agree* and 5. *I strongly agree*. An example of the questionnaire (in German) can be found the appendix of this work.

5.5 Shortcomings of the study design

While conducting the user study, I noticed some factors which possibly had negative impacts on user behavior during the study and, consequently, its results. On top of that, a few technical issues emerged which could have made the experience for the users less immersive and convincing. I will go over all these issues in the following subsections. I cannot describe the extent they influenced the study, this is solely meant to document them.

5.5.1 Task, instructions and questions

While I did my best to design the tasks and questions (see section 5.4) to be as clear as possible, during conducting the user study, some flaws became prominent. For more on what users remarked, see section 6.3. The following notes document my own, personal observations.

- The task of selecting *interesting* parts of the objects is both hard to describe and possibly not close enough to what *mesh saliency* is meant to identify. What is *interesting* and what is not, is a very personal, subjective decision, so results are bound to vary strongly. Participants in the user study were encouraged to select parts of the objects based on their personal perception and understanding of the term *interesting* because a wide range of results was desired. However, a more uniform definition of terms might have been a reasonable basis for the comparison at the centre of this work too.
- As a result of the former note, adding an opportunity to rate the instructions given on the questionnaire might have given more insights as to how well designed the user study really was. After taking some time to get familiar with navigation within the projection installation and using the selection application, I made sure users had no questions left before the actual study was started. Questions here were very scarce so this note is not directly based on user feedback. However, as the study went along, I noticed that this could have been a valuable addition to the questionnaire.

- Another possibly relevant question I missed to put on the survey was whether users felt pressured by the time set time limit of five minutes per object. While the majority users proclaimed they were satisfied with their selection before time ran out, there were a few participants I had to stop after five minutes for every object. I chose the time limit to make sure users would not lose motivation over time and for the sole purpose of having a constant time factor. Again, I received no feedback from users indicating that this was an issue but explicitly asking them about it might have been relevant for completeness of this work.

- Regarding the rating of how much users agree or disagree to the statements on the second page of the survey, using *I am indifferent* instead of *No comment* as the third, neutral choice could have possible influenced the outcome of the study as well. I did not include this option on purpose because I explicitly wanted to avoid neutral answers while also not *forcing* the users to make a decision.

5.5.2 Technical issues

The projection installation at the V2C, while still fully functioning and performing well for the majority of time, is not state of the art anymore. The hardware has been in use for xx years, some parts of it display signs of deterioration and some have even been replaced already. Maintenance work is frequently required. This section will cover some aspects that were a result of this and may have had influence on user behavior during the study.

- Weeks prior to the user study for this work started, parts of the projector that covered the top wall of the installation were replaced. This led to projections onto that wall running at a significantly higher frame rate. As a result, quick head movements, especially horizontal, with the image being spread over the upper wall and any of the others at the same time, caused the part visible on the upper wall to move much faster than the rest of the projection.

- On Monday, September 4th, the sixth day of the study, with five more days still ahead, one of the projectors died. The lower part of the *southern* wall could not be used for displaying anything from then on. I had to advice users to try and only use the other sides of the installation but this certainly impaired the immersion.

- Tracking of the stereoscopic glasses and the *wand* did not work properly when they got near to the bottom or the walls of the projection installation. If they entered that *outer area* of the room, projection would freeze in place until the user stepped closer to the centre of it again.

- This might have been a problem with my application - I could not reliably reproduce this error. Sometimes the application simply froze during the user selection process. I tried to reproduce this behavior but I can only speculate from what I have observed. This kind of freeze occurred more often when the *wand* was moved very quickly over an extend period of time, while constantly pressing the *select* button.

6 Results and discussion

6.1 Participant demographics

In total, 32 participants took part in the user study. 16 were female, 16 were male. No participant had a major visual impairment that would have caused their selection invalid. 14 participants stated they had previous experience with VR technology while the other 18 stated they had never used such technology before. Consider figures 6.1 and 6.2 for more information on demographics of participants.

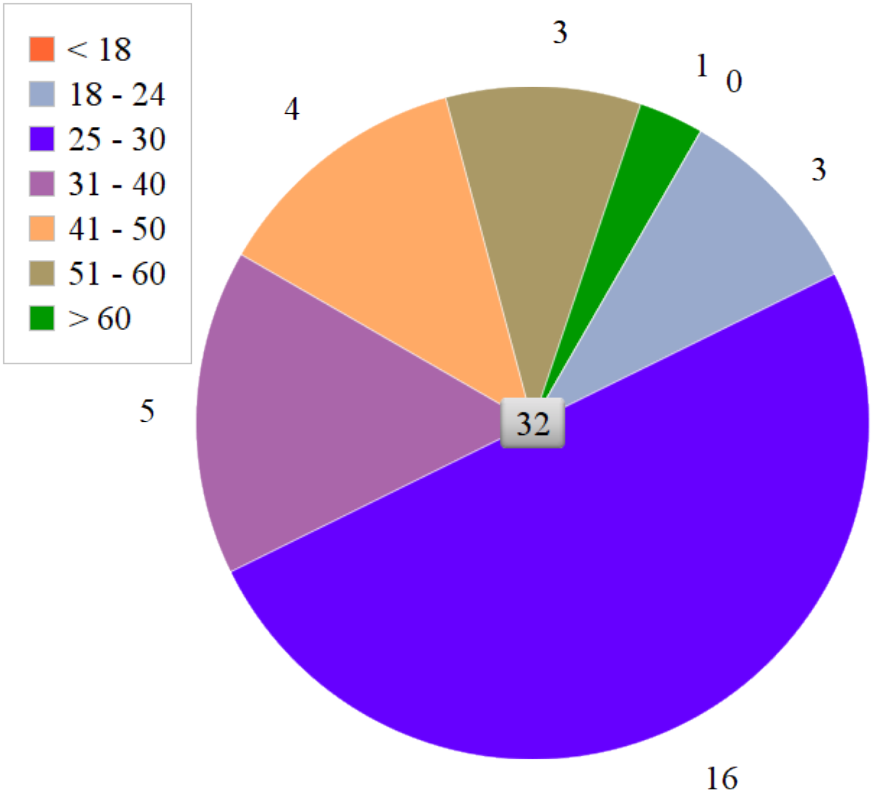

Figure 6.1: Age distribution of participants in the user study

The majority of participants was employed and between 25 and 30 years of age.

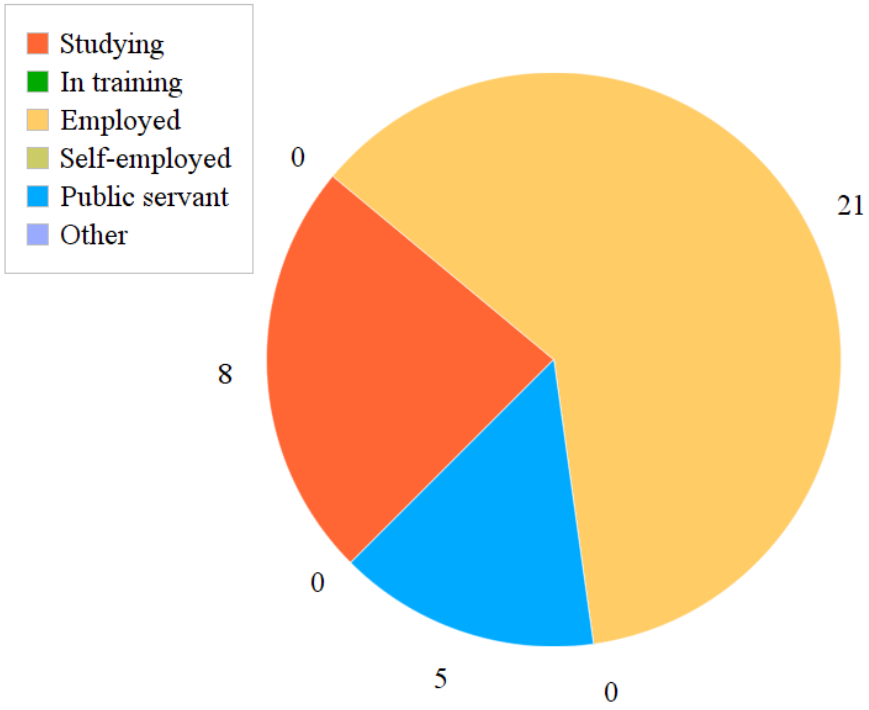

Figure 6.2: Working situation of participants in the user study

6.2 Feedback from the questionnaire

As described in section 5.4, at the end of the user study, participants were asked to rate a set of statements on a Likert-scale. Figure 6.3 shows the data gathered from this questionnaire. If a user ticked *No comment* for a statement, it was not considered for these average values. The statements are shortened in this figure, the full wording can be found in 5.4 as well as appendix

Figure 6.3 shows the average rating of statements on the questionnaire. Answers count as follows.

1. *I strongly disagree*: 2
2. *I disagree*: 1
3. *No comment*: 0
4. *I agree*: -1
5. *I strongly agree*: -2

.

Especially the precision of the selection process seemed to be not quit satisfactory to a lot of users. While the average result is still *positive*,

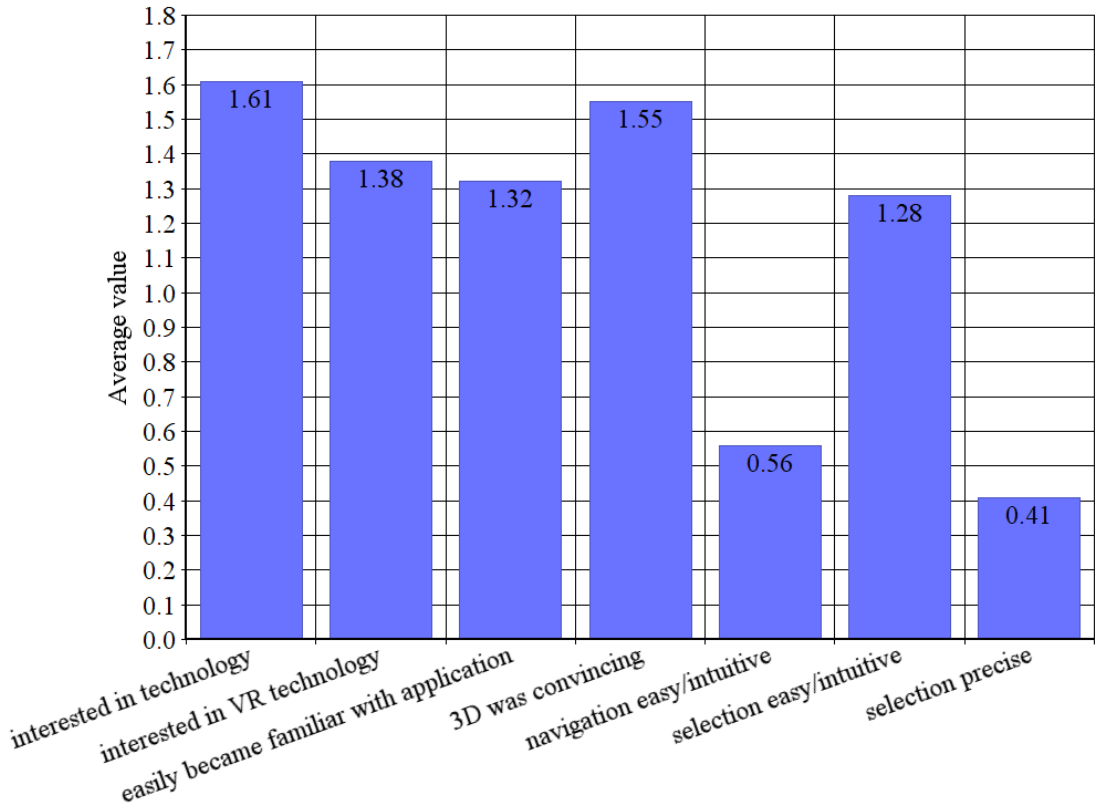

Figure 6.3: Feedback on the application gathered from the questionnaire. *No comment* answers are not considered.

6.3 Individual feedback and comments

Based on individual, informal feedback given by some participants, this section documents findings from these conversations.

- The navigation shown in figure 6.3, the navigation within the projection installation of the V2C was perceived as intuitive, easy or natural. A complete, formal description of how the navigation is designed, is not included in this work but one its general principles is that movement within the scene is always the opposite of the direction in which the user pushes the joystick of the hand-held input device, the so-called *wand*. A few users have independently stated that they found this contra-intuitive and confusing.

- Furthermore, regarding navigation within the virtual scene, users stated that rotating the scene is confusing. The scene rotates around the user. So when a loaded object is far away from the user, rotation will cause the object to seemingly travel quickly along a horizontal circle around the user. There are ways to prevent this and rotate the object around its local z-axis but this needs practise.

- Regarding how the *selection application* itself works, one user stated that a visible selection target with arbitrarily scalable radius would help immensely with improving

the perceived precision of the selection process. As seen in figure 5.3, the selection radius around the tracked position of the hand-held input device is not visible to the user. This was a design decision with the goal of keeping the interaction individual by not rendering semi-transparent, unnatural objects, however it is understandable that users who desire maximum precision would want have the selection radius visible and adjustable at all times.

6.4 Saliency and difference maps

This section contains figures depicting *mesh saliency* and *user saliency* maps for each object as well as *unweighted* (raw) and *weighted* difference maps. For each object, these maps are shown in three angles, front, side and top. All figures are orthographic renderings. The scale shown in figure 6.4 is used for all figures. Note that it is uniform in the sense that the color mapping represents values between 0.0 and 1.0. For saliency maps, this expresses the saliency (the *perceived importance* per vertex), for difference maps it expresses the differenc between saliency maps vertex wise.

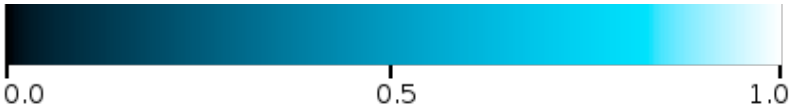

Figure 6.4: The uniform scale used for figures in this section.

6.4.1 Bunny

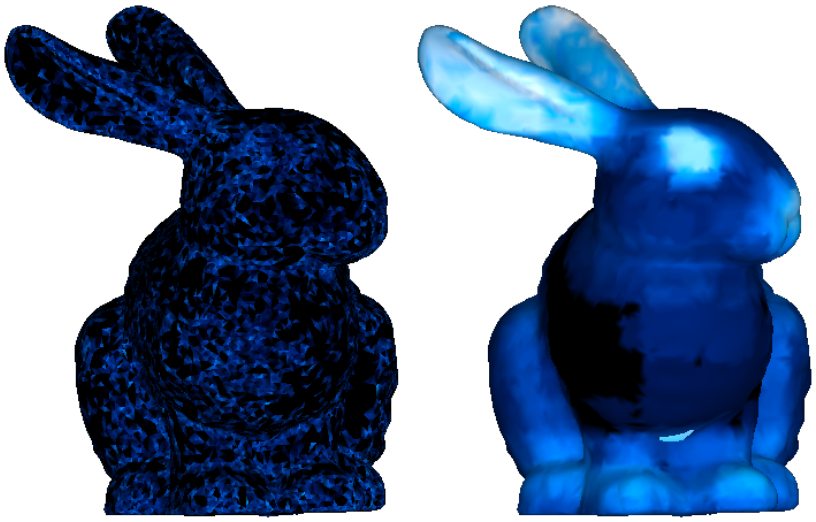

Figure 6.5: Normalised saliency maps for the first model, orthographic front view. Left: *mesh saliency*, right: *user saliency*

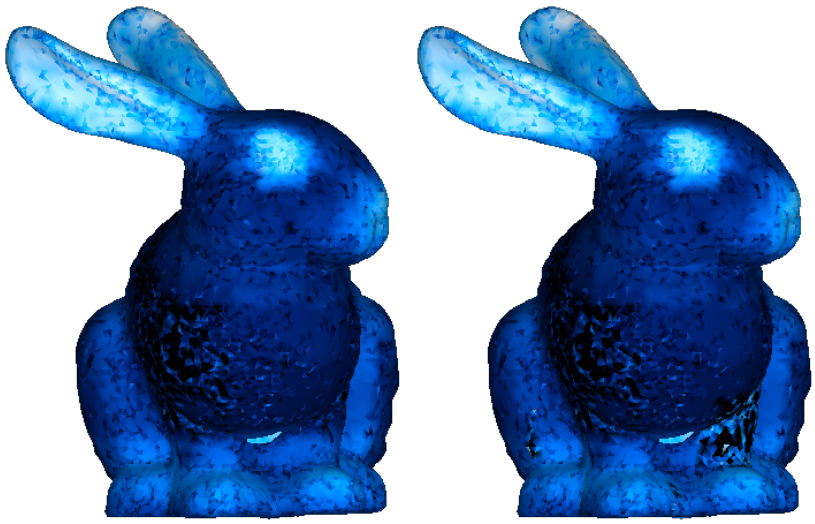

Figure 6.6: Normalised difference maps for the first model, orthographic front view. Left: *unweighted differences*, right: *weighted differences*

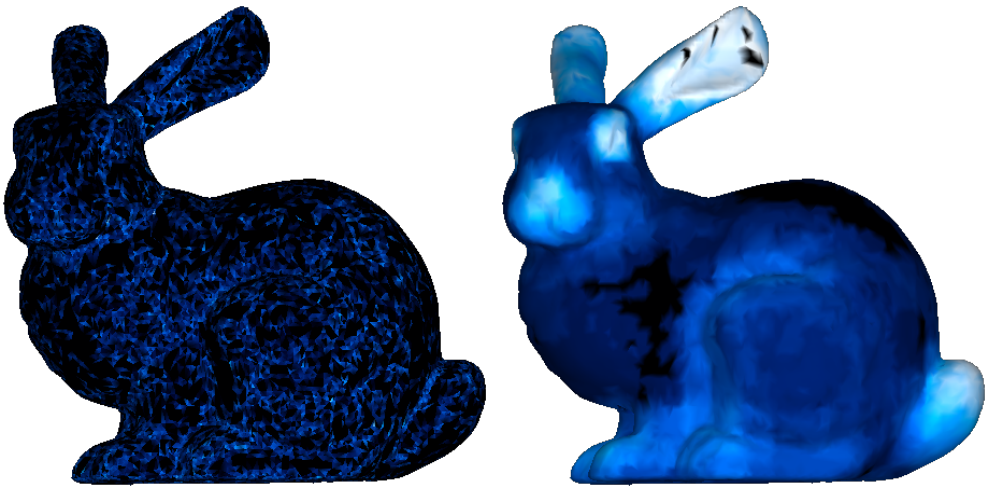

Figure 6.7: Normalised saliency maps for the first model, orthographic side view. Left: *mesh saliency*, right: *user saliency*

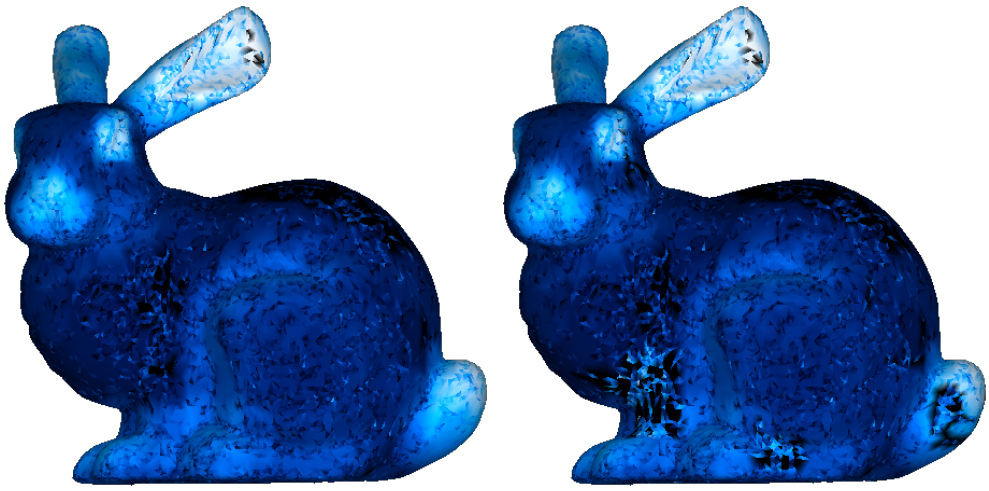

Figure 6.8: Normalised difference maps for the first model, orthographic side view. Left: *unweighted differences*, right: *weighted differences*

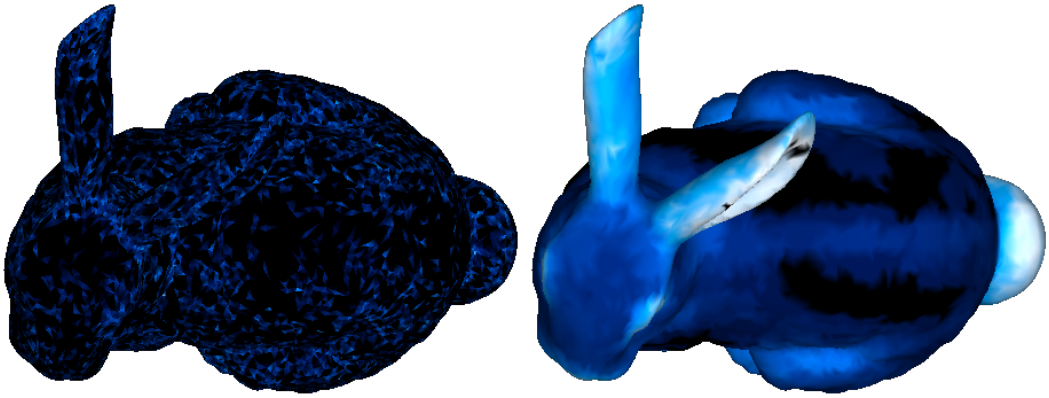

Figure 6.9: Normalised saliency maps for the first model, orthographic top view. Left: *mesh saliency*, right: *user saliency*

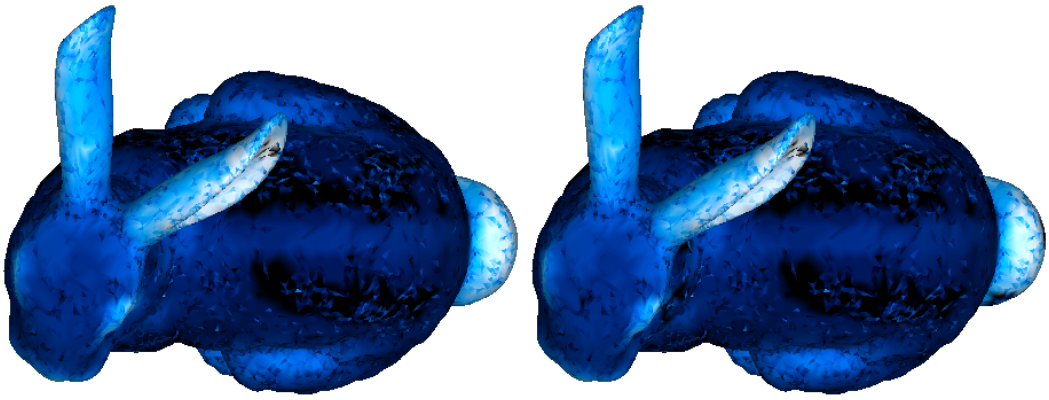

Figure 6.10: Normalised difference maps for the first model, orthographic top view. Left: *unweighted differences*, right: *weighted differences*

6.4.2 Cow

Figure 6.11: Normalised saliency maps for the second model, orthographic front view. Left: *mesh saliency*, right: *user saliency*

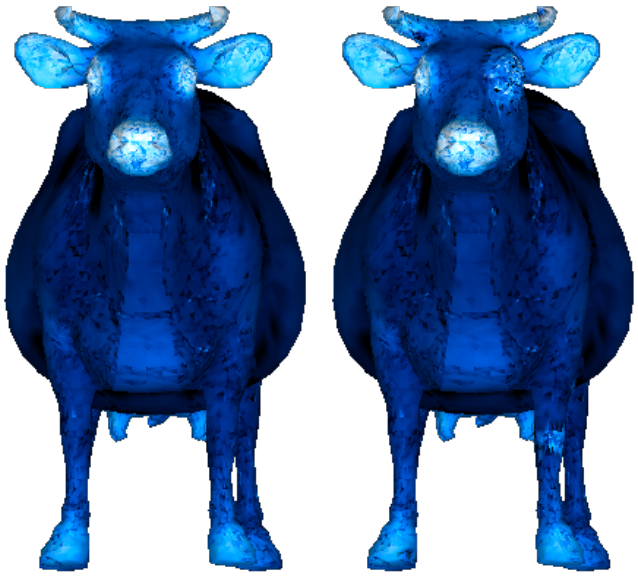

Figure 6.12: Normalised difference maps for the second model, orthographic front view. Left: *unweighted differences*, right: *weighted differences*

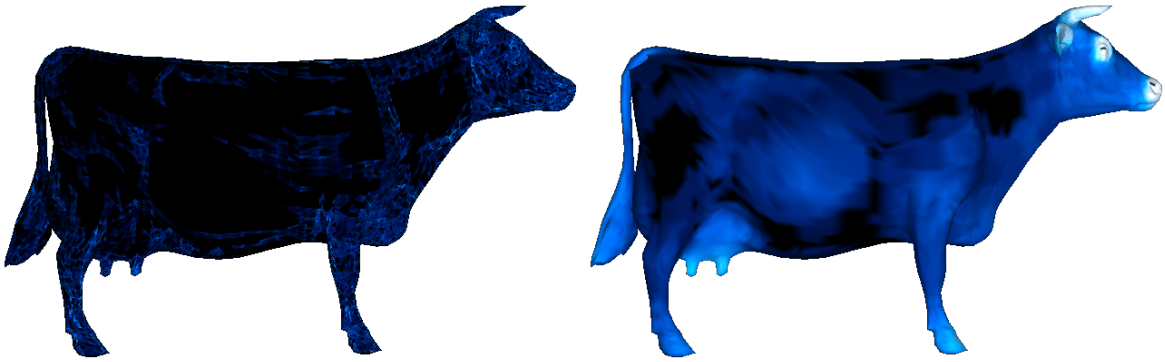

Figure 6.13: Normalised saliency maps for the second model, orthographic side view. Left: *mesh saliency*, right: *user saliency*

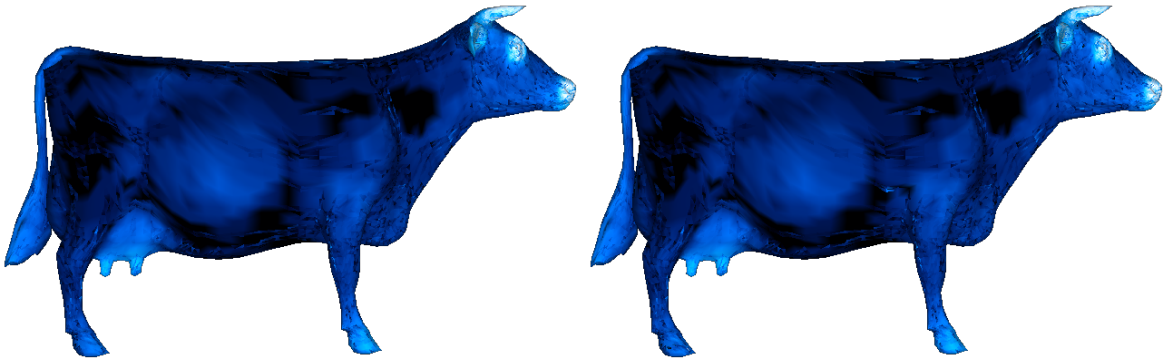

Figure 6.14: Normalised difference maps for the second model, orthographic side view. Left: *unweighted differences*, right: *weighted differences*

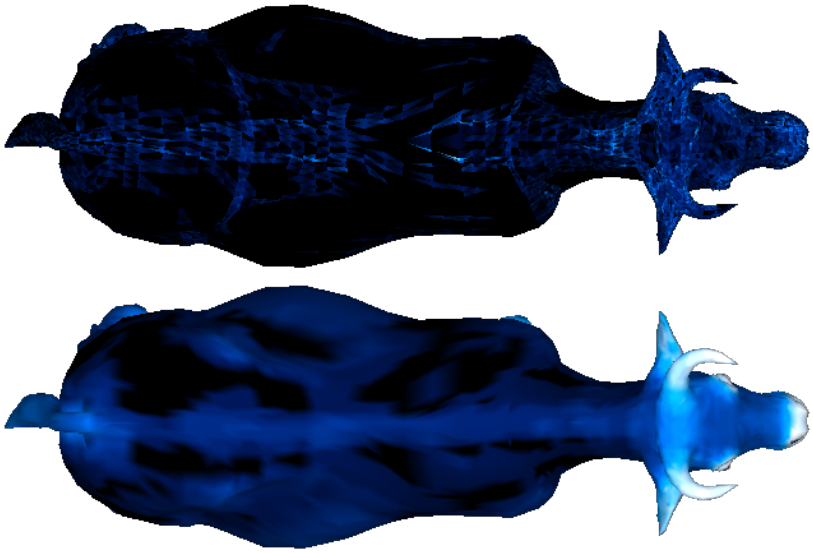

Figure 6.15: Normalised saliency maps for the second model, orthographic top view. Top: *mesh saliency*, bottom: *user saliency*

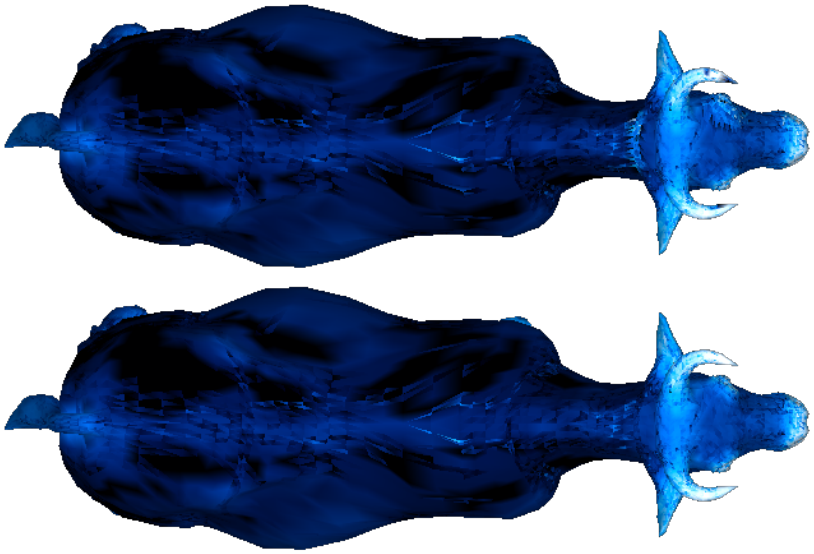

Figure 6.16: Normalised difference maps for the second model, orthographic top view. Top: *unweighted differences*, bottom: *weighted differences*

6.4.3 P51

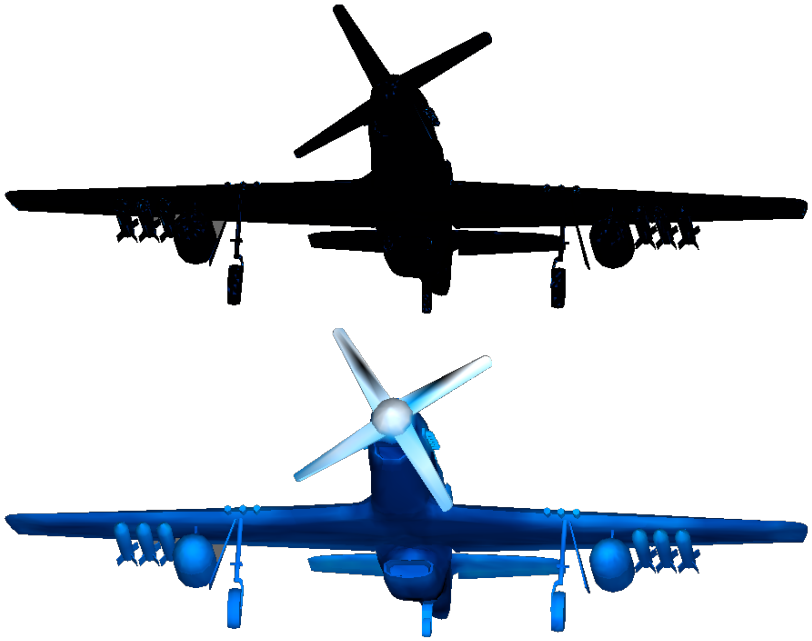

Figure 6.17: Normalised saliency maps for the third model, orthographic front view. Top: *mesh saliency*, bottom: *user saliency*

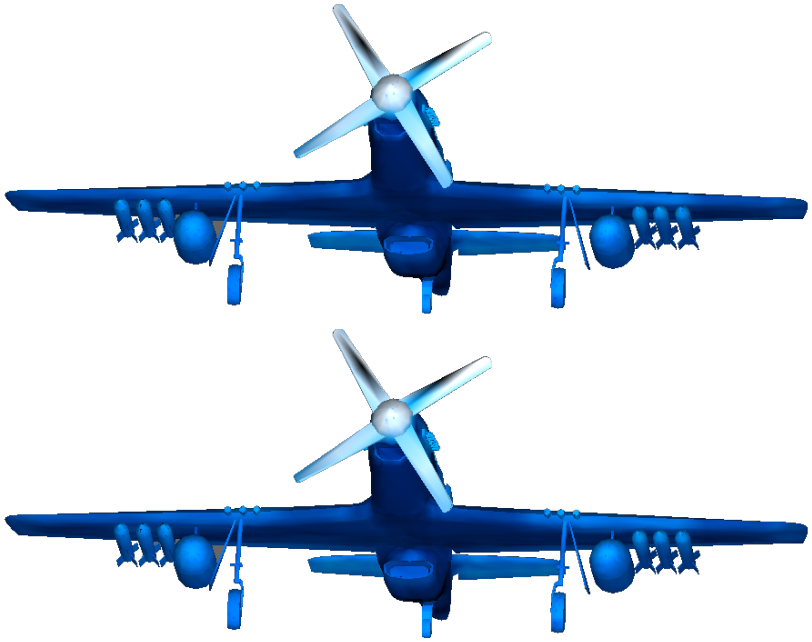

Figure 6.18: Normalised difference maps for the third model, orthographic front view. Top: *unweighted differences*, bottom: *weighted differences*

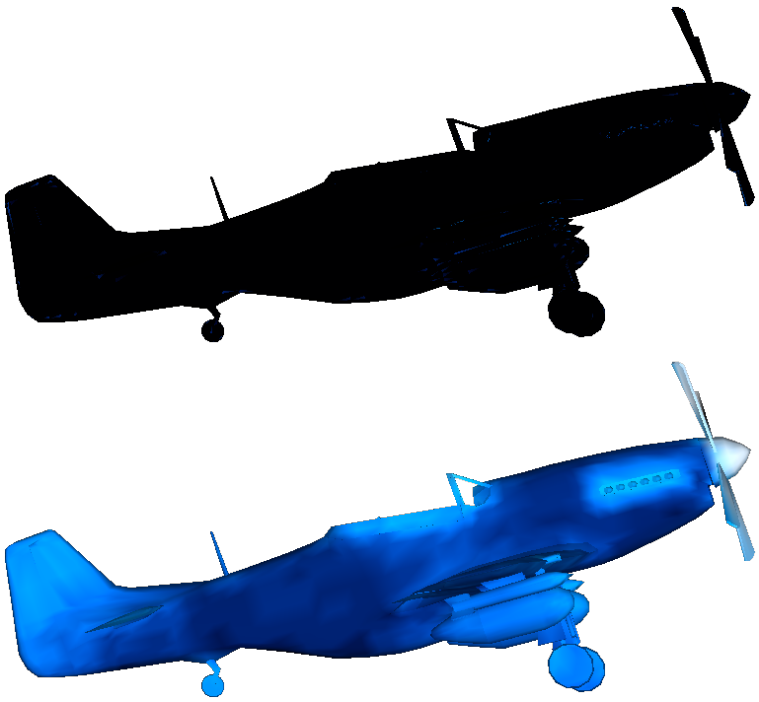

Figure 6.19: Normalised saliency maps for the third model, orthographic side view. Top: *mesh saliency*, bottom: *user saliency*

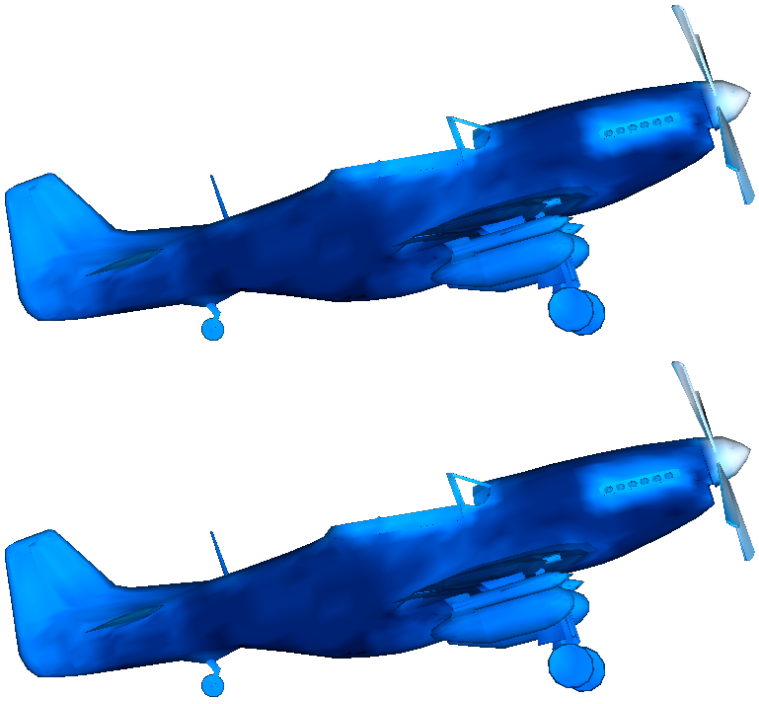

Figure 6.20: Normalised difference maps for the third model, orthographic side view. Top: *unweighted differences*, bottom: *weighted differences*

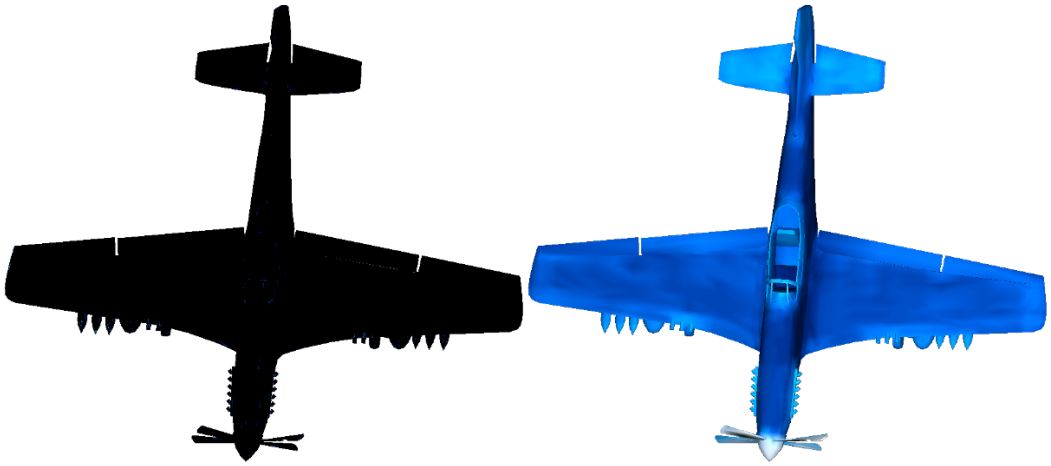

Figure 6.21: Normalised saliency maps for the third model, orthographic top view. Left: *mesh saliency*, right: *user saliency*

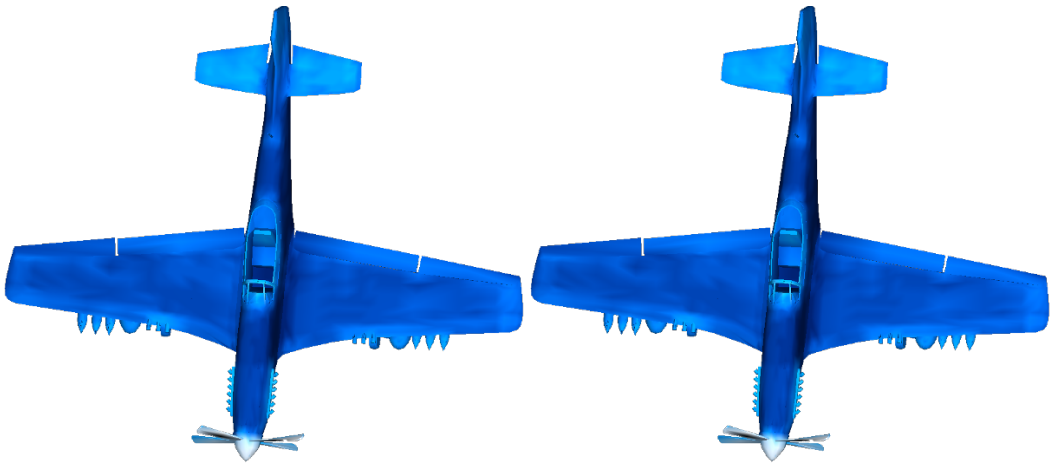

Figure 6.22: Normalised difference maps for the third model, orthographic top view. Left: *unweighted differences*, right: *weighted differences*

6.5 Evaluation of differences

This section contains quantitative observations of the resulting saliency and difference maps as well as a discussion of the measure of difference as suggested in section 6.5.1. Based on observations described here, it is fair to say that the measure of difference is not expressive enough and is not suitable to describe differences between user selections and results achieved through *mesh saliency* in a sound way.

6.5.1 Measure of difference

Table 6.1 shows the results of using the measure of difference as well as additional information about the three 3D models used for this work. The following abbreviations are used:

$$Dif_{raw} = \frac{\sum_{i \in V_o)} \Delta_{raw}(v_i)}{V_o}$$

$$Dif_{weighted} = \frac{\sum_{i \in V_o)} \Delta_{weighted}(v_i)}{V_o}$$

Where, for a given object o, V_o is the total number of vertices contained in o and $\Delta_{raw}(v_i)$ is the absolute value of the difference between the computed *mesh saliency* $S_M(v_i)$ value and the average *user saliency value* $S_U(v_i)$ for vertex v_i.

In other words, Dif_{raw} and $Dif_{weighted}$ can be interpreted as a percent number which expresses how much the average user selection differs from the computed mesh saliency map for a given object. In the case of the first model, Bunny, both the raw, unweighted and weighted overall difference amounts to roughly 23%. The difference only affects decimal places after the third one after the comma. For the second model, the raw overall difference amounts to about 27%, as does the weighted overall difference when rounded. For the third model, both ratios are the exact same at about 38%. n is the factor of ϵ used for computation of *mesh saliency* values for the objects where ϵ is 0.3% of the main diagonal of the object's bounding box.

The *mesh saliency* maps were computed using `climberpi's` implementation [clm] of of the original procedure suggested in [LVJ05]. It is implemented strictly following the concept yet, obviously, I was not able to achieve saliency maps that look similar to those included in the original paper. In the original paper, the authors suggest using scales $\sigma_n \in 1\epsilon, 2\epsilon, 3\epsilon, 4\epsilon, 5\epsilon, 6\epsilon,$. As shown in table 6.1, different values for the objects were used in order to get somewhat plausible *mesh saliency* maps. Trying multiple variations of n, where $n * \epsilon = \sigma_n$ did not lead to results much more convincing.

Model	Vertices total	Vertices weighted	Dif_{raw}	$Dif_{weighted}$	$n * \epsilon$
Bunny	68,754	2,667	0.23435	0.234545	4
Cow	69,648	3,200	0.27431	0.269036	4
P51	51,708	0	0.38002	0.38002	1

Table 6.1: Resulting differences and information on the 3D models

A is a major shortcoming of the measure of difference is the fact that every saliency value is normalised to range $[0, 1]$. As stated before, this was done with the goal of describing how much the user selection differs from the results of *mesh saliency* in percentage. However, the problem with dividing each individual saliency value by the highest value found is that a large number of near-zero saliency values is a result. This is obvious when considering figures 6.5, 6.7 and all the other images containing computed *mesh saliency* maps. The majority of values seems to be close to zero, resulting in most of what is visible shaded in black or a very dark blue. The most extreme extent to which this behavior can occur is shown in the figures depicting the third model in section 6.4.3. For more details on why *mesh saliency* - in its originally proposed, unaltered state - does not produce quality results for mechanical objects, see section 6.5.3.

6.5.2 Observation of saliency maps

When considering the average *user saliency* maps, four major observations are to be described. First, while *mesh saliency* maps still look very different, a general common theme as to which parts are more important and which are not, is there - especially when it comes to contours, geometrical details and *characteristic* parts of the objects. For models Bunny and Cow, a strong emphasis on the entire head part is noticeable, with a much stronger variation in values in the *user saliency* map. Again, a resemblance in general can be noted, but it is obvious that users had much more complex criteria for their selection than solely local curvature differences computed for on a per-vertex basis. This results in prominent differences in saliency maps, hardly possible to describe solely by numbers and ratios. Distinct patches of higher perceived importance by users contradict, in part, the results of *mesh saliency*. For example, the belly region of the second model (Cow) was marked as *important* by about 40% of users, whereas it is entirely *uninteresting* according to *mesh saliency*.

Second, within *user saliency* maps, seemingly zero-saliency patches can be found in the middle of cohesive parts of objects that seem to have been selected by almost 100% of participants. This behavior is especially prominent, for example, in the ear region of model one (Bunny) as well as the eye and snout part of model two (Cow). Model one lacks the perfect axial symmetry of models two and three, which might, in part, explain stronger variations in user selections but still, this behavior being caused by vague, *lazy* user selections is not highly believable. However, observing the *user saliency* map of model two, reveals that the small, seemingly zero-saliency *spot* only appears on one side of the head. This is further reason to assume this behavior is not caused by users. As described in section 4.3.4, I took steps trying to ensure that the *selection application's* implementation works correctly as much as possible, however it has not been verified on a conceptual, mathematical level. Minor bugs and weaknesses of the implementation can not be ruled out as a cause for this behavior. Lastly, the visual feedback to what is currently selected and what is not, might be another factor contributing to this behavior. If shading all currently selected vertices was not clear or precise enough, caused by interpolation between extremely close vertices, this might be another factor causing these zero-saliency spots.

Third, *user saliency* maps seem to very much more strongly regarding the range of values. Put simply, *mesh saliency* maps seem to only contain patches that are not interesting at all, patches that are a little interesting and patches that are somewhat interesting. *User saliency* maps on the other hand, seem to make full use of the scale, ranging from *not important at all* (selected by zero participants) to *of essential importance* (selected by virtually every

participant). At this point, it is worth repeating that both *mesh* and *user saliency* values are normalised to a decimal between 0 and 1, in relation to their respective maximum values. Based on the figures in section 6.4, it is fair to assume that vertices with maximum values for *mesh saliency* are very rare and distributed sparsely over the entire geometry. There are no large, cohesive regions of very high average importance as it seems. There are scattered almost-white dots, seemingly randomly distributed over the object. This is highly dependent on the object in question and how the model was built, might provide insights on this behavior. Whether an object was created with and exported from construction software meant for a certain type of object, modeled *by hand* with ordinary 3D modeling software or 3D scanned, can vastly affect the result of processing via *mesh saliency*.

Fourth, *user saliency* maps, for the most part, show clearly distinct patches describing parts of *essential importance* (selected by virtually every participant). This is most obvious for model two (Cow). Consider the horns, the eyes and the snout and how they are shaded according to their *user saliency*. Extremely high *user saliency* values lead the respective parts of the object being shaded almost 100% white in the figures in section 6.4. It is obvious that some of these patches can be described by clear fairly clear borders.

6.5.3 Mesh saliency with mechanical objects

During evaluation of the data gathered in the course of this work, the observation the nature of the 3D object at hand is of crucial influence on whether results of processing it using *mesh saliency* are convincing or seem intuitive. As described in section 5.2, the three models at hand were chosen with the declared intention of getting data that can be extended to a large range of types of objects. Model three (P51) represents the type of mechanical, *man-made* objects and is thus vastly different in its structure than objects one and two. Consider figure 6.23 which shows a part of model three in detail. It is a screenshot, taken directly from the visual output of `climberpi's` implementation [clm], displayed in an openGL window. Note that here, the mapping of resulting *mesh saliency* values using $\sigma_n = 1\epsilon$, is not normalised to the maximum saliency value.

The mapping of colors to *mesh saliency* values in figure 6.23 is similar to that used for all figures shown in section 6.4. The closer the color is to white, the higher *mesh saliency* value is at any given vertex. Again, note that values are not normalised here. The fact that, despite that, most of the object is still shaded nearly black and only small details have seem to be of any importance according to the process, shows that the originally proposed, unaltered version of *mesh saliency* might not be suitable for mechanical objects. Again, this is highly dependent of the object in question and mechanical objects, by nature, can hardly be attributed any common structural pattens.

As mentioned before, the measure of difference suggested in section 6.5.1 lacks expressiveness. However, the fact that both raw and weighted overall difference ratios amount to about 0.38 - which is the highest number among the three objects - might indicate that *mesh saliency* has to be greatly altered to produce intuitive results on mechanical objects. Furthermore, even the *user saliency* map for model three was not as distinct as for models one and two. Participants in the user study seemed to have many more, much different interpretations of what parts of an aircraft are *important* than they had regarding animals. So the fact that not much of a statement, other than *saliency maps for mechanical objects will differ stronger than for natural objects* can be made, is not caused by *mesh saliency* possibly not being perfectly suitable for mechanical objects alone.

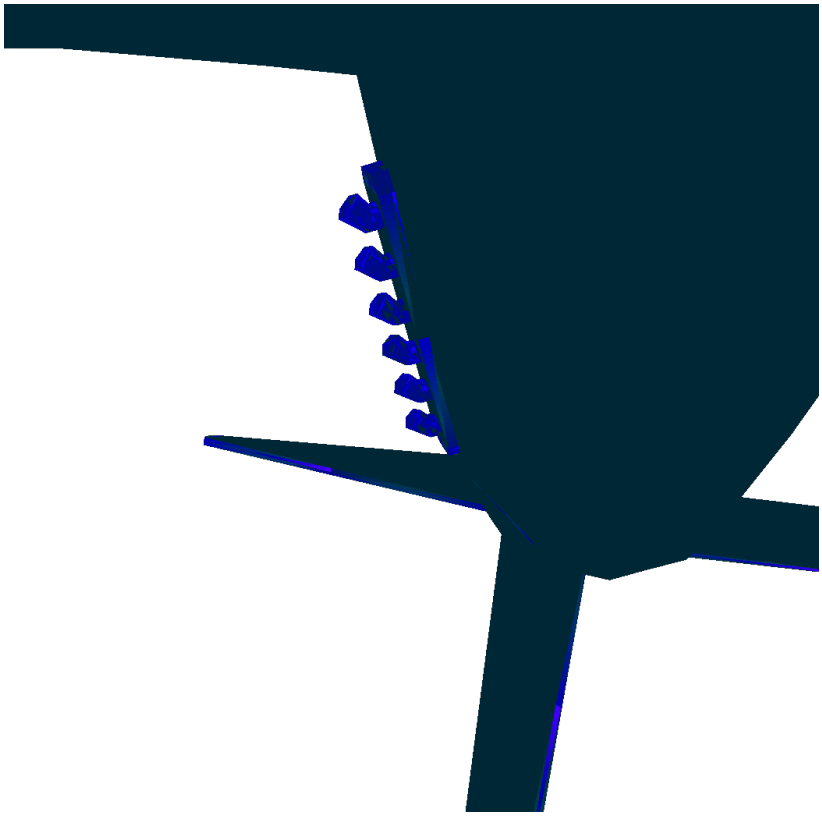

Figure 6.23: Perspective detail view on object three (P51)

7 Conclusion and future work

This section is a contains a brief summery of chapter 6 as well as an outlook on what can be done now to further research this problem. During this work, lots of data was collected but evaluation merely began on a surface level, due to other major tasks taking up so much of the workload.

7.1 Insights and conclusions

Difference ratios expressed via the measure of difference as suggested in section 6.5.1 are fairly low, ranging from about 0.23 for model one, to 0.27 for model two, up to 0.38 for model three. However, the expressiveness of this ratio is dubious. From a basic, visual comparison of computed *mesh saliency* and *user saliency* maps for model three, one might conclude that the two results are almost nothing alike. Based on processing via *mesh saliency*, almost every vertex of the mode seems to have a near-zero importance ranking. The exception, the only parts that stand out in the *mesh saliency map* for this very mechanical object, are fairly small geometrical details, exhaust pipes in this specific case. This is evident even when mapping non-normalised *mesh saliency* values to the object, as shown in figure 6.23. Again, it is not plausible that this single number, this measure of distance, cannot serve as the basis for a quantitative statement on how much average user selection differs from what can be regarded *important* through computational processing of the 3D data. It is to be seen as a suggestion, a possible starting point for more sophisticated ways of quantifying differences in saliency maps.

However, based on trivial observations of the resulting *importance* and difference maps, the measure of difference suggested in this work might still be useful for a rough estimation of which objects are more likely to have greater differences between *mesh* and *user saliency* value mappings. Depending on how strongly ratios computed via this method differ from one another, perhaps even a ranking of objects is possible in terms of which of them are most likely to show the most / the least overall differences in saliency values, in relation to one another.

User saliency and *mesh saliency* maps show visual similarities for models one and two, as one might have expected. Especially when it comes to finding contours, the results were close, even though the visual representation of computed *mesh saliency* maps was vastly different than to that of *user saliency* and difference maps. For all three objects, users seemed to put a heavier emphasis on parts that might possibly be described as *characteristic* to them. These parts may at times contradict the result of processing the object via *mesh saliency*. This seems like a likely outcome of the user study, as users were encouraged to select parts of the objects according to their own, personal interpretation of the term *interesting*. Human cognition and the process of identifying objects is based on recognising larger shapes rather than point-wise curvature. This might, in part, explain the emphasis on *characteristic* parts but diving into human neurology any more would go beyond the scope of this work. This observation is to be interpreted merely as one of many possible aspects to understanding

what users - especially in an immersive VR environment - find *interesting* when presented 3D models.

One of the main problems with the measure of difference is that all vertex-wise saliency values are normalised, i.e. divided by their respective maximum value, before getting subtracted from one another. This was done with the goal in mind of getting one percentage ratio which can describe the overall difference of importance distributions. As a result, expressiveness, especially with *mesh saliency* values gets lost this way. This is caused by maximum *mesh saliency* values being very sparse and, seemingly, not surrounded by other values that also have high computed saliency values.

The result of normalising *saliency* values, in combination with the fact that processing via *mesh saliency* seems to produce low values for the vast majority of overall vertices, is that the amount of almost zero-saliency values is high for each object. On the other hand, there are many parts of objects which are also of no interest to users in VR, which leads to a great amount of *user saliency* being zero as well. As a consequence, a great number of raw differences also amount to almost zero. So, put bluntly, for each high difference value, there might be so many near-zero values that the one high difference gets completely *watered* and ultimately loses its meaning through division by the number of vertices total.

Evaluation of model three (P51), being a mechanical object, is insightful in another regard. It further impairs the plausibility of the measure of difference, as its *user* and *saliency map* look almost nothing alike, see section 6.4.3. It also might imply that the original, unaltered version of *mesh saliency* as described in [LVJ05] is not suitable for mechanic objects. However, even contemplating general rules as to what shapes and which levels of detail are likely in mechanic objects in general, is a challenge unfathomable in its scope and complexity.

Models one and two showed a much stronger variation in saliency values as well as sometimes very clearly distinct patches of *essential importance*. Based on the data gathered during the user study, it is safe to assume that virtually every participant selected, for example, the region around the eyes of model two (Cow) as well as large portions of the ears of model one (Bunny). These highly interesting regions seemingly having such distinct borders might be a resourceful contribution to efforts of making the model *mesh saliency* more coherent with what the average human beholder of 3D data finds *interesting*. The effects of selection happening in an immersive virtual reality environment is to be further examined as well, since the feeling of standing in front an actual, live animal might greatly influence how people perceive it. This begs the question whether immersion in VR can be measured, at which level such effects become noticeable and how strong they scale with the quality of VR soft- and hardware.

7.2 Future work

This work is a first step to examining differences between what human users and mathematical procedures identify as *interesting* parts of 3D data. Collecting data, suggesting a set of basic requirements to software that allows such data to be gathered, and a quick estimation as to whether there are significant differences or not is the main ambition of it. As a result, there is plenty of promising follow-up work. This section describes three possible branches of expanding further into the higher level questions at hand, motivating this work. In addition, unused data gathered during this work and how it can potentially be used is described.

7.2.1 Improving measure of difference

As discussed before, the measure of difference suggested in section 6.5.1 has lots of potential ways to improve upon it. As it is described in this work, it might still be useful as a ranking of which objects are likely to have greater overall differences of saliency values than others but the following aspects can and should be developed further.

- Normalising both *mesh* and *user saliency* values (dividing them by the respective maximum value) causes results to get *watered* by an abundance of near-zero values. Finding a way around this, keeping the expressiveness of importance values, is an important first step towards an improved measure of difference.
- The idea of boiling down overall differences of 3D data to one single percentage ratio is meant to provide an estimation of differences as quickly and easily understandable as possible. However, a way of defining multiple difference ratios is certainly needed. This task can be tackled in multiple ways. Subdividing the whole range of computed values to discrete steps and computing difference values for each range is one of many trivial suggestions as to how this can be achieved.
- Pattern recognition techniques can provide a massive boost to expressiveness of any measure of difference in this context. If contiguous patches of vertices with similar saliency values can be identified, difference ratios for each such patch can be computed. Whether the same, and how many, such patches even get identified through user input would be the next, trivial step.

7.2.2 Altering variables

A lot of variables during the user study conducted for this work were set and not altered with the intention of getting similar information about similar data that is easy to compare. However, many of them can and should be altered to achieve further insights.

- The selection radius of the *selection application* is computed based on the structure of the object. To be precise, a combination of that structure and the essential parameters passed to the octree structure which indexes the loaded 3D information, is the basis for said radius. Again, it is 95% the size of the smallest possible subtree node, determined by `maxSplitDepth`. For this reason, this radius is of essential importance to how the selection process feels. Trying other combinations of determining this radius, maybe even allowing users to dynamically alter it during selecting parts of an object, is not only a desirable functionality of the application but is also certain to provide more insightful and rich data.
- The size, shape and color of the selection target has virtually unlimited possibilities. Whether it is shown at all, how it is shown and its behavior when passing through the projected surface of a model are promising aspects to try and use other rules for.
- The *selection application's* visual design is described in section 3.2. Objects can be shaded differently, textures can be used, dynamic light effects can be used and much more can be done to alter the users immersion and experience within the scene.

- The upper time limit for users to finish their selection process on a single object was five minutes. This was chosen to prevent the user study from being too long and frustrating the user. However, it might be set to a different value, perhaps based on complexity of the object of hand.

7.2.3 More extensive user study

A total number of 32 users took part in the user study conducted for this work. A lot of its conditions were set from the beginning and remained unchanged for its entire duration. It can be greatly expanded in one or more of the following ways.

- Get more users to acquire a richer pool of data. Best case, multiple hundreds of user inputs are certain to generate more refined, more generally applicable results.
- Other types of immersive VR might get tested and results can be compared with regards to the type of input hardware they were created with. In this work, the user study took place in a multi-wall projection installation. The first thing might come to mind as an alternative might be a head-mounted display VR setup.
- Collecting more types of data is certain to allow for more thorough conclusions to be drawn from gathered data. Users could be encouraged to give more qualitative, descriptive feedback. Different kinds of *important* regions might be determined and more questions, directly regarding the objects used might be included in a questionnaire.

7.2.4 Unused data

In addition to vertex IDs in user selections - the central information needed for the comparison of overall differences as designed in this work - a lot of other data was collected during the user study. Some of it is only mentioned in this work, other is not yet discussed due to necessary limitations to the scope of this work. The following types fo information are available.

- As briefly stated in section 3.3.2, user selection logfiles contain timestamps (in seconds since the application was started) next to each logged vertex ID. This timestamp expresses when the respective vertex was last added to or removed from the selection. While information on which vertices were removed is an interesting piece of information in itself, this timestamp might be used to determine which parts of the object were selected first and what type of implications on their perceived importance this might have.
- Demographic data about population of the user study as well as interest in technology are anonymously saved for each participant. It is reasonable to assume things like personal interest in technology, profession and maybe age contribute massively to how satisfactory using the *selection application* is for the user. Further examining this is another possible piece of follow-up work to this thesis.

I would like to thank

My advisor Markus Wiedemann for his ongoing, patient and professional help during the course of this work. For helping me put things back in perspective whenever I felt overwhelmed by the workload in front of me and for making the most challenging aspects of this work seem feasible.

His colleagues working at the Leibniz Supercomputing Centre, Thomas Odaker, Rainer Oesman and Dr. Rubén Garcia Hernández for letting me be a part of their dedicated, kind and motivational working environment.

My friend Christian Blank for motivation and helpful conversations, both personal as well as related to my studies. My friend Sebastian Dietl for motivation, occasional accommodation and making the Garching campus seem more homely. My friend Amy Eichorn for proof-reading and giving me a chance to practise writing in English.

My friends Christa Bichlmayer, Tobias Bjarsch, Martin Erthel, Robert Hetzner and Manuel Miksche for their support and motivation throughout the duration of this work.

The Ludwig Maximilian University of Munich, Prof. Dr. Dieter Kranzelmüller and the MNM-team for granting me the opportunity to conduct this work.

My parents and family for their continuous, unconditional love and support.

List of Figures

1.1 Schematic depiction of how the *selection application* works from a user perspective . . . 3

2.1 A model and its computes mesh saliency map. Published by Lee *et al.* [LVJ05]. Bright colors (yellow and red) indicate high saliency values for their respective vertices, dark colors (shades of blue) indicate low values. . . . 6

3.1 Notation in .OBJ format . . . 12
3.2 An octree structure indexing five vertices. Left: 3D view, right: tree structure; 2017, https://developer.apple.com/documentation/ . . . 13
3.3 A multi-scale mesh saliency map for an object including color scale. Published by Nouri *et al.* [NCL15] . . . 18
3.4 Graphic representation of the weighting function in relation to the *unweighted* difference values (straight, black line) . . . 20

4.1 Parent node (checkered) highlighted child nodes. Left: 000, right: 101 . . . 31
4.2 depiction of a simple `ocTree`. Left: 3D view, step-wise representation of the splitting process (downwards from the top). right: final 2D representation of the tree's structure . . . 33
4.3 Determing vertices to check and final set of selected vertices . . . 34
4.4 A basic vertex selection process . . . 39
4.5 A fairly complex query to the octree structure of the selection application . . . 40
4.6 A test file used during implementation, perspective view. Its bounding box and edges are depicted . . . 41

5.1 The hardware components available at the V2C. Left: A pair of active, stereoscopic shutter glasses. Right: The main input device, referred to as *wand* . . . 46
5.2 The selection process using the available hardware at the V2C . . . 47
5.3 A real selection process . . . 48
5.4 The 3D objects used for the user study . . . 49

6.1 Age distribution of participants in the user study . . . 53
6.2 Working situation of participants in the user study . . . 54
6.3 Feedback on the application gathered from the questionnaire. *No comment* answers are not considered. . . . 55
6.4 The uniform scale used for figures in this section. . . . 56
6.5 Normalised saliency maps for the first model, orthographic front view. Left: *mesh saliency*, right: *user saliency* . . . 56
6.6 Normalised difference maps for the first model, orthographic front view. Left: *unweighted differences*, right: *weighted differences* . . . 57

6.7 Normalised saliency maps for the first model, orthographic side view. Left: *mesh saliency*, right: *user saliency* 57
6.8 Normalised difference maps for the first model, orthographic side view. Left: *unweighted differences*, right: *weighted differences* 58
6.9 Normalised saliency maps for the first model, orthographic top view. Left: *mesh saliency*, right: *user saliency* 58
6.10 Normalised difference maps for the first model, orthographic top view. Left: *unweighted differences*, right: *weighted differences* 58
6.11 Normalised saliency maps for the second model, orthographic front view. Left: *mesh saliency*, right: *user saliency* 59
6.12 Normalised difference maps for the second model, orthographic front view. Left: *unweighted differences*, right: *weighted differences* 59
6.13 Normalised saliency maps for the second model, orthographic side view. Left: *mesh saliency*, right: *user saliency* 60
6.14 Normalised difference maps for the second model, orthographic side view. Left: *unweighted differences*, right: *weighted differences* 60
6.15 Normalised saliency maps for the second model, orthographic top view. Top: *mesh saliency*, bottom: *user saliency* 60
6.16 Normalised difference maps for the second model, orthographic top view. Top: *unweighted differences*, bottom: *weighted differences* 61
6.17 Normalised saliency maps for the third model, orthographic front view. Top: *mesh saliency*, bottom: *user saliency* 62
6.18 Normalised difference maps for the third model, orthographic front view. Top: *unweighted differences*, bottom: *weighted differences* 62
6.19 Normalised saliency maps for the third model, orthographic side view. Top: *mesh saliency*, bottom: *user saliency* 63
6.20 Normalised difference maps for the third model, orthographic side view. Top: *unweighted differences*, bottom: *weighted differences* 63
6.21 Normalised saliency maps for the third model, orthographic top view. Left: *mesh saliency*, right: *user saliency* 64
6.22 Normalised difference maps for the third model, orthographic top view. Left: *unweighted differences*, right: *weighted differences* 64
6.23 Perspective detail view on object three (P51) 68

Bibliography

[AK] Paul Atchley and Arthur F Kramer. Attentional control within 3-d space.

[ASP] ASSIMP - Open Asset Import Library official website. http://www.assimp.org/.

[BB14] Oliver Baus and Stéphane Bouchard. Moving from virtual reality exposure-based therapy to augmented reality exposure-based therapy: a review. *Frontiers in human neuroscience*, 8:112, 2014.

[bun] bunny stanford computer graphics laboratory - weblink. `https://graphics.stanford.edu/`. Accessed: 2017-06-07.

[CHR13] Arridhana Ciptadi, Tucker Hermans, and James M Rehg. An in depth view of saliency. Georgia Institute of Technology, 2013.

[clm] mesh saliency by climberpi - github repository weblink. `https://github.com/climberpi/Mesh-Saliency`. Accessed: 2017-06-07.

[CMS98] Paolo Cignoni, Claudio Montani, and Roberto Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22(1):37–54, 1998.

[cow] cow mit computer science and artificial intelligence laboratory - weblink. `https://groups.csail.mit.edu/graphics/classes/6.837/F03/models/`. Accessed: 2017-06-07.

[CSPF12] Xiaobai Chen, Abulhair Saparov, Bill Pang, and Thomas Funkhouser. Schelling points on 3d surface meshes. *ACM Transactions on Graphics (TOG)*, 31(4):29, 2012.

[DCG12] Helin Dutagaci, Chun Pan Cheung, and Afzal Godil. Evaluation of 3d interest point detection techniques via human-generated ground truth. *The Visual Computer*, 28(9):901–917, 2012.

[dRKH+14] Sandrine de Ribaupierre, Bill Kapralos, Faizal Haji, Eleni Stroulia, Adam Dubrowski, and Roy Eagleson. Healthcare training enhancement through virtual reality and serious games. In *Virtual, Augmented Reality and Serious Games for Healthcare 1*, pages 9–27. Springer, 2014.

[GH97] Michael Garland and Paul S Heckbert. Qslim simplification software, 1997.

[GLE] GLEW - The OpenGL Extension Wrangler Library official website. `http://glew.sourceforge.net/`. Accessed: 2017-06-07.

[GLU] GLUT - The OpenGL Utility Toolkit official website. `https://www.opengl.org/resources/libraries/glut/`. Accessed: 2017-06-07.

[HHO05] Sarah Howlett, John Hamill, and Carol O'Sullivan. Predicting and evaluating saliency for simplified polygonal models. *ACM Transactions on Applied Perception (TAP)*, 2(3):286–308, 2005.

[KG03] Youngihn Kho and Michael Garland. User-guided simplification. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 123–126. ACM, 2003.

[Kos03] AF Koschan. Perception-based 3d triangle mesh segmentation using fast marching watersheds. In *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE Computer Society Conference on*, volume 2, pages II–II. IEEE, 2003.

[KU87] Christof Koch and Shimon Ullman. Shifts in selective visual attention: towards the underlying neural circuitry. In *Matters of intelligence*, pages 115–141. Springer, 1987.

[LKC07] Sungkil Lee, Gerard Jounghyun Kim, and Seungmoon Choi. Real-time tracking of visually attended objects in interactive virtual environments. In *Proceedings of the 2007 ACM symposium on Virtual reality software and technology*, pages 29–38. ACM, 2007.

[LPP15] Quang Tuan Le, Akeem Pedro, and Chan Sik Park. A social virtual reality based construction safety education system for experiential learning. *Journal of Intelligent & Robotic Systems*, 79(3-4):487, 2015.

[LVJ05] Chang Ha Lee, Amitabh Varshney, and David W Jacobs. Mesh saliency. In *ACM transactions on graphics (TOG)*, volume 24, pages 659–666. ACM, 2005.

[MEC14] Stefan Marks, Javier E Estevez, and Andy M Connor. Towards the holodeck: fully immersive virtual reality visualisation of scientific and engineering data. In *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*, pages 42–47. ACM, 2014.

[MF14] Alma S Merians and Gerard G Fluet. Rehabilitation applications using virtual reality for persons with residual impairments following stroke. In *Virtual Reality for Physical and Motor Rehabilitation*, pages 119–144. Springer, 2014.

[MGC+14] Zahira Merchant, Ernest T Goetz, Lauren Cifuentes, Wendy Keeney-Kennicutt, and Trina J Davis. Effectiveness of virtual reality-based instruction on students' learning outcomes in k-12 and higher education: A meta-analysis. *Computers & Education*, 70:29–40, 2014.

[MJSS02] David W Mizell, Stephen P Jones, Mel Slater, and Bernhard Spanlang. Comparing immersive virtual reality with other display modes for visualizing complex 3d geometry. *University College London, technical report*, 2002.

[Mun07] Rodrigo Barni Munaretti. Perceptual guidance in mesh processing and rendering using mesh saliency. 2007.

[NCL15] Anass Nouri, Christophe Charrier, and Olivier Lézoray. Multi-scale mesh saliency with local adaptive patches for viewpoint selection. *Signal Processing: Image Communication*, 38:151–166, 2015.

[NNVL+16] Minh-Tu Nguyen, Hai-Khanh Nguyen, Khanh-Duy Vo-Lam, Xuan-Gieng Nguyen, and Minh-Triet Tran. Applying virtual reality in city planning. In *International Conference on Virtual, Augmented and Mixed Reality*, pages 724–735. Springer, 2016.

[octa] gkoctree - Apple Developer Documentation weblink. `https://developer.apple.com/documentation/gameplaykit/gkoctree`. Accessed: 2017-22-09.

[Octb] Octree - Wikipedia weblink. `https://en.wikipedia.org/wiki/Octree`. Accessed: 2017-06-07.

[ODK99] Kathleen M O'craven, Paul E Downing, and Nancy Kanwisher. fmri evidence for objects as the units of attentional selection. *Nature*, 401(6753):584–587, 1999.

[OF+15] Michela Ott, Laura FREINA, et al. A literature review on immersive virtual reality in education: state of the art and perspectives. In *Conference proceedings of eLearning and Software for Education(eLSE)*, number 01, pages 133–141. Universitatea Nationala de Aparare Carol I, 2015.

[OHH+15] Jivka Ovtcharova, Polina Häfner, Victor Häfner, Jurica Katicic, and Christina Vinke. Innovation braucht resourceful humans aufbruch in eine neue arbeitskultur durch virtual engineering. In *Zukunft der Arbeit in Industrie 4.0*, pages 111–124. Springer, 2015.

[Ope] OpenGL - The Industry Standard for High Performance Graphics official website. `https://www.opengl.org/`. Accessed: 2017-06-07.

[P51] P-51 Mustang cadnav.com - weblink. `http://www.cadnav.com/3d-models/model-37216.html`. Accessed: 2017-06-07.

[PPW97] Randy Pausch, Dennis Proffitt, and George Williams. Quantifying immersion in virtual reality. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 13–18. ACM Press/Addison-Wesley Publishing Co., 1997.

[PZV11] Ekaterina Potapova, Michael Zillich, and Markus Vincze. Learning what matters: combining probabilistic models of 2d and 3d saliency cues. In *International Conference on Computer Vision Systems*, pages 132–142. Springer, 2011.

[QTIHM06] Wen Qi, Russell M Taylor II, Christopher G Healey, and Jean-Bernard Martens. A comparison of immersive hmd, fish tank vr and fish tank with haptics displays for volume visualization. In *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*, pages 51–58. ACM, 2006.

[SDP+09] Beatriz Sousa Santos, Paulo Dias, Angela Pimentel, Jan-Willem Baggerman, Carlos Ferreira, Samuel Silva, and Joaquim Madeira. Head-mounted display versus desktop for 3d navigation in virtual reality: a user study. *Multimedia Tools and Applications*, 41(1):161, 2009.

[set] set - cplusplus.com reference weblink. http://www.cplusplus.com/reference/set/set/. Accessed: 2017-06-07.

[SG01] Eric Shaffer and Michael Garland. Efficient adaptive simplification of massive meshes. In *Proceedings of the conference on Visualization'01*, pages 127–134. IEEE Computer Society, 2001.

[Sha08] Ariel Shamir. A survey on mesh segmentation techniques. In *Computer graphics forum*, volume 27, pages 1539–1556. Wiley Online Library, 2008.

[SJRT13] R Stouffs, P Janssen, S Roudavski, and B Tunçer. What is happening to virtual and augmented reality applied to architecture? In *Conference on Computer-Aided Architectural Design Research in Asia (CAADRIA 2013)*, volume 1, page 10, 2013.

[SM14] Alcínia Z Sampaio and Octávio P Martins. The application of virtual reality technology in the construction of bridge: The cantilever and incremental launching methods. *Automation in construction*, 37:58–67, 2014.

[SSK+05] Vitaly Surazhsky, Tatiana Surazhsky, Danil Kirsanov, Steven J Gortler, and Hugues Hoppe. Fast exact and approximate geodesics on meshes. In *ACM transactions on graphics (TOG)*, volume 24, pages 553–560. Acm, 2005.

[Ste92] Jonathan Steuer. Defining virtual reality: Dimensions determining telepresence. *Journal of communication*, 42(4):73–93, 1992.

[Tau95] Gabriel Taubin. Estimating the tensor of curvature of a surface from a polyhedral approximation. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 902–907. IEEE, 1995.

[v2c] V2C official website. https://www.lrz.de/services/v2c_en/. Accessed: 2017-06-07.

[VCSF16] Luis Valente, Esteban Clua, Alexandre Ribeiro Silva, and Bruno Feijó. Live-action virtual reality games. *arXiv preprint arXiv:1601.01645*, 2016.

[vec] vector - cplusplus.com reference weblink. http://www.cplusplus.com/reference/vector/vector/. Accessed: 2017-06-07.

[WAB93] Colin Ware, Kevin Arthur, and Kellogg S Booth. Fish tank virtual reality. In *Proceedings of the INTERACT'93 and CHI'93 conference on Human factors in computing systems*, pages 37–42. ACM, 1993.

[WCAH+16] Robert E Wendrich, Kris-Howard Chambers, Wadee Al-Halabi, Eric J Seibel, Olaf Grevenstuk, David Ullman, and Hunter G Hoffman. Hybrid design tools in a social virtual reality using networked oculus rift: a feasibility study in remote real-time interaction. In *ASME 2016 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pages V01BT02A042–V01BT02A042. American Society of Mechanical Engineers, 2016.

[WL10] Jin Wei and Yu Lou. Feature preserving mesh simplification using feature sensitive metric. *Journal of Computer Science and Technology*, 25(3):595–605, 2010.

[WSZL13] Jinliang Wu, Xiaoyong Shen, Wei Zhu, and Ligang Liu. Mesh saliency with global rarity. *Graphical Models*, 75(5):255–264, 2013.

[ZLSZ12] Yitian Zhao, Yonghuai Liu, Ran Song, and Min Zhang. A saliency detection based method for 3d surface simplification. In *Acoustics, Speech and Signal Processing (ICASSP), 2012 IEEE International Conference on*, pages 889–892. IEEE, 2012.

[Zyd05] Michael Zyda. From visual simulation to virtual reality to games. *Computer*, 38(9):25–32, 2005.