INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Difference Analysis between Computer Generated and in VR User Selected Mesh Saliency Maps

Richard Metzler

Draft vom July 11, 2017

INSTITUT FÜR INFORMATIK
DER LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Master's Thesis

Difference Analysis between Computer Generated and in VR User Selected Mesh Saliency Maps

Richard Metzler

Aufgabensteller:	Prof. Dr. Dieter Kranzlmüller
Betreuer:	Markus Wiedemann
Abgabetermin:	7. Juli 2077

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig 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

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 seperate 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 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 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 commonly used term to describe regional perceived importance of digital representations of real-world (or synthetic) objects as well as 2D images, in the context of human perception 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. Both of these sets of scientific works will provide a solid knowledge of terms and methods commonly used in this field and describe the current state of the art.

2.1 Mesh saliency and human perception

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

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

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

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

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

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

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

The concept of *mesh saliency* has since been refined, augmented and adapted to serve as a basis for a multitude of specific use-cases and applications. When processing single vertex saliency, Wu *et al.* [WSZL13] took into consideration not only the curvature of the region surrounding the vertex, but also the global context of it. In other words, for each vertex to be attributed a value describing its saliency, its *global rarity*, derived from comparing its features to those of every other vertex of the object, is computed. They performed a user study in which they had participants choose one out of two saliency maps for a set of objects, presented in a random order. One map was generated using their approach, the other one with the model presented in Lee *et al.* Participants were asked to pick the one that was a closer representation of what they would have considered interesting regions and features. Since their method got picked in almost 58 per cent of cases, while the results produced by the model presented in Lee *et al.* were favored in about 42 per cent, this can be considered a true improvement of the way *mesh saliency* can be computed.

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

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

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

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

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

2.2 Human attention in Virtual Reality

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

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

With the ambition to develop a predictive model for the positive outcome using a VR setup can have compared to its expenses, Pausch *et al.* [PPW97] found that, while not being being able to help users perform search tasks in virtual space faster than with a desktop setup, users with a head-mounted display were able to complete the tasks with more certainty. They spent significantly less time re-examining areas, which they commonly did with the desktop setup - up to 41% more time. The task users were given in this work was to find a specific letter hidden on the walls of a virtual room which were textured with evenly spread sets of letters, or confidently declare that the letter wasn't present. The target letter was not actually present in the room in 50% of the tests. Based on the observation that VR users in this study barely spent any time rescanning parts of the virtual room, the authors assumed that a VR setup can have a greatly beneficial impact on systematic search tasks. They based this on the fact that spatial understanding and navigation skills are naturally very well developed parts of human cognition and proposed that the immersive experience did such a sufficiently well job at mimicking a real life environment that these skills could be used to a greater extent than in a desktop setup. Again, this is further reason to be optimistic about finding interesting patterns in what users in a VR setup find to be highly significant regions of 3D objects.

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

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

3 Concept

QuadTree, Userstudy

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 summery of relevant C++ classes the description will be further segmented subdivided according to its abstract, key requirements. The goal of this chapter is to describe how the application, especially the Octree[Oct] was designed and implemented. Accordingly, key lines of source code as well als 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 grahpics, 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 pc, 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 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 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 memopry of the video card, ensuring extremely fast access times.

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 and 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 ASSIMPs 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 applicatoin. For a more detailed description of the most crucial functions as well as a complete UML diagram representation of the application, 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 proccessing unit and drawing are implemented here. Some of the applications most crucial functionalities such as adding to and removing vertices from the global selection of vertics 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. It has been one of the most labour-intensive parts of the application since formal guides to implementing it, independent of coding language, are next to non-existent 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[l*3, l*3+1, l*3+2]	X min	X max	Y min	Y max	Z min	Z max
000	p.X min	p.X mean	p.Y mean	p.Y max	p.Z mean	p.Z max
001	p.X mean	p. X max	p.Y mean	p.Y max	p.Z mean	p.Z max
010	p.X min	p.X mean	p.Y min	p.Y mean	p.Z mean	p.Z max
011	p.X mean	p.X max	p.Y min	p.Y mean	p.Z mean	p.Z max
100	p.X min	p.X mean	p.Y mean	p.Y max	p.Z min	p.Z mean
101	p.X mean	p.X max	p.Y mean	p.Y max	p.Z min	p.Z mean
110	p.X min	p.X mean	p.Y min	p.Y mean	p.Z min	p.Z mean
111	p.X mean	p.X max	p.Y min	p.Y mean	p.Z min	p.Z mean

Table 4.1: child node bounding values

4.3.1 Spatial Indexing via Octree

As mentioned above, the `ocTree` class handles spatial indexing and, therefor, provides quick access to every vertex of an imported 3D object via a set of integer-like (`size_t`) indices. The general approach to this implementation of the concept of ocTrees was desigend 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 save 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, the 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 accessig 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 any time. At any level l, those values of *Id* can be found at positions *l*3*, *l*3+1* and *l*3+2* within it. If *l*3+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 stands refers to the parent node, $ID[n]$ is the nth element of identifer *ID*. 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 determing the maximum amount of vertices that one leaf node can store and 3. an integer determing the maximum split depth, in other words the maximum depth of the tree. As an optional fourth parameter, a boolean flag can be passed as well. Its default value is set to be `false`, if it is set to `true`, additional information regarding the recursive construction of the tree, including identifiers, level, dimensions and number of vertices held by each subtree, 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:

```
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 determing 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 wit 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 determing whether, during the recursive building proccess of the subTree, iformation 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  int identifierSize = m_parentIdentifier.size();
2  int levelOffset = m_level*3-3;
3  std::vector<bool> id(identifierSize);
4
5  for (int i = 0; i < levelOffset; i++) {
6      id[i] = parId[i];
7  }
8
9  id[levelOffset] = splitDirections[0];
10 id[levelOffset+1] = splitDirections[1];
11 id[levelOffset+2] = splitDirections[2];
12
13 for (int j = levelOffset+3; j < identifierSize; j++) {
14     id[j] = false;
15 }
16 m_identifier = id;
```

After setting up the essential variables, lines 5 - 7 copy the parts of the parents 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 9 - 10 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 13 - 15 take care of assigning `false` as placeholders to every position that is not relevant due to the node's level. In line 16, 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 (expecially the ones that are located on boundaries of subtrees) to exactly one subtree - for the root node as well as all subtree nodes.

4.3.1.4 setDimensions()

This simple private function takes care of setting up correct minimum, mean and maximum values in x, y and z dimension for a newly created subtree node. The following parameters are required. 1. an array of nine `float` values containing the parent nodes' bounding and mean values and 2. a reference to an `std::vector` of three boolean values containing the *split directions*. Based on the *split directions* given via the second parameter, setting up the bounding values for the new subtree is a matter of assigning the correct value of the first parameter. Said second parameter - a vector of `float` values - 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.minX	p.meanX	p.meanY	p.maxY	p.meanZ	p.Z max
N_5	p.meanX	p.maxX	p.meanY	p.maxY	p.minZ	p.meanZ

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

```
if (splitDirections.at(0) == false) {
    m_minZ = parentDimensions[8]; // minZ = p.meanZ
    m_maxZ = parentDimensions[7]; // maxZ = p.maxZ
} else {
    m_minZ = parentDimensions[6]; // minZ = p.minZ
    m_maxZ = parentDimensions[8]; // maxZ = p.meanZ
}

if (splitDirections.at(1) == false) {
    m_minY = parentDimensions[5]; // minY = p.meanY
    m_maxY = parentDimensions[4]; // maxY = p.maxY
} else {
    m_minY = parentDimensions[3]; // minY = p.minY
    m_maxY = parentDimensions[5]; // maxY = p.meanY
}

if (splitDirections.at(2) == false) {
    m_minX = parentDimensions[0]; // minX = p.minX
    m_maxX = parentDimensions[2]; // maxX = p.meanX
} else {
    m_minX = parentDimensions[2]; // minX = p.meanX
    m_maxX = parentDimensions[1]; // maxX = p.maxX
}

m_meanZ = (m_minZ+m_maxZ)/2;
m_meanY = (m_minY+m_maxY)/2;
m_meanX = (m_minX+m_maxX)/2;
```

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.

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

Lines 1 - 4 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` …, the children nodes of this node will be able to retrieve their spatial bounding values directly from `parDimensions`.

Lines 7 and 8 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 11 - 13 and 14 - 16 show the initialisations of two new `ocTree` objects using the class' private subtree constructor. Note that the two shown calls differ only in one parameter, `split`n. This is also true for the remaining six (not shown) objects to be constructed.

Lines 19 and 20 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.

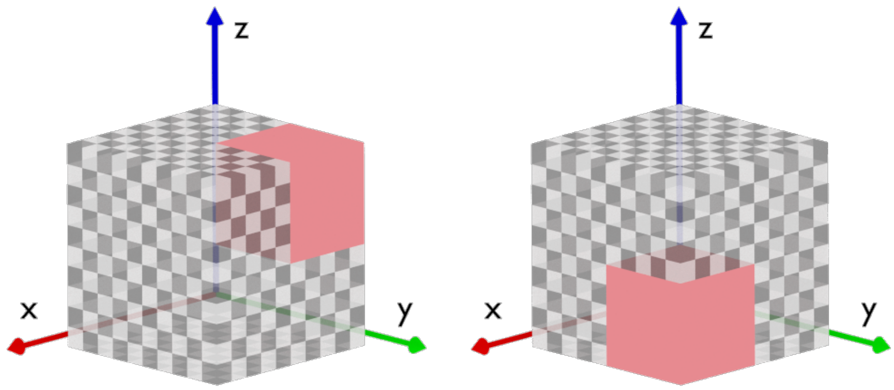

Figure 4.1: Parent node (checkered) highlighted child nodes. Left: 000, right: 101

`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 acces 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 to a leaf node 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 thorugh the entirety of the set of passed vertices. Its basic procedure is depicted in the following simplified C++ code snippet.

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

Lines 1 - 10 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 contain by their parent node. So the root node of an `ocTree` will always check every single vertex of the loadedc 3D object but the higher their level, the subtree nodes will have to check fewer and fewer vertices.

Line 12 shows the crucial check wheter the number of vertices within the bounds of the calling node exceeds the maximum allowed number of vertices per node. If this is the case and the maximum allowed level (`m_maxSplitDepth`) of subtrees is not reached yet (line 13), a call of `split()` by this node follows.

Figure 4.2 depicts a simple `ocTree` structure that could result from indexing a small set of 3D data. This particular tree has a root node at level zero, represented as a basic cube, in the upper left part of the figure. As the number of vertices within the bounds of the root exceeds the maximum numbers of vertices a node may hold in this particular tree, the root 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 created a total of 16 new child nodes, each at level 2.

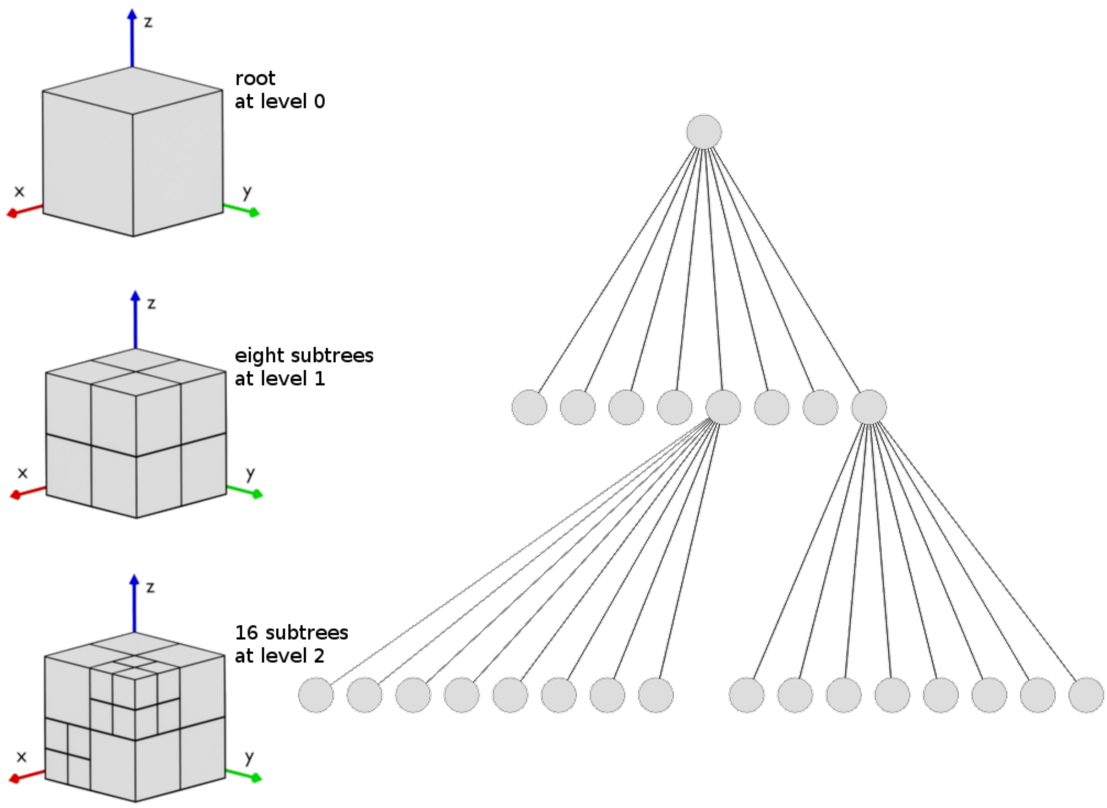

Figure 4.2: depiction of a simple `ocTree`. Left: 3D view, step-wise representation of the splitting process (downwards from the top). right: final 2D representation of the tree's structure

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()`

4.3.2.2 `removeVerticesFromSelectionByCoordinates()`

4.3.3 Tracking Selection

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 correclty.

List of Figures

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.

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

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

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

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

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

[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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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