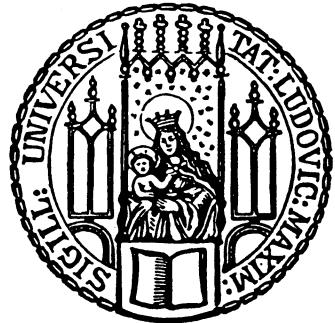


INSTITUT FÜR INFORMATIK
DER LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN



Master's Thesis

**Evaluation of Virtual Reality based
Mesh Saliency Maps**

Richard Metzler

Draft vom September 4, 2017

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 Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 7. Juli 2077

.....
(Unterschrift des Kandidaten)

Abstract

Hier steht eine kurze Zusammenfassung der Arbeit. Sie darf auf gar keinen Fall länger als eine Seite sein, ca. eine drittel bis eine halbe Seite ist optimal.

Contents

1	Introduction	1
2	Related work	3
2.1	Mesh saliency and human perception	3
2.2	Human attention in Virtual Reality	6
3	Concept	9
3.1	Implementing a selection application for the V2C's projection installation	9
3.1.1	Spatial indexing of 3D data	9
3.1.2	Selection process	11
3.2	Conduct user study with the selection application	13
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	20
4	Selection Application	21
4.1	Additional third party libraries	21
4.1.1	OpenGL	21
4.1.2	GLUT	22
4.1.3	GLEW	22
4.1.4	ASSIMP	22
4.2	Relevant class files	23
4.2.1	Object	23
4.2.2	Mesh	23
4.2.3	Octree	23
4.3	Key Features	23
4.3.1	Spatial indexing via Octree	24
4.3.1.1	Root constructor <code>ocTree()</code>	24
4.3.1.2	Subtree constructor <code>ocTree()</code>	25
4.3.1.3	<code>getRootDimensions()</code>	26
4.3.1.4	<code>setDimensions()</code>	26
4.3.1.5	<code>split()</code>	27
4.3.1.6	<code>getNodeByIdentifierArray()</code>	28
4.3.1.7	<code>buildTreeRecursively()</code>	29
4.3.2	User selection	31
4.3.2.1	<code>addVerticesToSelectionByCoordinates()</code>	31
4.3.2.2	<code>removeVerticesFromSelectionByCoordinates()</code>	36
4.3.3	Tracking selection	36

Contents

4.3.4 Testing setup	37
5 Results and discussion	39
5.1 User study data	39
5.2 Interpretation	39
6 Conclusion and future work	41
6.1 Insights and conclusions	41
6.2 Future work	41
List of Figures	43
Bibliography	45

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 plentitude of recent works showing that it bears great potential and positive possible contributions to architecture and construction [SM14], [LPP15], [SJRT13], healthcare 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] etc.

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.

Little has been done so far concerning research on *mesh saliency* on a vertex level in a virtual reality environment. Whether saliency maps computed via known methods affect user behavior immersed in VR scenes at all, or to which extent has not been investigated in a vertex selection scenario. Furthermore, as of now, the effect on perceived visual quality of saliency-based object simplification as well as user behavior when given the opportunity to declare salient regions themselves is barely touched on at all.

For the selection of parts of objects which seem important to beholders, I had access use the Virtual Reality and Visualisation Centres five-sided projection installation at the Leibniz Supercomputing Centre in Munich. 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 separate two images for the spectator - one for each eye. The glasses are equipped with an electromagnetic 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

1 Introduction

the constant worry of stumbling over and accidentally damaging them.

Finally, it is worth noting that the effect on perceived quality of objects in 3D scenes that can possibly be achieved through saliency-based simplification is not limited to virtual reality applications. Long-term goals of efficiency and optimisation will continue to be accompanied by the need for semi-automatic complexity reduction of objects without a great loss of visual appeal, regardless of the type of media they will be presented on. So the best case outcome of this work is to find possible approaches to identifying segments of objects which are of high importance to the average beholder based on the immersive nature of the selection process of said segments.

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 perceived 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 perceptual 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 \mathcal{C} defines a mapping from each vertex of a mesh to its mean curvature $\mathcal{C}(v)$. 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 approach 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 $G(\mathcal{C}(v), \sigma)$ and present the following way of computing it.

2 Related work

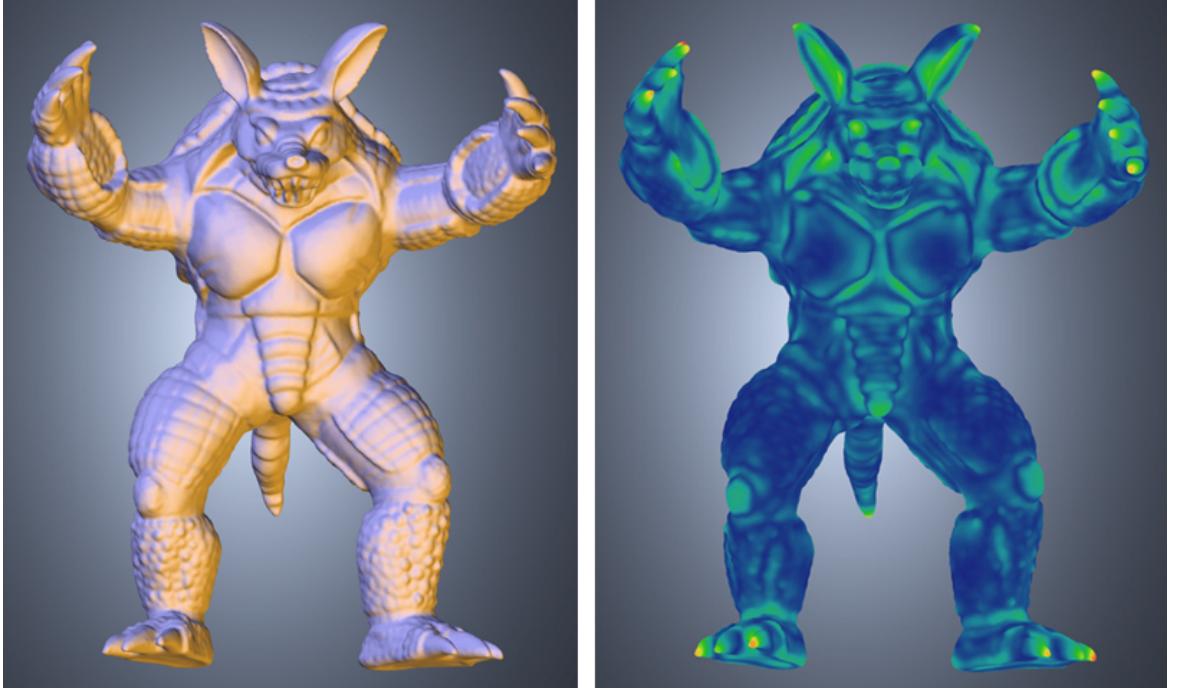


Figure 2.1: A model and its computed mesh saliency map. Published by Lee *et al.* [LVJ05]. Bright colors (yellow and red) indicate high saliency values for their respective vertices, dark colors (shades of blue) indicate low values.

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

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 $\mathcal{S}(v)$ of a vertex v is defined as the absolute difference between the Gaussian-weighted averages, as seen in the formula below.

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

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

2.1 Mesh saliency and human perception

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 computed 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

2 Related work

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 which 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 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

2 Related work

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

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

1. implement a selection application for the V2C's projection installation
 - 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 the V2C's projection installation

One of the main challenges of this work was developing a piece of C++ software that allows users in the five-sided projection installation of the Leibniz Supercomputing Centre to select vertices of 3D objects using the existing soft- and hardware components at hand there. Designing, implementing and adjusting this software to being executable as a multi-threaded client-server application in said projection installation, was a challenging and time-consuming 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

3 Concept

```
# 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

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).

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 containing vertices (lines starting with v), vertex texture coordinates (lines starting with 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 multiple 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 as 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

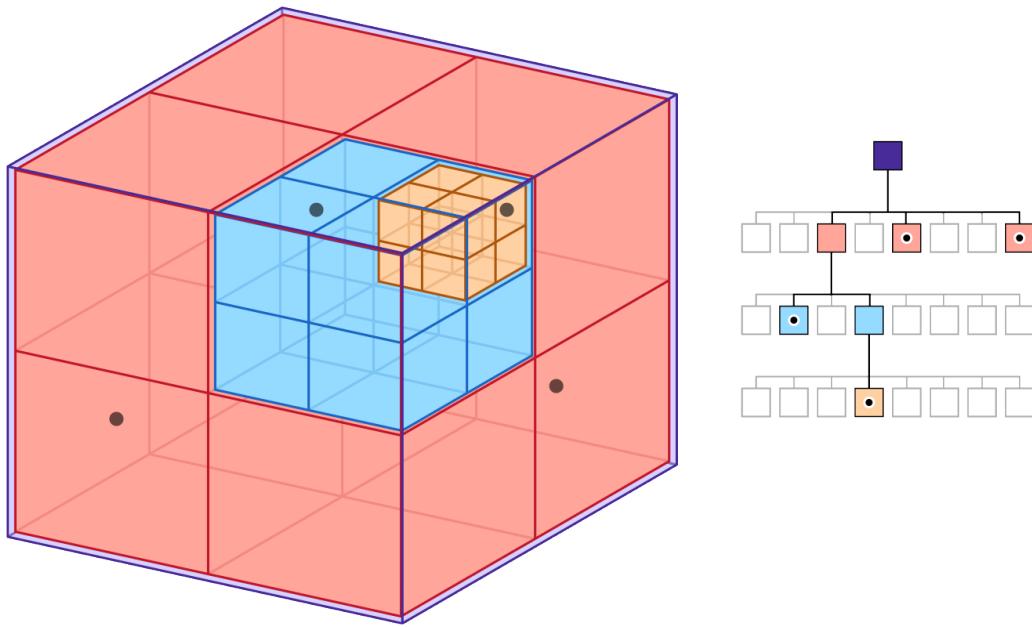


Figure 3.2: An octree structure indexing four vertices. Left: 3D view, right: tree structure; 2017, <https://developer.apple.com/documentation/>

of subtrees.

Figure 3.2 taken from [octa] shows a simple octree structure indexing four vertices, both 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 clearly visualizes how a 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, there are two leaf nodes that hold one vertex each, on level 2 and 3, there is one leaf holding one vertex.

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 was still a leaf. It is now a root no longer and will create 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, I decided to implement the non-balanced version of an octree where this is not the case. The reason for this is that in 3D objects, the data is not evenly 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.

3.1.2 Selection process

To gather data to compare with computer generated mesh saliency maps, tracking user input was needed. In the five-sided projection installation of the [v2c], 3D objects would

3 Concept

be loaded and spatially indexed using octrees as described in the section above. Now, the selection function had to be implemented. Using a handheld input device, a so-called *wand*, as depicted in figure 3.3, 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.

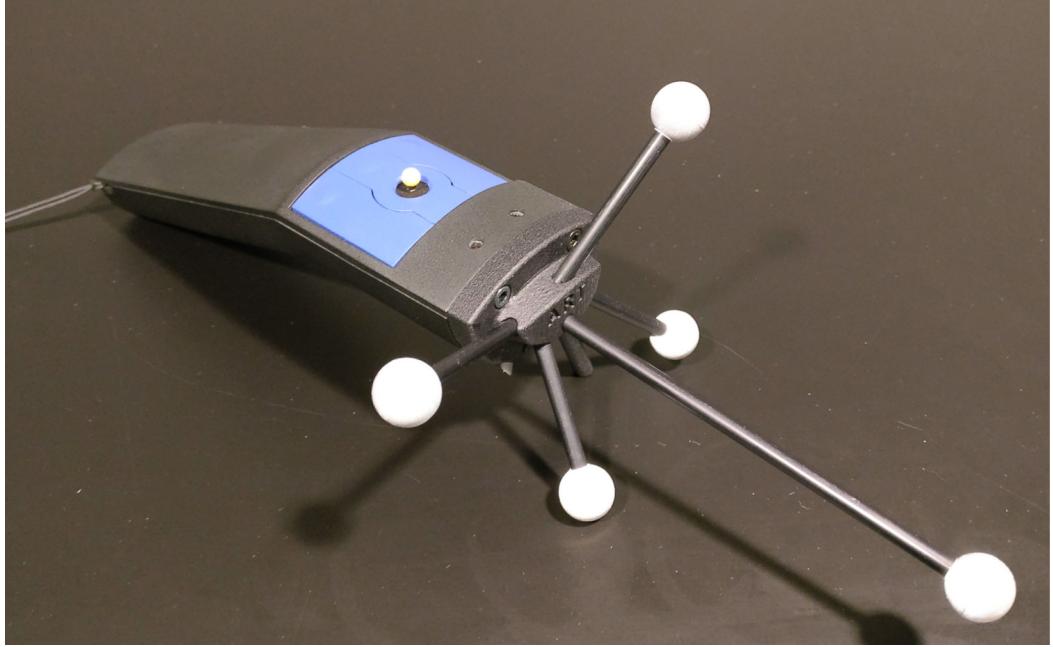


Figure 3.3: The input device used at the V2C

Figure 3.3 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. The two essential functions of the application are mapped to two of the three buttons surrounding the joystick: The (from the perspective of a user holding the *wand*) left button adds vertices near the current position of the wand to the vertex selection and the righthand button removes such vertices from it. The button in the middle does not have a function in this application, as well as the trigger-like fourth button on the button side of the *wand* (not in view in the figure). In accordance with the simplistic structure of the user study, the interaction was 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 separating 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 left button of the *wand*, 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 `glm::vec3` variable, a basic 3D vector type provided by OpenGL's mathematics library. 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 as a `float` value and a reference to a temporary set of `size_t` values called `intermediateSelection`, 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 returned and written to it. If the *add* button was pressed, the content of `intermediateSelection` will be added to another set holding the list of currently selected vertices total. If the *remove* button was pressed, each of the temporarily selected vertices will be 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.

3.2 Conduct user study with the selection application

Another essential part of this work was conducting a user study where all of the aspects described in this chapter so far would be put to use. The goal was to collect data on what parts of 3D objects users found visually interesting or important. Users put on the stereoscopic, trackable glasses depicted in figure 3.4, stepped inside the five-sided projection installation of the [v2c] and were asked to mark regions of the object currently displayed that they considered to be interesting or important. The exact wording of the task given to participants in the user study reads as follows.

Please select parts and regions of the object that

- *you would consider visually interesting or important*
- *you would assume are natural focus points of attention*
- *are vital for identifying the object*

After taking up to five minutes getting familiar with navigating and interaction within the projection installation, as well as selecting and deselecting vertices with the wand (3.3, the main part of the user study began).

Selections of the users would be gathered to compute user-weighted importance maps for the displayed 3D objects. These would be compared to previously computed mesh saliency maps. 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 *wand* as input coordinates. To give users a clear, live feedback on that tracked position, for every frame, a ten-sided, diamond-shaped object was 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

3 Concept



Figure 3.4: The trackable, stereoscopic glasses used at the V2C

of the pre-computed selection radius. Again, 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. I chose not to adapt the size of the ten-sided object representing the position of the wand and keep it the same size for every object with the intention of providing more consistency within the scene, independent of the currently loaded object. This was also a design decision that went 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

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 a 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 another constraint to be considered. I decided to use three objects described table in 3.1 and depicted in figure 3.5.

Figure 3.5 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

3.2 Conduct user study with the selection application

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 3.1: 3D objects used in the study

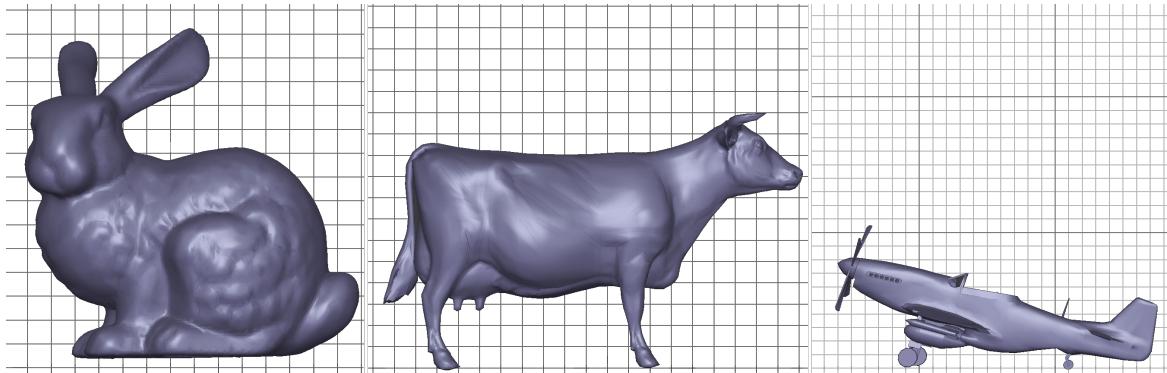


Figure 3.5: The 3D objects used for the user study

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.

I took two steps 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 was given five minutes to get familiar with the navigation and selection workflow. Selections made here were not considered for the results of the userstudy. After that, the three objects were presented in a randomly chosen order for five minutes each. The users were given the tasks stated at the beginning of this section and were free to use as much of the five minutes as they wanted to perform vertex selection and deselection operations.

The five minute time limit was an absolute upper limit for each selection process. Users were free to end selection prematurely whenever they stated they were satisfied with the selection as it is.

Besides the five minute *tutorial phase* - the time users could take to get familiar with the application - and the actual tasks to be fulfilled (as described at the beginning of this section), only one more minor request was asked to be considered by users: An approximate symmetrical selection, meaning that, in cases where it was possible, parts of objects users selected on one side of an object would also be selected on the opposite side. Other than that, no additional information or hints were given. Users were told that there is no *correct* way of to fulfill the selection task. They were encouraged to select whatever they deemed relevant according to the tasks based on their personal judgement.

3.3 Measure of differences

Since a large portion of the workload of this work was the implementation of the selection application itself, as well as customizing it to the point it is executable in the V2C, the comparison of user generated and pre-computed mesh saliency maps was designed to be quick and easy. In this chapter I will describe how I conceptualised 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. 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 summary 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 times it was selected by users in relation to the most-selected vertices.

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

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

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

ω - a threshold value derived from mesh saliency values $S_M(v_1), \dots, S_M(v_j)$ for all vertices $v_1, \dots, v_j \in$ a *proximity set* $V_{\text{proximty}}(v_i)$ of v_i

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

Table 3.2: Sample content of a user selection log file

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, we computed a mesh saliency map using climberpi's implementation of *mesh saliency* [clm], 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 were added to the selection (in seconds since the application was started) in every line, separated by a tab symbol as depicted in table 3.2.

In the simple evaluation tool used for this work, before each logfile is read, a `std::vector<size_t, int>` is created with size equivalent to the number of vertices of the currently considered object. After the last log file is processed, this set *userselections* contains a mapping for each vertex v_i , describing how many users selected it with the `int` value at offset i , which equals the vertex ID represented as a `size_t` value. 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 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 precise 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.

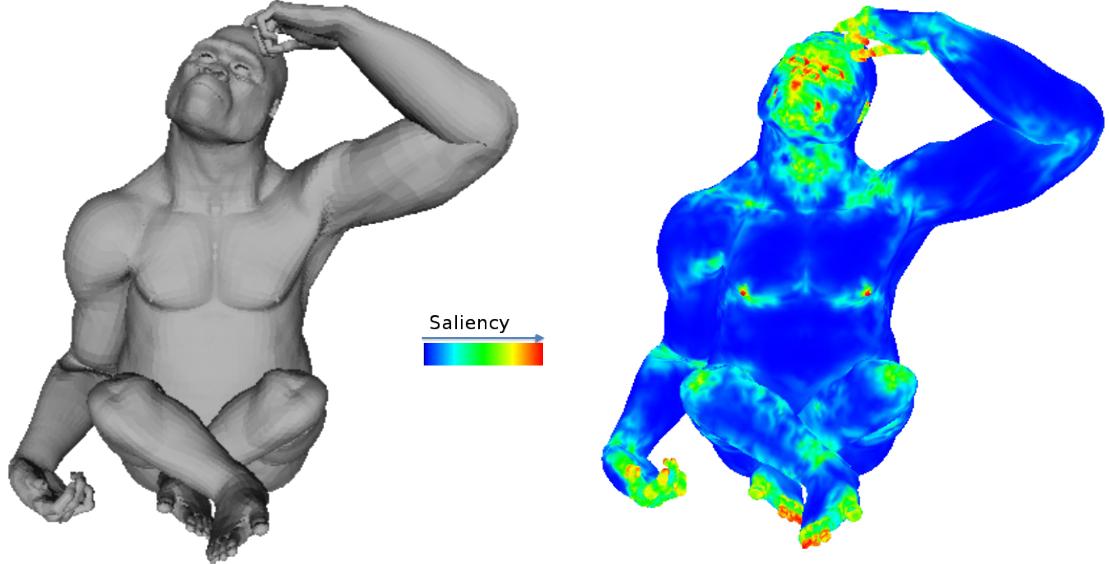


Figure 3.6: A multi-scale mesh saliency map for an object including color scale. Published by Nouri *et al.* [NCL15]

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 was 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 very low *user saliency value* of vertices which lays in a yellow or red shaded area is more interesting than one that is located in a green or blue area, I based the decision which vertex-wise difference ratios would be weighted on a comparison of other vertices surrounding them. 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

I decided to base this simple evaluation process on the assumption that realistic, reliable results are achieved by the model of *mesh saliency*. So I focused on finding deviations within parts and regions of objects that have higher 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, I aimed for 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. So I chose to use an altered, restricted function describing a circle that has its centre at $x = 1.0, y = 0.0$ and radius $r = 1.0$. Figure 3.7 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).

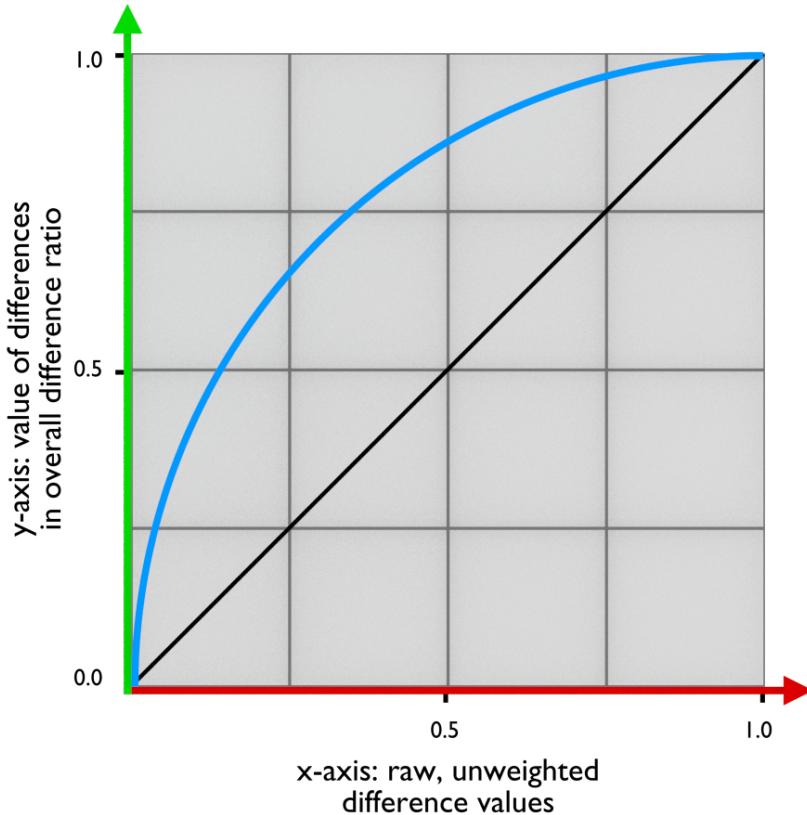


Figure 3.7: Graphic representation of the weighting function in relation to the *unweighted* difference values (straight, black line)

Consider figure 3.7. The straight, black line, being a steady identity function of float values, shows normalised *unweighted* vertex-wise difference values. For such values, both in

3 Concept

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 , I assume it 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 for generating the overall *weighted difference ratio*.

3.3.4 step-wise summary

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

1. Compute the *mesh saliency* map for a given object 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 `std::vector` containers `computed_saliency_values` and `user_saliency_values`.
4. calculate vertex-wise saliency differences and store them in an `std::vector` `differences_unweighted`.
5. Iterate through the list of vertices of the currently considered object. For every such vertex v_i , find vertices $v_1, \dots, v_j \in V_{proximty}(v_i)$ 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, \dots, 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 store 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

In this chapter, I will describe the implemented selection application used for this work. After a rundown of third party requirements and a summary of relevant C++ classes, the description will be further subdivided according to its abstract, key requirements. The goal of this chapter is to describe how the application, especially the Octree[Octb], was designed and implemented. Accordingly, key lines of source code as well as plenty of explanatory comments will be 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 were 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 was 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 was 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 the 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 which can perform fast, basic operations on every vertex of a vertex data input stream. They are fed vertex attribute data, as specified in a VAO by the call of a openGL draw function. In many cases, vertex

shaders will solely compute the position of 3D data on a 2D screen as well as paint it in basic colors. For more complex applications, vertex shaders can also perform super fast manipulation of large amounts of vertex data to achieve transitions in geometry, texture mapping, directions of normals ...

Fragment Shaders Fragments, not to be confused with triangles or faces, are sets of values resulting from rasterisation. They are frequently, informally described as *potential pixels*, meaning the color value they describe is what could be seen on a screen at a specific pixel under certain circumstances. For each fragment, one or more sets of values may be computed, so things like lighting and postprocessing effects, bump maps, antialiasing or simply occlusion by another object may be determining factors as to which one will be rendered to one specific pixel on the output screen. Fragment shaders are used to define how color values for fragments will be selected to be displayed at their respective pixel.

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 handheld controller in the user study was 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 was used 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 will be 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 will cover all the relevant C++ classes used to implement the selection application. Note that these descriptions will only cover the general structure and purpose of these classes within the context of the application. For a more detailed description of the most crucial functions, please refer to Key Features.

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 ASSIMP) 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 processing 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 Key Features.

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 was stored in the object and mesh classes above required an extensive amount of customisation.

4.3 Key Features

This section will describe 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	$p.X \text{ min}$	$p.X \text{ mean}$	$p.Y \text{ mean}$	$p.Y \text{ max}$	$p.Z \text{ mean}$	$p.Z \text{ max}$
001	$p.X \text{ mean}$	$p.X \text{ max}$	$p.Y \text{ mean}$	$p.Y \text{ max}$	$p.Z \text{ mean}$	$p.Z \text{ max}$
010	$p.X \text{ min}$	$p.X \text{ mean}$	$p.Y \text{ min}$	$p.Y \text{ mean}$	$p.Z \text{ mean}$	$p.Z \text{ max}$
011	$p.X \text{ mean}$	$p.X \text{ max}$	$p.Y \text{ min}$	$p.Y \text{ mean}$	$p.Z \text{ mean}$	$p.Z \text{ max}$
100	$p.X \text{ min}$	$p.X \text{ mean}$	$p.Y \text{ mean}$	$p.Y \text{ max}$	$p.Z \text{ min}$	$p.Z \text{ mean}$
101	$p.X \text{ mean}$	$p.X \text{ max}$	$p.Y \text{ mean}$	$p.Y \text{ max}$	$p.Z \text{ min}$	$p.Z \text{ mean}$
110	$p.X \text{ min}$	$p.X \text{ mean}$	$p.Y \text{ min}$	$p.Y \text{ mean}$	$p.Z \text{ min}$	$p.Z \text{ mean}$
111	$p.X \text{ mean}$	$p.X \text{ max}$	$p.Y \text{ min}$	$p.Y \text{ mean}$	$p.Z \text{ min}$	$p.Z \text{ mean}$

Table 4.1: Child node bounding values

4.3.1 Spatial indexing via Octree

As mentioned above, the `octree` class handles spatial indexing and, therefore, 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 was 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 identifier `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 will create a new instance of the class `octree`. Parameters required are 1. a sequence container, such as an array (`std::vector` was used in this work) holding the mesh objects to be spatially indexed, 2. an integer determining the maximum amount of vertices that one leaf node can store, and 3. an integer determining 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, will be printed to the console via `std::cout`. During subsequent creation of subtrees, this parameter will be 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 will create 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 will be 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 was used in this work), 2. An integer determining the level of the parent node, the level of this new subtree will be set to that level plus one, 3. An array of nine float values describing its parents dimensions (together with the *split directions*, this will be 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 determining whether, during the recursive building process of the subtree, information will be 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 was 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 will be 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 will be 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 (especially 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 - will always contain 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	$p.\text{minX}$	$p.\text{meanX}$	$p.\text{meanY}$	$p.\text{maxY}$	$p.\text{meanZ}$	$p.Z \text{ max}$
N_5	$p.\text{meanX}$	$p.\text{maxX}$	$p.\text{meanY}$	$p.\text{maxY}$	$p.\text{minZ}$	$p.\text{meanZ}$

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 have been stored by the calling node up to the point this function was called, will 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.

4 Selection Application

```

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 will be 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, `splitn`. 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 will return 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 will perform 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.

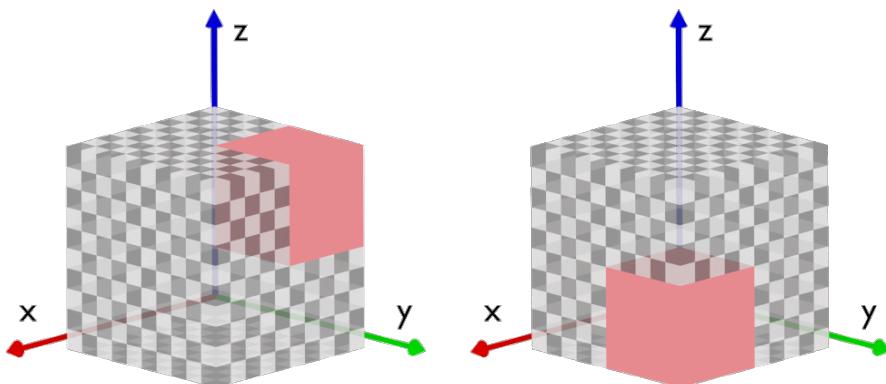


Figure 4.1: Parent node (checkered) highlighted child nodes. Left: 000, right: 101

`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 access to any one desired node within an `ocTree` structure.

4.3.1.7 `buildTreeRecursively()`

A call to this function will cause 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, was used in this work.

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

4 Selection Application

```

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 will consider every vertex its parent node held 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 will 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` will always check every single vertex of the loaded 3D object. However, the higher the level of its subtree nodes, the fewer vertices will have to be checked by those nodes.

Line 13 shows the crucial check whether 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 will call `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 will have 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 create a total of 16 new child nodes, each at level 2.

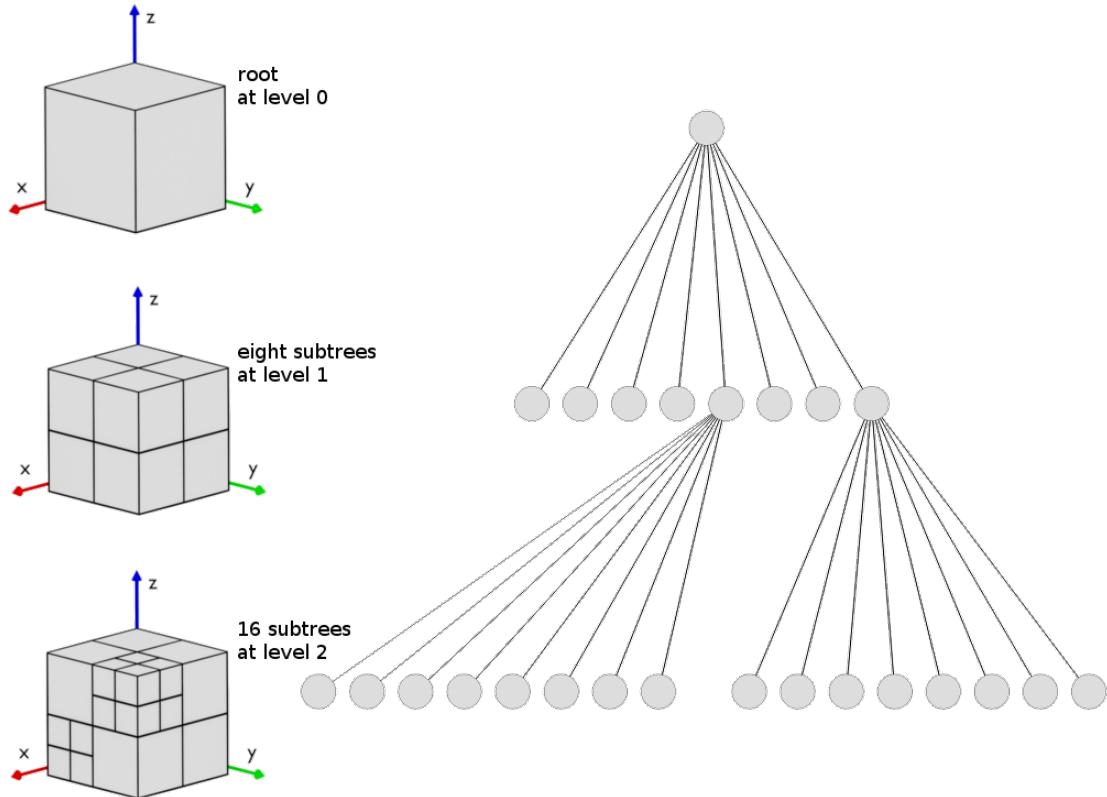


Figure 4.2: depiction of a simple `octree`. Left: 3D view, step-wise representation of the splitting process (downwards from the top). right: final 2D representation of the tree's structure

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 will 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 was using the input device (a `glm::vec3` was used in this work), 2. A `float` value describing the radius around the 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>` was used in this work).

An additional boolean flag determining whether, during the recursive search for the correct node, information will be 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:

4 Selection Application

1. Check if any of the given input point's coordinates are located outside of the bounds of the root of the octTree 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 was 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 will 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.

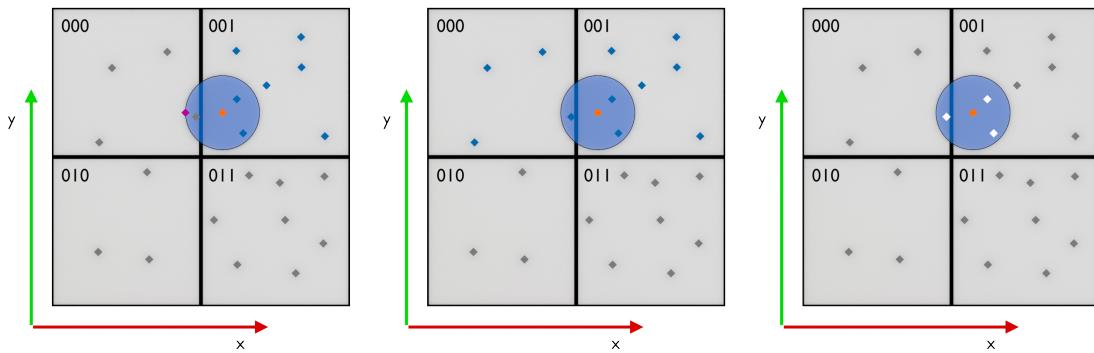


Figure 4.3: Determining vertices to check and final set of selected vertices

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) was 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 $r' = 0$ was 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 was 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 was 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`,

4 Selection Application

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 could 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 evoke 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 was found that they are located within a neighbouring node of N_q (lines 16 and 24). After this, additional 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 radius 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 `octTree` 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.

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

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 minimum 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 will be 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`

4 Selection Application

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 modify 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 will be 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 was 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 handheld 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 was 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.

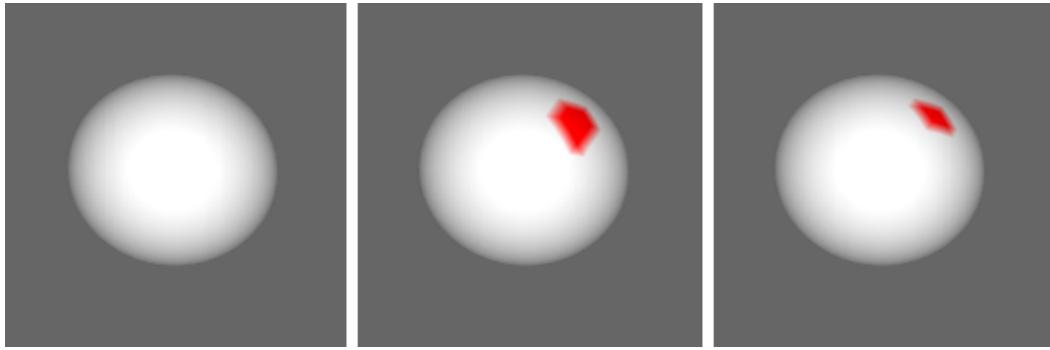


Figure 4.4: A basic vertex selection process

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 known at startup of the application, hence an ordinary array was not a suitable datatype here.

After the setup process of the application, verticesList will essentially hold 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 will describe the steps I took to ensure that the selection application meets all its requirements and ensures its key features described above are implemented correctly.

5 Results and discussion

5.1 User study data

5.2 Interpretation

6 Conclusion and future work

6.1 Insights and conclusions

6.2 Future work

List of Figures

2.1	A model and its computes mesh saliency map. Published by Lee <i>et al.</i> [LVJ05]. Bright colors (yellow and red) indicate high saliency values for their respective vertices, dark colors (shades of blue) indicate low values.	4
3.1	Notation in .OBJ format	10
3.2	An octree structure indexing four vertices. Left: 3D view, right: tree structure; 2017, https://developer.apple.com/documentation/	11
3.3	The input device used at the V2C	12
3.4	The trackable, stereoscopic glasses used at the V2C	14
3.5	The 3D objects used for the user study	15
3.6	A multi-scale mesh saliency map for an object including color scale. Published by Nouri <i>et al.</i> [NCL15]	18
3.7	Graphic representation of the weighting function in relation to the <i>unweighted</i> difference values (straight, black line)	19
4.1	Parent node (checkered) highlighted child nodes. Left: 000, right: 101 . . .	29
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	31
4.3	Determining vertices to check and final set of selected vertices	32
4.4	A basic vertex selection process	37

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.

Bibliography

- [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] gkocetree - Apple Developer Documentation weblink. <https://developer.apple.com/documentation/gameplaykit/gkocetree>. 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 arbeit-skultur 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.

Bibliography

- [set] set - cplusplus.com reference weblink. <http://wwwcplusplus.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 Kirсанов, 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://wwwcplusplus.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.

Bibliography

- [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.