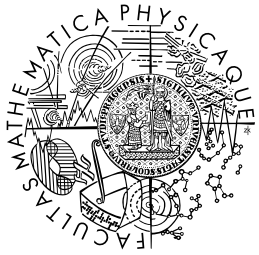


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Jan Horešovský

Visualization of the difference between two triangle meshes

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: RNDr. Josef Pelikán
Study programme: Computer Science
Study branch: Programming and Software Systems

Prague 2018

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank Ida, Lenka and Petr for their support, my supervisor RNDr. Josef Pelikán for his base code, his feedback and ideas and the Laboratory of 3D Imaging and Analytical Methods of the Faculty of Science at Charles University for data. Special thanks belong to Doc. RNDr. Jana Velemínská, Ph.D. for distributing the user study and to all volunteers who have participated in it.

Title: Visualization of the difference between two triangle meshes

Author: Jan Horešovský

Department: Department of Software and Computer Science Education

Supervisor: RNDr. Josef Pelikán, Department of Software and Computer Science Education

Abstract: Visualization of the difference between two triangle meshes is useful in geometric morphometrics where the shapes of biological objects such as bones, facial symmetries and others are studied. Existing visualizations are mostly done by encoding various difference metrics into vertex color. However, this one-dimensional information is not enough to display multiple metrics at the same time. To overcome this limitation, we implemented an algorithm which employs the techniques of vector field visualization and uses clustered 3D arrows to encode the metrics. Focusing on visual appearance, we applied it in several types of visualizations in an experimental application called MeshDiff. We also conducted a user study of both existing and new visualizations to compare their performance in various use cases and investigate the possibilities for future improvement.

Keywords: visualization, triangle mesh, difference, clustering, vector field

Contents

Introduction

Visualization of the difference between two triangle meshes[1]is often used in geometric morphometrics[2]where the shapes of biological objects such as bones, facial symmetries and others are studied. In order to compare these objects, various difference metrics have been devised such as the distance between two overlaid meshes or the difference in curvature of corresponding parts of their surface. For demonstration and publication purposes it is extremely useful to visualize these difference metrics in such a way that they are easily understood. Presenting the differences on either raw triangle meshes or raw metric values is highly unsuggestive due to the large amount of data[3].

We will first examine how mesh difference visualizations are handled in publicly available software, we will mention their disadvantages and then we will state how we aim to improve them.

Existing Mesh Difference Visualizations

Our main reference program will be Morphome3cs[4], a piece of software developed by researchers at Charles University which allows users to process and analyze 3D data, mostly of anthropological origin.

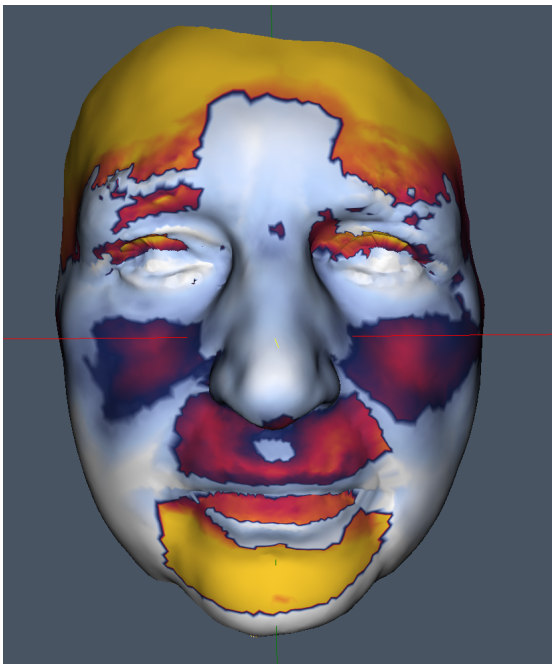


Figure 1: Morphome3cs - Corresponding vertex distance visualization

[1]A triangle mesh is a collection of vertices, edges and triangles which is used for representing 3D objects in computer graphics.

[2]The word "morphometrics" is of Greek origin and translates to "the quantitative analysis of form".

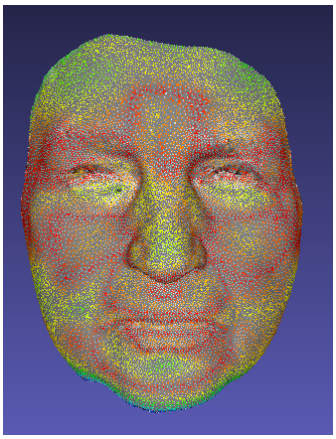
[3]Triangle meshes used in this thesis have over 15,000 vertices and it is not uncommon for triangle meshes of everyday objects to have over 100,000 vertices.

[4]CGG MFF UK [2015]

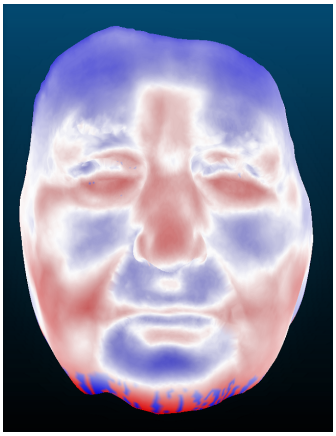
Morphome3cs is able to generate a homologous[5]pair of triangle meshes from two arbitrary triangle meshes and use it to compute and visualize the difference between them. Currently, Morphome3cs is able to produce color-based visualizations of multiple difference metrics. These metrics are:

- Corresponding vertex distance (Fig. 1)
- Corresponding vertex distance projected into surface normal
- Angle between corresponding surface normals
- FESA[6]
- Curvature difference

The disadvantage of these color-based visualizations is that they fail to capture multidimensional information. For example, when using corresponding vertex distance as a metric, it is impossible to encode both its magnitude and its direction into color at the same time while maintaining visual clarity.



(a) MeshLab - Hausdorff Distance visualization



(b) CloudCompare - Vertex distance visualization

Figure 2: Visualizations in MeshLab and CloudCompare

Other approaches can be found for example in MeshLab [7] (Fig. 2a) or CloudCompare[8] (Fig. 2b). In MeshLab, the difference between two arbitrary triangle meshes can be visualized directly without the homologization step using the Hausdorff Distance[9]encoded into color. CloudCompare is also able to visualize the difference between two arbitrary triangle meshes (or even a triangle mesh and a point cloud or two point clouds) without homologization. It does it by finding

[5]Two triangle meshes are homologous if they have the same number of vertices, edges and triangles and there is a one-to-one mapping between them. Vertices are numbered and a vertex $v_i \in Mesh_1$ corresponds to a vertex $w_i \in Mesh_2$. Similar rules apply to edges and triangles.

[6]Finite Element Surface Analysis - captures the difference between corresponding triangle areas

[7]Cignoni et al. [2008]

[8]CloudCompare [2018]

the nearest triangle to a given vertex, computing the distance between them and encoding this metric into color [CloudCompare, 2015].

Another drawback of the presented visualizations is that they only capture local differences. This makes answering questions such as "Which face has a larger nose?" more difficult because the observer has to synthesize various local pieces of information manually. Such an approach is then time-consuming and prone to inaccuracies.

Our Goal

In order to overcome the limitations of the above color-based visualizations, this thesis is looking to create an arrow-based visualization technique combined with clustering which will be able to display multidimensional information and also group similar information together automatically. As a proof of concept, we will apply it in various visualizations using the corresponding vertex distance metric because this metric suffers from information loss when visualized using colors only. Other metrics which can be represented by arrows can be easily added. We will follow the approach applied in Morphome3cs, therefore our input will be two homologous triangle meshes. We will focus on the visual appearance of the devised visualizations and their implementation in an experimental application called MeshDiff. Lastly, a user study will be conducted to assess the quality of the new as well as the existing visualizations in various use cases. This study will also serve as a basis for further development and the potential incorporation of the visualizations into Morphome3cs.

Thesis Structure

Chapter 1 of this thesis is concerned with the description of the proposed visualizations and the main ideas behind them. Chapter 2 delves into the implementation details of the visualizations in MeshDiff. Chapter 3 describes the user study and chapter 4 presents its results and suggests possible future improvements. MeshDiff parameter description and user documentation can be found in the attachments.

[9]One-sided Hausdorff Distance between two point sets S_1, S_2 is defined as $E(S_1, S_2) = \max_{p \in S_1}(\min_{p' \in S_2} d(p, p'))$ where d is the Euclidean distance between two points. Details of the computation used in MeshLab can be found in Cignoni et al. [1998].

1. Problem Analysis & Solution

In this chapter, we will explain the difference visualization problem more thoroughly and we will outline the process of designing our new visualizations. We will conclude the chapter by presenting the resulting visualizations and some of their properties.

1.1 Problem Input

We will begin by illustrating the input to our visualization problem.

Our input are two homologous triangle meshes which are to be compared. In order to clearly distinguish between these two meshes, we will call one of them the *rendered mesh* and the other one the *reference mesh*. The mesh which carries the visualization is the *rendered mesh*. For example, Fig. 1 shows a *rendered mesh*. The visualization demonstrates how different this *rendered mesh* is from its *reference mesh*.

For the purposes of our arrow-based visualization technique, it is appropriate to use difference metrics which can be represented by 3D vectors because these can be directly rendered as arrows. We will restrict ourselves to corresponding vertex distance because this metric, used in Morphome3cs, suffers from information loss when visualized using colors only. In addition to that, we will also include corresponding vertex distance projected into surface normal despite the fact that this metric has only one dimension. We will use it to compare arrow-based and color-based visualizations. Other metrics can be easily added if needed.

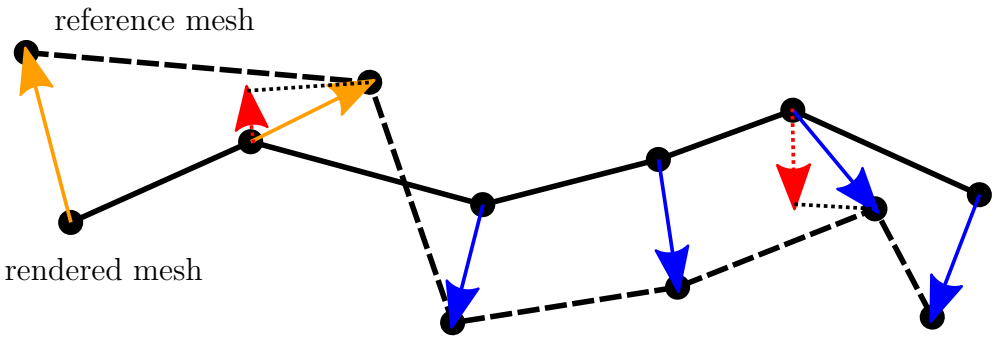


Figure 1.1: A simplified 2D schema of difference metric vectors. Yellow and blue arrows represent corresponding vertex distance, whereas red arrows show how the distance is projected into surface normals. Yellow arrows point *outwards* and blue arrows point *inwards*. Notice that all arrows are placed in the *rendered mesh* and point towards the *reference mesh*

After a difference metric is computed, it is placed into the corresponding ver-

tices of the *rendered mesh* in the form of 3D vectors. For a better understanding, here is the meaning of such a representation in case of both chosen metrics:

- For corresponding vertex distance, a vector is placed in the vertex of the *rendered mesh* and if both meshes were overlaid, its apex would lie in the corresponding vertex of the *reference mesh*. This is therefore the most straightforward difference metric and it carries information in multiple dimensions, namely tangential difference and normal difference.
- For corresponding vertex distance projected into surface normal, the situation is largely similar, only the metric has lost the dimension of tangential difference and is only represented by a single number. When a unit surface normal is multiplied by this number, it yields our vector placed in a vertex of the *rendered mesh*.

We will split the vectors into two groups. Vectors which have an acute angle with the surface normal will be said to point *outwards* and all other vectors will be said to point *inwards* (see Fig. 1.1). This differentiation will become more meaningful when the underlying meshes are for example human faces or other enclosed objects which are very common in geometric morphometrics.

To summarize, we are now working with a *rendered mesh* containing vectors in all its vertices and we are looking for a way to visualize these vectors using an arrow-based visualization. See Fig. 1.1 for a schema of this situation.

1.2 Seeing Mesh Difference as a Vector Field

If all vectors were drawn directly onto the *rendered mesh* as arrows, the visualization would become too cluttered (see Fig. 1.2). We also need a way to group similar vectors together to provide a more global difference visualization which existing color-based visualizations described in the introduction are incapable of providing.

One way to solve this problem is to devise a simple abstraction and use tools available for this abstraction.

Vectors saved in the vertices of a triangle mesh form a discrete bounded vector field. Visualization of discrete vector fields is a very active research area with applications in engineering, molecular modelling and computational fluid dynamics. There exist many scientific papers studying this topic, such as Telea and van Wijk [1999], Garcke et al. [2000], Du and Wang [2004] or Peng et al. [2012].

Drawing inspiration from these papers, we will employ a clustering technique to reduce the number of vectors and subsequently visualize this reduced information.

To summarize, we are now visualizing a discrete bounded vector field.

1.3 Vector Field Clustering

Clustering in general is a very subjective task (see Fig. 1.3) and there are numerous approaches to clustering available. For this reason we present an overview of clustering techniques used in vector field visualization to gain a better understanding of which technique might best suit our purposes and why.

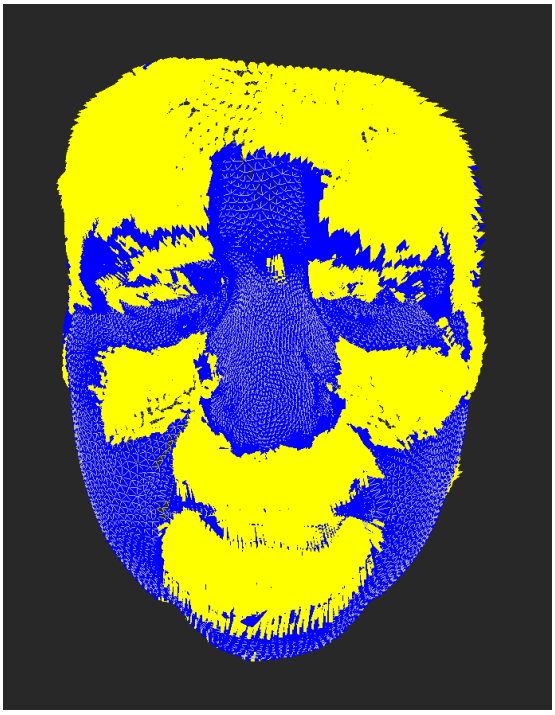


Figure 1.2: MeshDiff - Unclustered arrows

1.3.1 Overview of Vector Field Clustering Methods

Telea and van Wijk [1999] use hierarchical clustering[10]where neighboring clusters with the lowest clustering error are merged first. Each cluster has a representative vector and during the merge, the weighted average of the two vectors is computed and assigned to the newly formed cluster. In order to compute the clustering error, the paper introduced elliptic iso-error contours[11]. This clustering method is primarily aimed at 2D rectilinear vector fields but can be also used in 3D when the error function is modified appropriately. It is possible to use up to seven parameters to configure the clustering.

Garcke et al. [2000] use a continuous clustering method[12]based on the physical model of Cahn and Hilliard [1958] which is used to describe phase separation and coarsening in binary alloys. This model is applied to vector field data which results in a diffusion problem rather than a splitting and merging problem. Their algorithm also assumes an either 2D or 3D rectilinear grid.

[10]Hierarchical clustering, also called bottom-up clustering, starts with each data point representing an elementary cluster. In each step of the algorithm, two clustering candidates are found from the available clusters according to certain criteria and subsequently merged. Hierarchical clustering creates a binary tree, also called a dendrogram, where each node represents a cluster and has two children from which the cluster was created. Arbitrary number of clusters covering the whole data set can then be obtained by taking roots of disjoint subtrees which, when combined, contain all the leaves of the dendrogram. See Fig. 1.4.

[11]Clustering error in Telea and van Wijk [1999] is computed by measuring the "distance" between the representative vectors v, w of the two clusters. The elliptic iso-error contour is an ellipse which has a center on the line determined by v and intersects the apex of w. All vectors w' which share the same ellipse have the same "distance" from v. The shape of the ellipse and the location of its center can be controlled by parameters. See Fig. 1.4.

[12]In continuous clustering methods, there is no notion of merging or splitting in each step of the algorithm. Instead, "a continuous scale of successively coarser cluster sets is created."

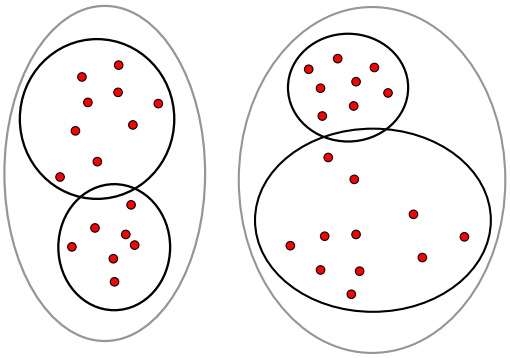


Figure 1.3: The subjectivity of clustering. How many clusters are there in the image? Two? Four? Five?

Du and Wang [2004] use iterative clustering[13]where Voronoi regions[14]are created around the initial cluster centers and a distance function is applied to each of them. The set of clusters which has the lowest value of the distance function is selected as the final cluster set. This method works with 2D and 3D rectilinear vector fields and can be influenced by two parameters and by the distribution of the initial clusters.

Peng et al. [2012] use hierarchical clustering similar to Telea and van Wijk [1999] with the difference that the clustering is computed on a GPU by encoding a given static view of the vector field into a rasterized image. The computation is then done for this specific image. In order to obtain the clustering error of clusters C_1, C_2, a very simple formula is used:

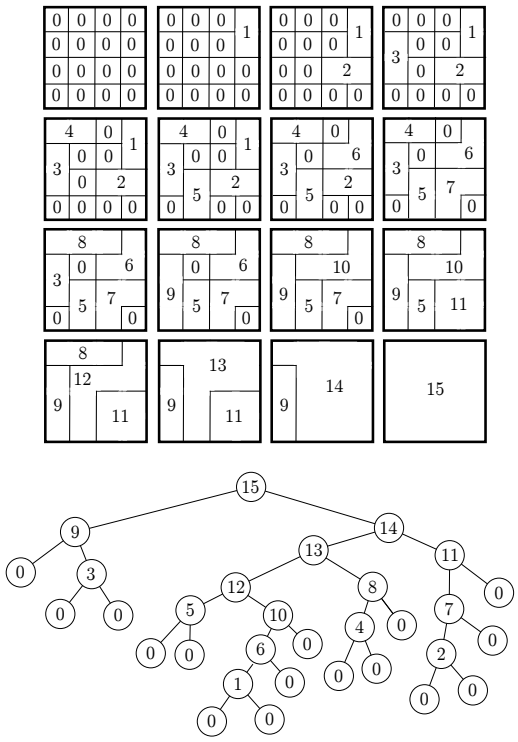
$$\boldsymbol{e}(C_1, C_2) = k_d \cdot \frac{d_{C_1C_2}}{d_{max}} + k_v \cdot \frac{v_{C_1C_2}}{v_{max}} + k_\alpha \cdot \frac{\alpha_{C_1C_2}}{\alpha_{max}} + k_m \cdot \frac{m_{C_1C_2}}{m_{max}} \tag{1.1}$$

where $k_d + k_v + k_\alpha + k_m = 1$ are weight coefficients. The other components are the following:

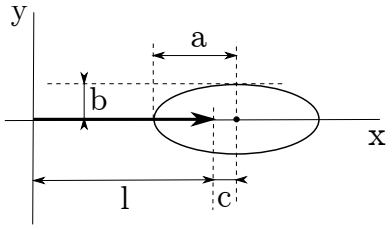
- $d_{C_1C_2}$ is the Euclidean distance between the positions of the representative vectors of the clusters. The maximum distance d_{max} is the length of a diagonal of the geometry bounding box.
- $v_{C_1C_2}$ is the difference between the lengths of the representative vectors and v_{max} is the largest length in the whole data set.
- $\alpha_{C_1C_2}$ is angle between the representative vectors. The maximum angle is $\alpha_{max} = 180°$.

[13]Iterative clustering algorithms start by assigning the data points to a given number of clusters in a random way or by using a clustering approximation. In each step, the clustering error of all clusters is computed and data points are reassigned in a way which decreases the overall clustering error.

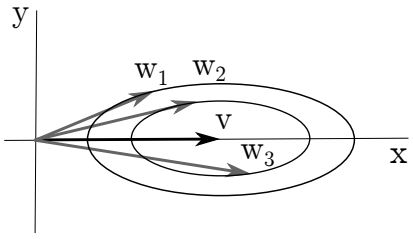
[14]Voronoi diagram is a partitioning of a plane into regions. The partitioning is done by selecting some initial points as seeds. Then a region is formed around each seed in such a way that a point p of the plane belongs to the region of seed s if and only if $d(p, s') \geq d(p, s)\ \forall s'$ seeds such that $s' \neq s$.



(a) Various clusterings of a data set and the associated dendrogram



(b) The ellipse and its parameters.



(c) The iso-error contours of multiple vectors

Figure 1.4: The illustration of clustering in Telea and van Wijk [1999]

- $m_{C_1C_2}$ is the sum of the mesh resolutions of the two clusters. m_{max} is the largest value of m in the whole data set.

The mesh resolution component differentiates this approach from all the others because it represents an approximation of the density of the underlying mesh in a given local area[15]. Including it in the error formula assigns higher error to dense clusters which results in a larger amount of clusters (higher precision) in dense areas of the mesh and a smaller amount of clusters (lower precision) in sparse areas of the mesh. To compute it, we use an approximation from Peng et al. [2012] where for each vertex v, mesh resolution $m_v = \frac{1}{\overline{e}_{avg}}$ where $\overline{e}_{avg}$ is the mean length of all edges from the neighborhood of v depicted in 1.5.

This method is therefore aimed at non-rectilinear 3D vector fields. It has five parameters for user configuration (four weights and the number of clusters to be retrieved from the dendrogram).

1.3.2 Selection Criteria for Our Clustering Method

For our visualization purposes, we will utilize fragments of the presented clustering methods and introduce certain modifications to accommodate our needs. Here are the criteria for choosing our clustering method:

[15]We use the term *mesh resolution* in this thesis to be aligned with the terminology in Peng et al. [2012].

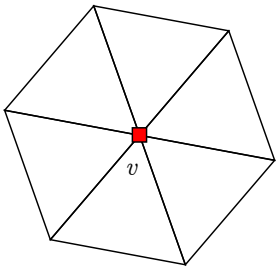


Figure 1.5: The neighborhood of vertex v used for computing mesh resolution

- Suitability for the specifics of our vector field
- Simplicity
- Ease of user configuration

Methods which are suitable for our specific vector field (i.e. a triangle mesh with a vector in each vertex) will help us tailor the clustering process and achieve better results.

Simple methods will allow us to quickly obtain a baseline which will help us decide which modifications will improve the algorithm in our conditions.

Lastly, the ease of user configuration will make our algorithm user-friendly and therefore it will increase the adoption rate.

1.3.3 Our Clustering Method

We will now describe our clustering method.

Clustering Algorithm

The preference of hierarchical clustering over other mentioned clustering methods is justified by its simplicity and also the fact that a dendrogram is created during this clustering. This will allow us to quickly react to user requests for various cluster counts with given clustering parameters. Hierarchical clustering will thus make our approach user-friendly. We will, however, introduce one optional condition as to which clusters can be merged together. Our clustering candidates will not only have to be neighbors (as in Telea and van Wijk [1999]) but their representative vectors will both have to point either *inwards* or *outwards* because this distinction may be of importance to the user. On the other hand, we leave this condition as optional because the resulting dendrogram is not a tree but a forest instead. This may result in undesirable artifacts in the visualization because for certain cluster counts, the clustering will not cover the whole data set. See Fig. 1.6 for an illustration. Clustering resulting in a forest dendrogram will be called *signed clustering* and clustering resulting a tree dendrogram will be called *simple clustering*.

We are not going to use the GPU-based approach to clustering computation from Peng et al. [2012] because it does not allow the user to view the final visualization from varying angles in real time. In order to support interactive

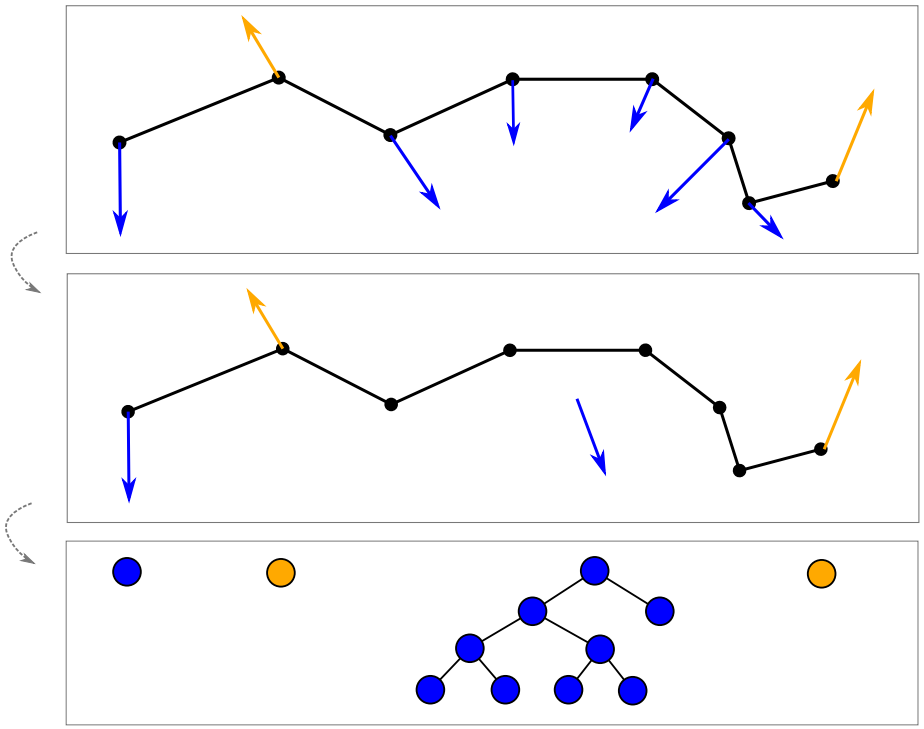


Figure 1.6: An illustration of *signed clustering* and the resulting dendrogram. In this case, it is impossible to cover the whole dataset with less than four clusters

visualization viewing, we will store our vector field in memory and compute the clustering on the whole field in one process instead (as in Telea and van Wijk [1999]).

Clustering Error Function

We will use the error function presented in Peng et al. [2012] (see equation 1.1) because it assumes vector fields on other than rectilinear grids. Its mesh resolution component makes it suitable to use in the clustering of vector fields on triangle meshes. The function is also simpler and more scalable than the elliptic iso-error contours presented in Telea and van Wijk [1999] which lead to cube root equations in 3D.

Cluster Merging

Besides clustering error computation, merging is the second crucial part of a hierarchical clustering algorithm. In the clustering of vector fields, a representative vector is usually assigned to a newly formed cluster. In our case, this shall be a weighted average of the representative vectors of the child clusters where the weight will be the geometrical area of the given cluster. Geometrical area is a better metric for cluster size than data point count because our underlying triangle mesh is expected to be of varying density.

Summary

Our clustering method will use the hierarchical algorithm and CPU-based clustering computation from Telea and van Wijk [1999] (*simple clustering*) with the added optional condition of merging only clusters whose representative vectors both point in the same direction - either *inwards* or *outwards* (*signed clustering*). We will use the error function from Peng et al. [2012] (see equation 1.1). Merging of two clusters will be done by computing the area-weighted average of their representative vectors and assigning the resulting vector to the newly formed cluster.

After the clustering step, we are visualizing a set of clusters, each of which has a representative vector encoding a difference metric averaged across the whole cluster.

1.4 Proposed Visualizations

Here we present the descriptions of the new visualizations. One of them is arrow-based as was our goal and the rest are complementary visualizations leveraging the clustering process in order to enhance the visual appearance and clarity of existing visualization techniques.

1.4.1 Arrows

Once a clustering of a vector field is obtained, representative vectors of all clusters are visualized using 3D arrows. For this purpose we have prepared a simple 3D model of an arrow (see Fig. 1.7) which is copied into the scene at a specific position, angle and scale given by the representative vector and the cluster it belongs to.

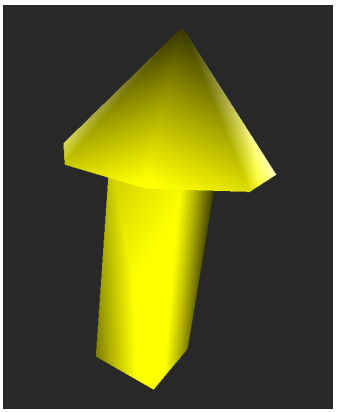


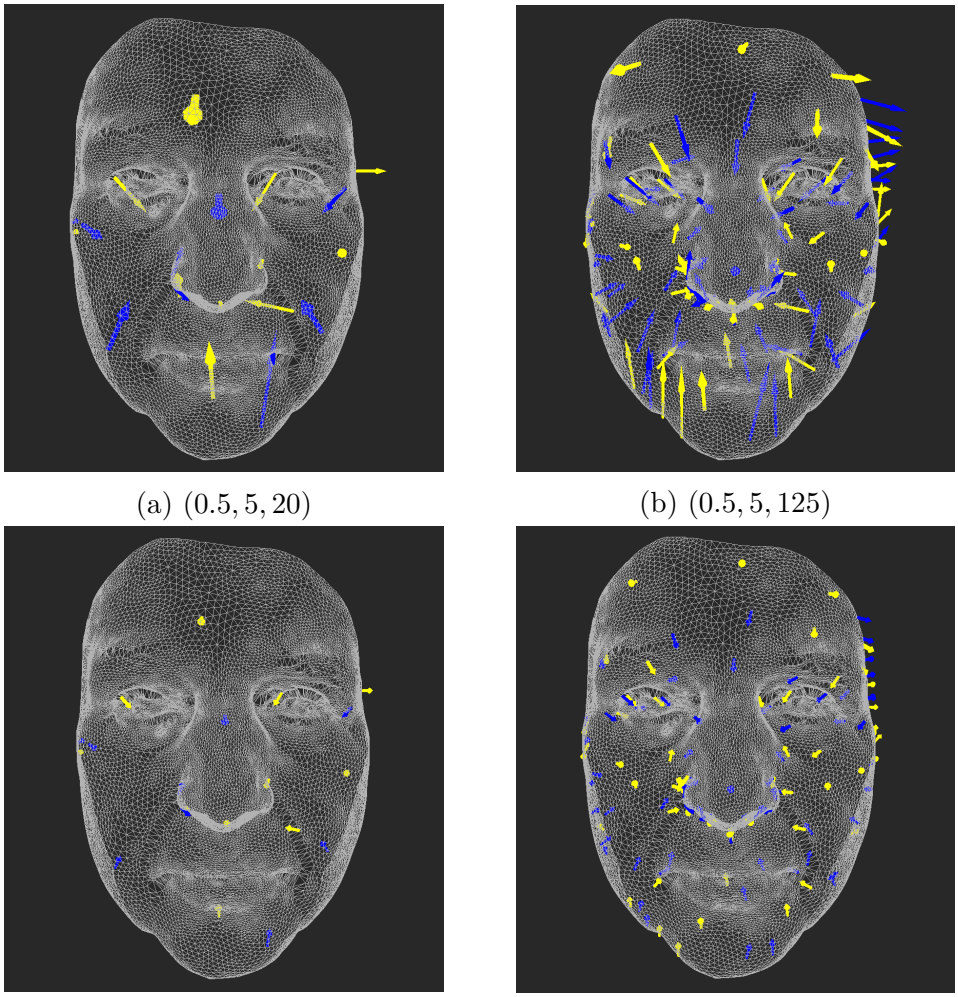
Figure 1.7: The 3D model used for arrow visualizations

The length of the representative vector influences the length of the 3D arrow. Because the values of the metric can be very close to zero and because their value range is not generally very large, we have decided to set a certain minimum scale and maximum scale, which are adjustable by the user, and map the metric values (vector lengths) to the interval between them. In general, such an approach gives a more visually pleasing and clearer results, especially when the interval is chosen to be large enough.

The scale of the 3D arrows also reflects the geometrical area of the clusters. The larger the clusters are, the thicker the arrows. Areas are again mapped to a user-defined interval for a clearer result. Large clusters are usually important because they represent a general trend in a given area and users should be able to see them more easily and also distinguish them from less significant clusters.

Lastly, and most importantly, the direction of the representative vector is directly reflected in the direction of the 3D arrow. In addition to that, arrows pointing *inwards* can have a different color than arrows pointing *outwards* to make their differentiation easier for the user.

We have therefore managed to encode three-dimensional information into our visualization. See Fig. 1.8.



(a) $(0.5, 5, 20)$ (b) $(0.5, 5, 125)$

(c) $(0.5, 1, 20)$ (d) $(0.5, 1, 125)$

Figure 1.8: MeshDiff - Arrow visualizations with various parameter settings[16]

Expected Performance

Arrow visualizations combined with clustering are expected to perform well in highlighting the general trends in large parts of the mesh. They are also expected

[16]Parameter format in captions: `([minimum height & width scale],[maximum height & width scale],[cluster count])`

to perform considerably better than color-based visualizations when asking about the direction of the difference, which is particularly important in cases where the difference has a very small angle with the surface of the mesh.

1.4.2 Cluster Color

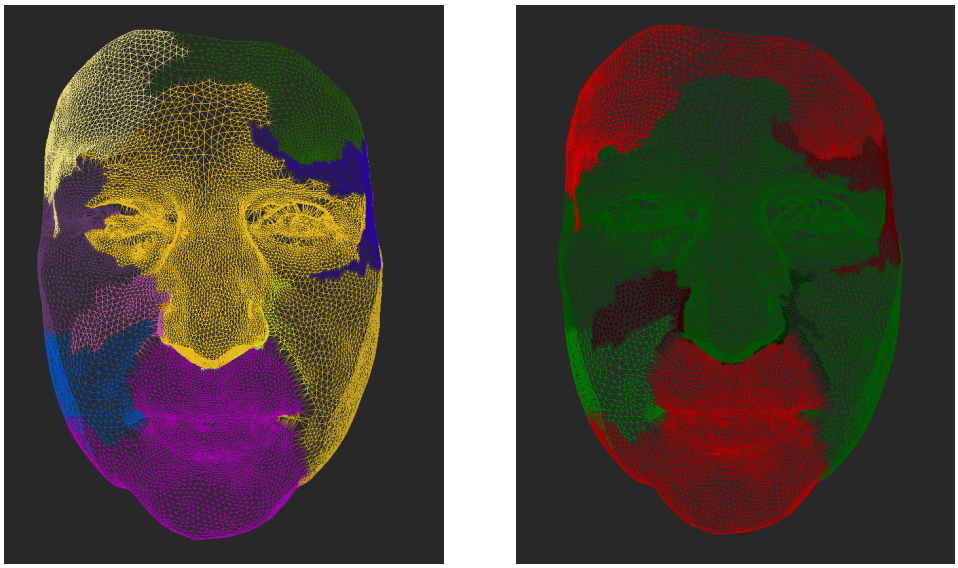
Next, we introduce two color visualizations of clusters: random and metric-based.

Both of these visualizations need to be aware of which mesh vertices belong to a given cluster. Then either a random color is assigned to all vertices in a given cluster (see Fig. 1.9a) or a color based on the length of the representative vector of the cluster is assigned (see Fig 1.9b). The former case is basically the original color visualization of a metric, only applied to clusters.

We propose two modes of assigning the color to the vertices based on metrics, the relative mode and the absolute mode. In both modes, a distinct user-defined color hue is assigned to clusters whose representative vectors point *inwards* and *outwards*. What differentiates the two modes is the color shade assigned to clusters of a specific metric value.

In the relative mode, the highest and the lowest metric value is evaluated first. Black color is then assigned to the lowest value and the brightest user-defined color is assigned to the highest value. All other values are mapped to this interval and receive a color which is a proportional mixture of the user-defined color and black.

In the absolute mode, black color is assigned to metric values of zero and the brightest color is assigned to metric values higher than or equal to a user-defined value. This allows the user to compare the absolute value of the difference metric across several visualizations.



(a) Random (b) Metric-based

Figure 1.9: MeshDiff - Cluster color visualizations

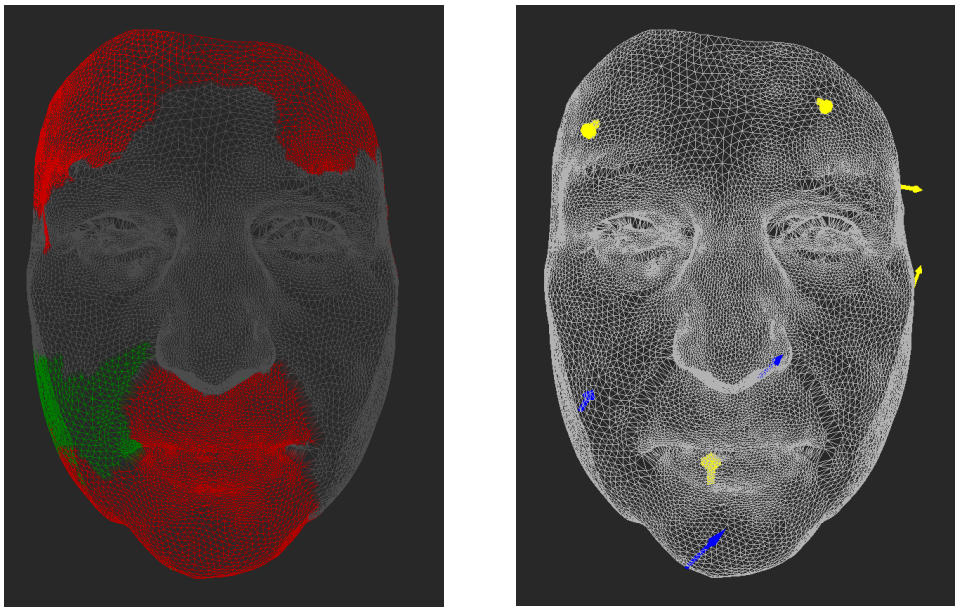
Expected Performance

Random cluster color is expected to be used for the purpose of configuring the clustering parameters as the edges of the clusters are clearly visible in this case.

Metric-based cluster color is expected to perform well in cases when we have found the most important differences using a more sophisticated visualization and want to present those using a visualization which is as clear as possible.

1.4.3 Thresholding

For all types of metric-based visualizations including the original color-based ones, we present thresholding. Thresholding excludes either clusters whose metric values are too low or clusters whose area is too small. Therefore all visualization elements related to a cluster which has not passed through the thresholding are not shown in the result. See Fig. 1.10.



(a) Vertices of excluded clusters are grayed out

(b) Arrows of excluded clusters are missing

Figure 1.10: MeshDiff - Thresholded visualizations where clusters with metric values less than 3 are excluded

Expected Performance

Thresholding is expected to enhance the effect of metric-based cluster color visualizations by performing very well when segmenting and emphasizing a previously discovered difference which is important to the user. It is also expected to help answer questions about the largest and the smallest differences between two meshes.

1.4.4 Combined Visualizations

The last visualization type presented in this thesis is the combination of color and arrow visualizations. See Fig. 1.11.

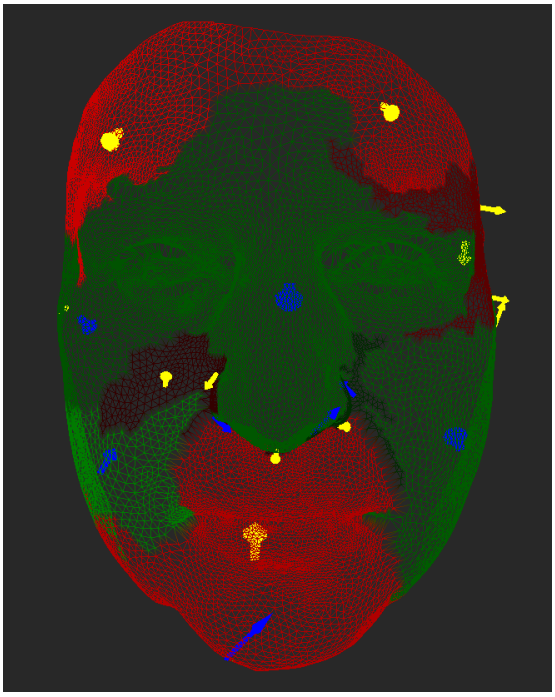


Figure 1.11: MeshDiff - Combined visualization

Expected Performance

Combined visualizations complement each other in areas when only one visualization is not sufficiently clear. Color visualization is expected to highlight the dimension of the metric we are interested in the most, for example vertex distance magnitude, while arrow visualization is expected to carry other dimensions like direction and cluster size. This method is expected to have a good balance between clarity and information richness.

1.5 The Effect of Clustering Parameters

The function used for computing the clustering error (see equation 1.1) has four parameters:

- Direction weight
- Position weight
- Magnitude weight
- Resolution weight

A specific configuration of these parameters can influence the outcome of the clustering process considerably. In general, setting one of them higher than the others results in finer clustering in that dimension and coarser clustering in the others. We will now describe the effect of all the parameters in more detail.

1.5.1 Direction Weight

High direction weight forms many small clusters in areas of high surface curvature and large clusters in flat areas. Therefore, it mostly captures the high-curvature changes of shape. Resulting clusters have uneven sizes. See Fig. 1.12a.

1.5.2 Position Weight

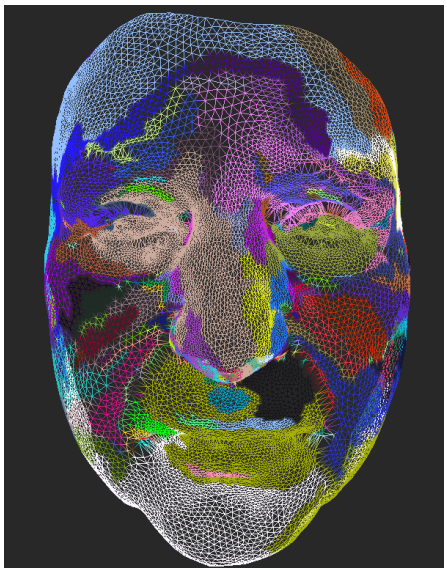
Setting the position weight higher than others results in clusters of even size where each of them represents the overall difference in a certain area regardless of the variety of directions and magnitudes in that area. See Fig. 1.12b.

1.5.3 Magnitude Weight

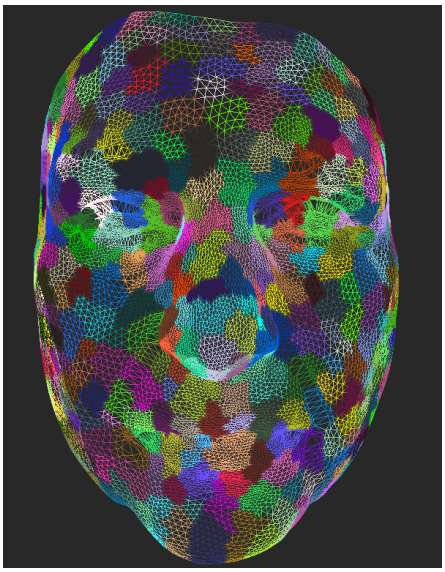
Magnitude weight can play a significant role when using the metric-based cluster color visualization because its high value will highlight iso-magnitude contours like in a geographical map. Such an approach can be useful when grouping and segmenting areas with a certain absolute value of the difference metric. See Fig. 1.12c.

1.5.4 Resolution Weight

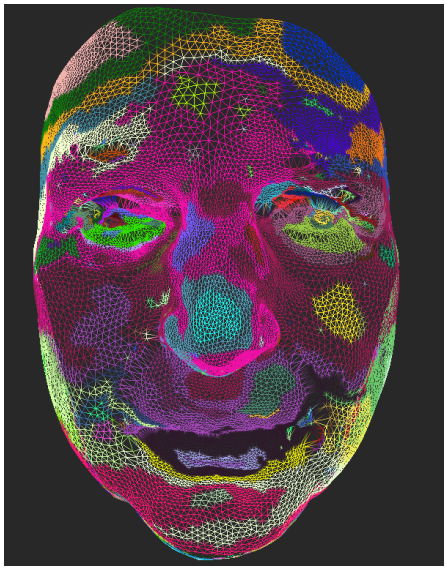
High resolution weight prefers clustering in sparse areas of the mesh and will therefore increase the precision of the visualization in very dense areas. This effect partially complements high direction weight because high-curvature areas of triangle meshes are usually more dense in order for the high curvature to be captured well in the mesh. See Fig. 1.12d.



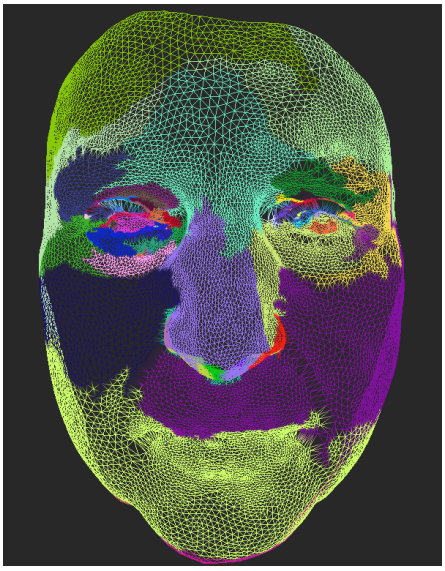
(a) High direction



(b) High position



(c) High magnitude



(d) High resolution

Figure 1.12: MeshDiff - Various clustering parameter settings

2. Implementation

In this chapter, we will delve into the implementation details of the presented visualizations applied in our experimental application called MeshDiff. We will also describe the overall architecture of MeshDiff.

2.1 Visualization Algorithm

All presented visualizations share a common workflow (algorithm) in MeshDiff. In this section, we will discuss all parts of the workflow, namely:

- Difference metric computation
- Vector clustering
- Cluster visualization

First, however, we will describe the internal representation of triangle meshes in MeshDiff and the framework that the visualization algorithm fits into.

2.1.1 Triangle Mesh Representation

We are using an implementation of triangle mesh boundary representation with a corner table which was presented in Rossignac et al. [2003]. The implementation was written by Josef Pelikán. For brevity, we will refer to this representation as a *scene*.

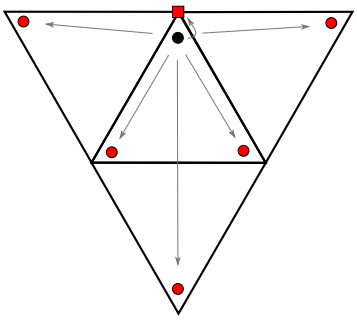


Figure 2.1: An illustration of the corner table. The black corner has direct access to all the red elements - its vertex and certain surrounding corners

In this representation, we do not primarily store the vertices of the triangle mesh but the corners of the triangles. The corner table then allows us to obtain certain surrounding corners of a given corner and associated vertices in constant time. This makes traversing an arbitrary triangle mesh very simple and fast.

We have added several methods to this implementation in order for us to be able to quickly obtain the list of neighbors of a given vertex and compute the geometrical area of clusters.

2.1.2 Algorithm Framework

The core class of MeshDiff which encapsulates the visualization algorithm is called `DiffAlgo`. `DiffAlgo` is initialized by the *rendered scene* and the *reference scene* and therefore only operates on these two specific *scenes* throughout its whole lifetime.

The main public method of `DiffAlgo` is `CreateVisualization()`. It is mainly responsible for metric computation and vector[17]clustering. All this computed data is stored inside `DiffAlgo` instances. When the clustering is ready, a visualization is created and outputted from `CreateVisualization()` using a visualizer object.

The fact that all intermediate results are stored inside a `DiffAlgo` instance means that when `CreateVisualization()` is called again, data which is difficult to compute, especially the vector clustering, can be reused if possible.

The C#-styled[18]pseudocode below describes the visualization workflow which will be further discussed in the following sections.

Algorithm 1 CreateVisualization()

```
Require: metricType, clusteringParams, visParams, visualizer, ref
    outputScene1, ref outputScene2   ▷ other variables are contained
    in the DiffAlgo object

                                          ▷ Difference metric computation
 1: if required metric values not available then
 2:     arrows = GetArrows(renderedScene, refScene, metricType);
 3:     clusteringObject = ClusteringFactory(arrows, renderedScene);
 4:     currentMetricType = metricType;
                                                    ▷ Vector clustering
 5: clusters = clusteringObject.GetClusters(clusteringParams);
                                                 ▷ Cluster visualization
 6: visualizer.BakeVis(clusters, visParams, ref outputScene1);
 7: visualizer.BakeVisInv(clusters, visParams, ref outputScene2);
    return
```

2.1.3 Difference Metric Computation

In order to compute a difference metric, this part of the workflow requires the type of the metric it should compute and also both *scenes*. The former is passed in as an argument of `CreateVisualization()` and the latter two are stored inside `DiffAlgo`.

As mentioned in section 1.1, we have included two difference metrics in the MeshDiff application, both of which can be represented by a 3D vector:

- Corresponding vertex distance
- Corresponding vertex distance projected into surface normal

[17]As mentioned in section 1.1, all our difference metrics can be represented by a vector.

[18]The `ref` keyword used with method arguments is meant to emphasize that a value is returned via these arguments.

We have created a common representation for both of these metrics called `Arrow` which encapsulates a 3D vector, its origin and other useful data and acts as input to the clustering algorithm.

Here are the most important fields of `Arrow`:

- `Origin` - the position of the vector metric, initially coinciding with a *scene* vertex (this can change during the clustering process)
- `Direction` - a 3D vector representing the metric itself
- `Orientation` - tells whether the arrow points *inwards* or *outwards*
- `VertexHandle` - if `Origin` coincides with a *scene* vertex, this is its index in the *scene* representation, otherwise it is -1

If the required metric type is equal to the current metric type stored in `DiffAlgo`, old data is reused and this step is skipped, otherwise all corresponding vertices of both *scenes* are enumerated and for each pair, the configured metric is computed on the *rendered scene* and relative to the *reference scene*. For example, in the case of corresponding vertex distance, consider a *rendered scene* S_p, a *reference scene* S_r, vertices (vectors) $\overrightarrow{v} \in S_p$, $\overrightarrow{w} \in S_r$ and a difference metric vector $\overrightarrow{m}$. Then $\overrightarrow{m_{vw}} = w - v$. Therefore, m_{vw} behaves as shown in Fig. 1.1.

Output

A list of `Arrow` instances indexed by the handles of the corresponding vertices in the *rendered scene*.

2.1.4 Vector Clustering

The clustering step requires the list of `Arrow` instances from the previous step, the clustering parameters (see attachment A.2.1) and, most importantly, the clustering type. The parameters are passed in as arguments of `CreateVisualization()`. We will now talk about the clustering type.

As mentioned in 1.3.3, our clustering has two types. For the sake of implementation consistency, we also introduce a third type, the "empty" clustering, which does not reduce the number of vectors in any way and can be used when no clustering is needed. Each clustering type has an associated class, all of which share a common interface.

Overall, there are three clustering types in MeshDiff:

- None
- *Simple*
- *Signed*

The clustering type used in `DiffAlgo` is determined by a factory method passed to its constructor. A `DiffAlgo` instance can therefore create and use the clustering object without knowing its type and at the same time it is limited to using only one clustering type. Thanks to this, computed clusterings of multiple

types can be saved simultaneously (in different `DiffAlgo` instances) and reused if needed. It is important to note that this reuse happens every time the user chooses to view a new visualization which differs from the previous one only in the number of clusters to be generated, thanks to the dendrogram (see Fig. 1.4) which contains all clusterings of 1 to n clusters where n is the number of original arrows. When a reuse is impossible, a new clustering object is created using the factory method. Another reason why this encapsulation approach was chosen is that dendrograms should be stored in the clustering objects and not in the `DiffAlgo` instances in order for the implementation details to stay hidden from `DiffAlgo`.

We will now introduce a new class called `Cluster` which `Arrow` instances are converted to in the clustering process and whose instances form nodes of the dendrogram. `Cluster` instances are able to compute the error function (see equation 1.1) given another `Cluster` instance and also to perform the merge. They contain all the information needed for the visualization to be created. Here are the most important fields of `Cluster`:

- `Neighbors` - a set of `Cluster` instances adjacent to the cluster
- `Level` - marks the step of the clustering algorithm in which this cluster was created (low number also means low clustering error in general), it is illustrated in Fig. 1.4
- `RepresentativeArrow` - an `Arrow` instance representing the metric value for this cluster
- `Size` - the geometrical area of the cluster. We have chosen this to be the sum of the areas of the underlying mesh triangles which belong to the cluster
- `LeftChild` and `RightChild` - `Cluster` instances out of which this cluster was created
- `PrimaryArrows` - a list of all the original unclustered `Arrow` instances which are tied directly to the underlying *scene* and which belong to this cluster

`Cluster` instances can be sorted and the sorting field is `Level`.

If the new `Arrow` instances are the same as the current ones `DiffAlgo` contains, the old clustering object is reused. Otherwise, the factory method is used to initialize a new clustering object with the new `Arrow` instances. After that, clustering parameters including the required cluster count are passed to the clustering object. The clustering object checks if it already has a clustering corresponding to the given parameters and if it does, it simply extracts the required number of clusters from the available dendrogram and returns them. This is how the extraction works when the dendrogram is a tree[19]:

[19]Corresponds to *simple clustering.*

Algorithm 2 Cluster Extraction from a Tree

```
Require: MaxHeap chosenClusters, requiredClusterCount

 1: chosenClusters.Clear();
 2: chosenClusters.Insert(dendrogram root);
 3: i = 0;
 4: while i < requiredClusterCount do
 5:    highestCluster = chosenClusters.ExtractMax();
 6:    chosenClusters.Insert(highestCluster.LeftChild);
 7:    chosenClusters.Insert(highestClusters.RightChild);
 8:    i++;
    return chosenClusters as list
```

Here is the extraction algorithms for forest dendrograms[20]:

Algorithm 3 Cluster Extraction from a Forest

```
Require: MaxHeap chosenClusters, requiredClusterCount

 1: chosenClusters.Clear();
 2: chosenClusters.Insert(all dendrogram roots);
 3: i = 0;
 4: while i < requiredClusterCount do
 5:    highestCluster = chosenClusters.ExtractMax();
 6:    if highestCluster.Level == 0 then
 7:       chosenClusters.Insert(highestCluster);
 8:       break;
 9:    chosenClusters.Insert(highestCluster.LeftChild);
10:    chosenClusters.Insert(highestClusters.RightChild);
11:    i++;
12: list = chosenClusters.ExtractMaxNTimes(requiredClusterCount);
    return list
```

The condition on line 6 of algorithm 3 is fulfilled for example when five clusters are requested from the forest in Fig. 1.6. In this case, `highestCluster` is a root without any children. At the same time, however, all of these roots have already been added to `chosenClusters` on line 2. We therefore insert the `highestCluster` back and break the loop because there are no more clusters to discover. We will remove the redundant clusters in the next step.

Line 12 of algorithm 3 ensures that clusters which were chosen only because they are roots (line 2) and not because their level is high enough are excluded from the selection. This is necessary to return the cluster count requested but at the same time it introduces the limitation we discussed in section 1.3.3 which is that for certain cluster counts, the returned clusters do not cover the whole *scene*.

If the desired clustering is not available (i.e. the corresponding dendrogram is not built), the clustering object performs the clustering algorithm. Here is the

[20]Corresponds to *signed clustering.*

outline of *simple clustering* (largely similar to Telea and van Wijk [1999]):

Algorithm 4 Clustering

```
Require: arrows, MinHeap clusteringCandidates                    ▷
   clusteringCandidates are ordered by clustering error

 1: for arrow_i in arrows do
 2:    initialCluster = makeCluster(arrow_i);
 3:    initialCluster.Level = 0;
 4:    initialClusters.Add(initialCluster);

 5: for cluster_i in initialClusters do
 6:    for cluster_j in cluster_i.Neighbors do
 7:       e = clusteringError(cluster_i, cluster_j);
 8:       clusteringCandidates.Insert((cluster_i, cluster_j))
 9:       mark cluster_i and cluster_j as NOT_CLUSTERED;

10: level = 0;
11: while clusteringCandidates.Count > 0 do
12:    (cluster_i, cluster_j) = clusteringCandidates.ExtractMin();
13:    if cluster_i and cluster_j are both NOT_CLUSTERED then
14:       level++;
15:       newCluster = mergeClusters(cluster_i, cluster_j);
16:       newCluster.Level = level;
17:       mark newCluster as NOT_CLUSTERED;
18:       mark cluster_i and cluster_j as CLUSTERED;
19:       for cluster_k in newCluster.Neighbors do
20:          e = clusteringError(cluster, newCluster);
21:          clusteringCandidates.Insert((cluster, newCluster));
   return newCluster as the root of the dendrogram
```

`mergeClusters()` and `clusteringError()` are implemented in accordance with section 1.3.3. We will now mention the most important implementation details of `mergeClusters()`.

First of all, when a new cluster is created, it needs to be placed in the clustering space. This is done by modifying all the neighboring clusters of its two children to be the neighbors of the new cluster instead, except for the children themselves. Also, the neighboring relation has to be made symmetrical by updating the neighbor lists of the surrounding clusters. The orientation of the new representative arrow is either inherited (*signed clustering*) or estimated based on the dominant orientation among the `PrimaryArrows` of the new cluster (*simple clustering*). New mesh resolution is obtained by averaging the resolutions of the two original clusters. Lastly, the size of the new cluster is computed directly when the cluster is sufficiently small, otherwise it is determined by the sum of the sizes of the original clusters.

Output

A list of `Cluster` instances is returned, even in the case of "empty" clustering where the only operation is the conversion of `Arrow` instances into `Cluster` instances.

2.1.5 Cluster Visualization

When the visualization itself is to be generated from the list of `Cluster` instances, visualization parameters (see attachment A.2.2), a visualizer object and two output *scenes* are required. `CreateVisualization()` receives all of these as arguments. We will now talk about visualizer objects.

In section 1.4, we have proposed several types of visualizations. Each of them has an associated visualizer object which generates it based on `Cluster` instances supplied to it. It is useful to compare these objects with clustering objects described in section 2.1.4 because both are utilized differently in `DiffAlgo`.

Clustering objects are designed to store dendrograms for potential later reuse because their computation is time-consuming and a reuse is likely. Visualizer objects, on the other hand, do not store any state or intermediate results because visualizations are fast to compute and reuses are not very helpful because outputting a visualization requires similar time as creating it. They can therefore be passed to `CreateVisualization()` from the outside and used only to directly output visualizations.

All visualizer objects have two public methods, one is intended for visualizations for the *rendered scene* and the other is intended for visualizations for the *reference scene*. The latter is obtained by inverting the former. See Fig. 1.1 for an illustration of the situation. Creating visualizations for both of the compared scenes allows for both of them to be displayed next to each other which enhances the effect of the visualizations. See Fig.2.2.

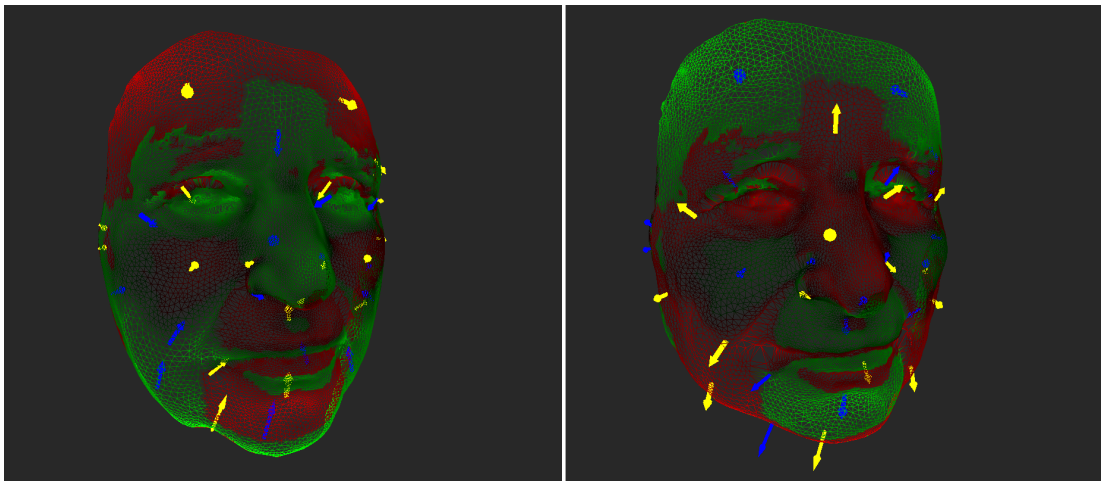
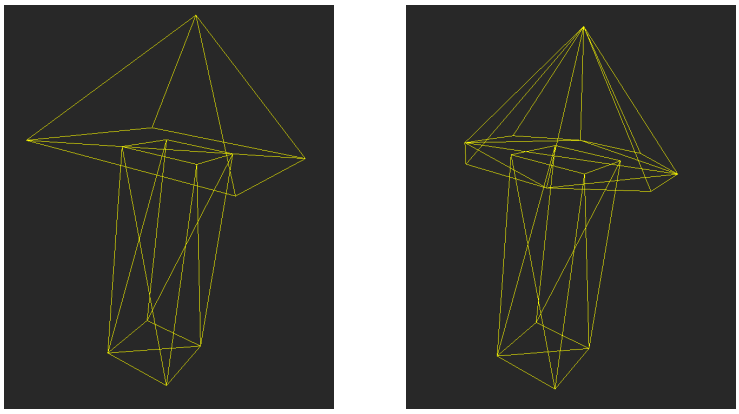


Figure 2.2: MeshDiff - Two mutually inverted visualizations displayed next to each other

We will now focus on two main types of visualizers - arrow visualizers and color visualizers - and the way they bake the visualization into the output *scenes* mentioned in algorithm 1.

Arrow Visualizers

Arrow visualizer objects only have one field which is initialized in the constructor. This field holds a basic *scene* representing a 3D arrow. The triangle mesh of this arrow was created manually in order to have the least amount of vertices and triangles possible to reduce the overall amount of data. It has 17 vertices and 16 triangles. The tip of the arrow is an eight-sided pyramid which was found to have much better appearance than the basic four-sided pyramid and this justified the slightly larger vertex count.



(a) Four-sided tip version (not used)

(b) Eight-sided tip version (used)

Figure 2.3: Wireframe models considered for arrow visualizations

The visualizer receives a list of clusters as input, loops through those which passed through thresholding and makes a scaled copy of the basic arrow in each iteration. The scale of the copy is based on the representative vector of the cluster as described in section 1.4.1 and on the visualization parameters supplied. Colors are assigned to the arrow *scenes* depending on whether the representative vector points *inwards* or *outwards* (see attachment A.2.2). All created arrow *scenes* are then copied into the output *scene* supplied.

The inverted visualization is obtained by multiplying the representative vector by -1 and assigning the arrow *scene* colors based on this version of the vector.

Color Visualizers

For each color visualization type mentioned in section 1.4 there is a separate visualizer object. These objects determine how color is assigned to vertices. In general, each visualizer loops though all `Cluster` instances supplied and for each of them it generates a color based its metric value. Clusters which passed through thresholding receive a color based on the length of the representative vector and its orientation (see attachment A.2.2). Once a color is created, it is inserted into a list at precisely those indices which belong to the `PrimaryArrows` (see section 2.1.4) of the associated cluster. It is important to respect this indexing in order to make the mapping of the colors to the *scene* vertices easier. The colors are then applied to the vertices of the output *scene* supplied[21].

The inverted visualization is also obtained by multiplying the representative

vector by -1 and generating the colors based on this version of the vector.

Output

The generated visualization is baked into the first supplied output *scene* and an inverted visualization is baked into the second supplied output *scene*.

2.2 MeshDiff Architecture

Similarly to MeshLab[22] and Morphome3cs[23], in MeshDiff, the core functionality is the ability to load and store triangle meshes and to view them interactively using the mouse cursor. Because this functionality is present in almost all programs which work with triangle meshes and at the same time it is non-trivial, we reused available code which provides it, more specifically we built MeshDiff on top of a mesh viewer application written by Josef Pelikán.

In this chapter, we will mention the platform MeshDiff is built on and intended for and we will talk about the features we added to the reused mesh viewer and the changes we made to the user interface to support of visualization rendering purposes. At the end, we will reveal how the visualization workflow is incorporated into MeshDiff.

2.2.1 Platform

The reused code by Josef Pelikán is a C# application with a WinForms user interface which uses the OpenTK library, encapsulating the OpenGL API, to render graphics. The only targeted operating system is Windows which is sufficient for our experimental purposes.

In order to clearly mark which parts of MeshDiff are authored by us and which are reused, we have stated the origin of the code in each source file and in this thesis, we will use the terms *reused code* and *original code* to differentiate between the two.

2.2.2 Triangle Mesh Viewing

The *reused code* of MeshDiff supports two standard triangle mesh formats: `.obj`[24] and `.ply`[25]. It is able to load and store files in these formats and also convert between them and the internal *scene representation* (see section 2.1.1) of a triangle mesh. The *scene* can be prepared for rendering by storing its data in the vertex buffer object in the GPU. This is accomplished by calling one of the *scene's* own methods. In each frame, a rendering method is called which comprises OpenTK calls tied to a view panel showing the triangle mesh. The view panel is a custom OpenTK control placed directly in the main form of the application. This process

[21] It is worth mentioning that when the clusters supplied to this visualizer correspond to the original vectors (i.e. no clustering was performed), a per-vertex visualization is generated similar to those in MeshLab or Morphome3cs (see the introduction).

[22] Cignoni et al. [2008]

[23] CGG MFF UK [2015]

[24] Wavefront .obj file format. Not to be confused with object files used in compilers.

[25] Polygon File Format.

can be configured by a set of toggles which can change the viewing mode to wireframe, enable shading, etc.

Interactivity is handled by a class called `Trackball` which intercepts mouse events from the view panel and modifies the model matrix used in the rendering process.

We have added several modifications to this basic setup to support the rendering of visualizations of the difference between two triangle meshes. Here are the most significant ones:

- We have added a second viewing panel for the inverted visualization on the *reference mesh* (see section 2.1.5). This required duplicating buffer objects, `Trackball` instances and also fields which stored the loaded *scenes*.
- Based on that we have added the option to either control both views separately, or to control both of them at the same time. This is done by first modifying the `Trackball` instance according to the mouse event associated with the view panel in was intended for and then copying the resulting rotation matrix from that instance to the second instance. Because the coordinates in the mouse events are tied to their view panels, simply routing the events to the second view panel would only work if both panels had the same dimensions. This solution is more general and bypasses coordinate recalculation.
- When visualizations are to be created, the visualizer objects are supplied with copies of the raw *scenes* to bake the visualizations to (see section 2.1.5). These visualization *scenes* are then stored alongside the raw scene. This allows users to quickly switch between viewing visualizations and raw *scenes*. When a new visualization is requested, another pair of copies is be made out of the raw *scenes* and when the visualization is baked, it replaced the old visualization pair.
- For the case when arrows are part of a visualization, we have added the option to toggle wireframe view independently for them and the underlying *scene*. The arrows therefore need to be stored in a separate *scene* and the option to render two *scenes* at once was added to the rendering methods. In the visualization process, this is achieved by baking the color visualization into the underlying *scene* and the arrow visualization into an empty *scene* (see section 2.1.5).

To summarize, next to the pair of raw *scenes* we also store four additional *scenes*: one pair of raw *scenes* with a color visualization and one pair of arrow *scenes*. See Fig. 2.4 for an illustration of this.

2.2.3 User Interface

We have extended the *reused* user interface by adding functionality which allows the user to configure the visualization before it is generated. For the complete description of the configuration options, see attachment A.2. The user can also save this configuration to a file or load it from a file. See attachment A.3 for more details on this.

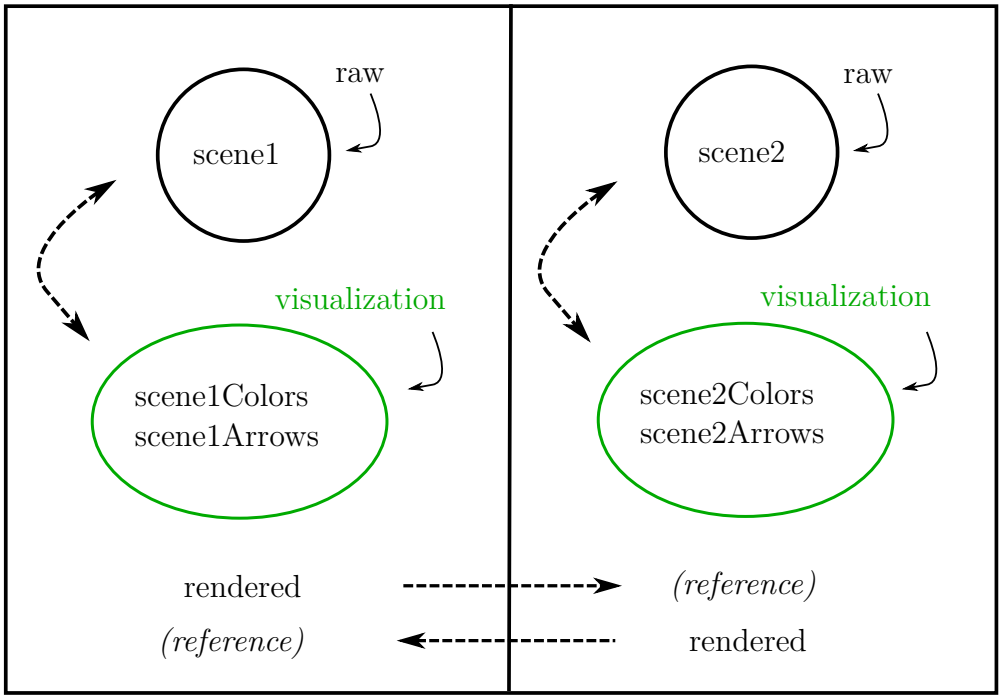


Figure 2.4: An illustration of the two view panels of MeshDiff and various scenes which can be displayed in them

We have decided to duplicate clustering parameters and have a separate set for clustering in color visualizations and arrow visualizations. This makes MeshDiff more flexible because if there was only one set, it would not be possible to visualize using unclustered color and clustered arrows at the same time.

The most important clustering and visualization parameters can be modified directly from the main form where both triangle meshes can be interactively viewed as well. Metric type and all other parameters have their separate dialog windows.

There are four places in the code where this configuration is held. The `Trackball` class remembers the model matrix and zoom value for each of the displayed triangle meshes. Currently chosen difference metric type is stored in a single variable. Clustering parameters have their own dedicated objects and so do the visualization parameters which control the appearance of the visualizations.

Apart from being able to pass the parameters to the `DiffAlgo` class in a compact way, this approach also has the advantage that each class encapsulating a certain group of parameters is also able to write them to a file and it is also able to initialize itself from a file[26].

Each of the parameter classes implement individual parameters as properties and handle value checks. Exceptions thrown by these checks are not being caught, however, because the user interface is designed in such a way that invalid values

[26]Metric type is stored in a variable but there is a dedicated class called `Metric` in the program which handles metric computation and on top of that is also able to handle the input/output operations.

cannot be assigned. Invalid assignments from code are then correctly detected and an exception is thrown in such cases.

See Fig. A.1 for the modified user interface.

2.2.4 Visualization Infrastructure

The visualization is purely handled by our *original code*. Once the user has finished their configuration and started the visualization generation, an asynchronous job is initialized to handle this process in a user-friendly way.

The behavior of the job is fully determined by a configuration package class called `JobParameters`. Here is the full list of its fields:

- `DiffForColors` - A `DiffAlgo` instance which handles clustering for color visualization
- `DiffForArrows` - A `DiffAlgo` instance which handles clustering for arrow visualization
- `Metric` - Metric type
- `ClusteringParametersArrows`
- `ClusteringParametersColors`
- `VisualizationParameters`
- `VisualizerColor` - Color visualizer object
- `VisualizerArrow` - Arrow visualizer object
- `Scene1Color` - The *rendered scene* for color visualization baking
- `Scene2Color` - The *reference scene* for inverted color visualization baking
- `Scene1Arrows` - The *rendered scene* (or an empty *scene*) for arrow visualization baking
- `Scene2Arrows` - The *reference scene* (or an empty *scene*) for inverted arrow visualization baking

The are two `DiffAlgo` instances to support different clustering for arrow visualization and color visualization as described in section 2.2.3.

Once a job is initialized with this package, its `Run()` method can be assigned to a thread. A job operates directly with the `DiffAlgo` class, initializes it and calls its methods according to the configuration package. The thread is started together with a dialog window which shows progress and enables the user to cancel the process. Once the visualization process finishes, the program checks whether it has finished successfully or not and assigns based on that either renders the computed visualization *scenes* or reports an error and renders the previously shown *scenes*.

Checking for visualization cancellation is done mainly during the clustering loop because it is the most time-consuming part of the program. It is also done

after the visualization is generated because otherwise the process would finish even when the user has requested a cancellation. When a process is canceled, all the intermediate data related to that process is deleted and the previously shown *scenes* are rendered.

3. User Study

In this chapter, we will introduce the user study and describe how it was conducted.

3.1 Setting

The goal of our study was to simulate use cases of both new and existing visualizations of the corresponding vertex distance metric and assess their performance in those use cases. Other difference metrics are out of scope of this thesis. Our general approach was to choose a pair of triangle meshes, find a specific difference between them and then generate several visualizations which would then help the user find the same difference. This corresponds to the situation where a certain difference metric is computed and needs to be presented in a clear way. In some cases, we asked a question we did not know the answer to. This corresponds to the situation where visualizations directly help users devise a result. Good visualizations therefore had to either help the participant answer correctly or create a clear agreement among the majority of participants.

We have prepared 10 questions, each of them tied to a specific pair of triangle meshes and a specific difference between them. Each question was accompanied by four different visualizations. We have split all the visualizations into four groups of ten such that each group contained one visualization for each question. Participants were assigned to one of the groups randomly at the beginning of the study and answered the questions with the help of visualizations from their group (see Fig. 3.1 for the schema of the study). At the end, we have compared the answers to each question among the four visualization groups.

Another criterion we used to compare visualizations was the time it took the participants to answer. The best visualizations should allow not only for correct but also for quick answers. This would mean that they contain enough information (correctness) and represent it clearly (speed). Participants did not know that time was measured.

3.1.1 Data

All triangle meshes used in the study were 3D scans of human faces kindly provided to us by the Laboratory of 3D Imaging and Analytical Methods of the Faculty of Science at Charles University. Overall, we have used six distinct mesh pairs to be able to study a larger range of difference and also to make the study more interesting to the participants.

3.1.2 Visualizations

For each question, we have chosen the four visualizations to be presented to the participants according to the following rules:

- Meshes without a visualization have to be included to allow for the contribution of visualizations in general to be measured.

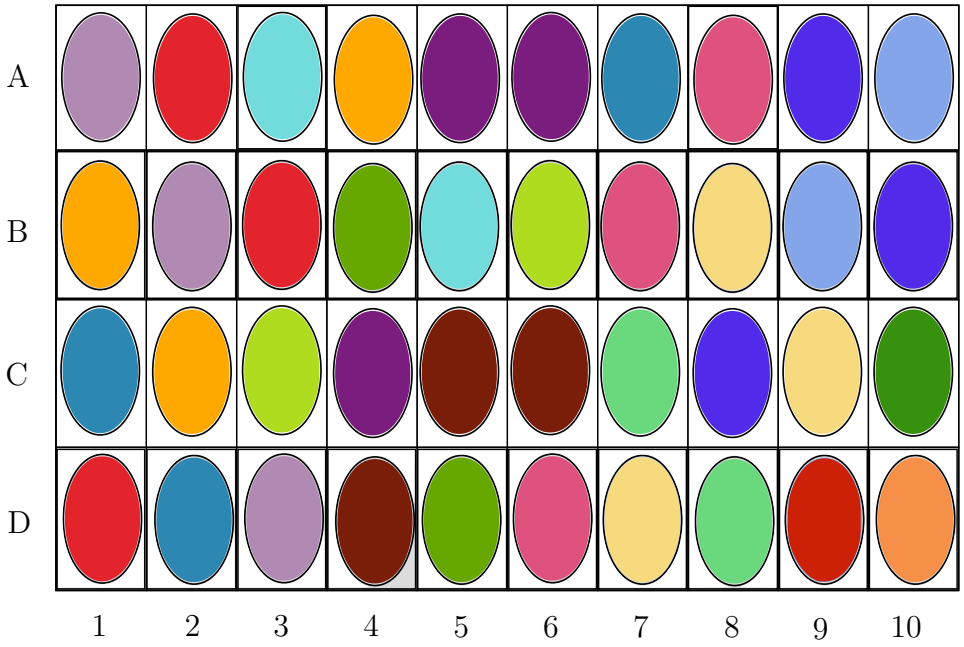


Figure 3.1: A schema of the study. Columns represent questions and rows represent groups. In each cell there is a visualization. Visualizations cannot repeat in a column but they can repeat in a row. Each participant is assigned to a row

- The visualization which we expected to be the most suitable for the given task (see the expectations in section 1.4) has to be included.
- The remaining visualizations should be as distinct as possible in order to account for unexpected results.
- At least one existing and one new visualization type have to be included to allow for the differences between their performance to be captured.
- Each visualization type presented in this thesis has to be present at least once in the whole data set.

After applying the visualizations to the six chosen mesh pairs, we have obtained 25 distinct pairs.

3.1.3 Questions

We have created the following types of questions based on the way they can be answered:

Left/Right In these questions, participants were supposed to answer by clicking on one of the meshes.

Yes/No In these questions, participants were supposed to answer by choosing either "Yes" or "No" from a drop-down menu.

Direction In these questions, a specific direction selected from a drop-down menu was accepted as an answer.

Location In these questions, participants answered by choosing one of six predefined locations in a mesh from a drop-down menu.

Each question type also provided the possibility to answer "Not sure" when the participant did not understand the question or if the question was too difficult for them.

3.1.4 Program

We have modified MeshDiff for the purposes of the study. The only features which remained were the ability to load and interactively view pregenerated visualizations and to modify the view by toggles described in attachment A.4.4. We have added a tutorial at the beginning which gives the full instructions to the participant, therefore no prior knowledge of the program nor the subject of the study is required. The tutorial also lets the participant answer one sample question.

We will now describe the course of one session of the study. At the beginning, the participant completes the tutorial. The program then provides feedback to the sample answer and explains why it was correct or incorrect. After that, 10 other questions are presented. Each of them begins with a description of the visualization currently being shown. When the participant agrees they have understood the description, the program starts to secretly measure time. When the answer is chosen and confirmed, the time elapsed is saved along with the value of the provided answer and the next question is presented.

The study program runs locally without the ability to connect to the Internet and the participant has to manually upload the file containing their answers to a provided URL after they have finished.

3.1.5 Participants

Total of 37 volunteers of various backgrounds, ages and nationalities have participated in the study. Due to this number being relatively low, we were not able to analyze the answers of domain experts and the general public separately, nor were we able to make any other distinction. The primary intention, however, was to target the general public because we believe that this is aligned with the purpose of visualizations as a tool to make the understanding of data or concepts as easy as possible.

3.2 Results

We enclose the complete results of the study along with a guide on how to interpret them in attachment A.5.

4. Discussion

In this chapter, we will conclude the user study with a discussion of its results and suggest possible future improvements both to the study and to mesh difference visualizations in general.

4.1 Significant Results

The study has yielded three significant results which show the contribution of visualizations in general and also a good performance of the newly presented visualizations of vector-based difference metrics.

4.1.1 The Overall Contribution of Visualizations

The answers to question 7 (see attachment A.5.7), which requires participants to click on the face which has larger cheekbones, has shown an overwhelming dominance of visualizations over raw triangle meshes. Any of the three types of visualizations presented helped participants to answer correctly, whereas when no visualization was shown, answers were almost equally distributed among all the possible options. Moreover, when participants were shown meshes without a visualization, it took them longer to arrive at an answer. Similar effect could be observed in the answers to question 8 (see attachment A.5.8).

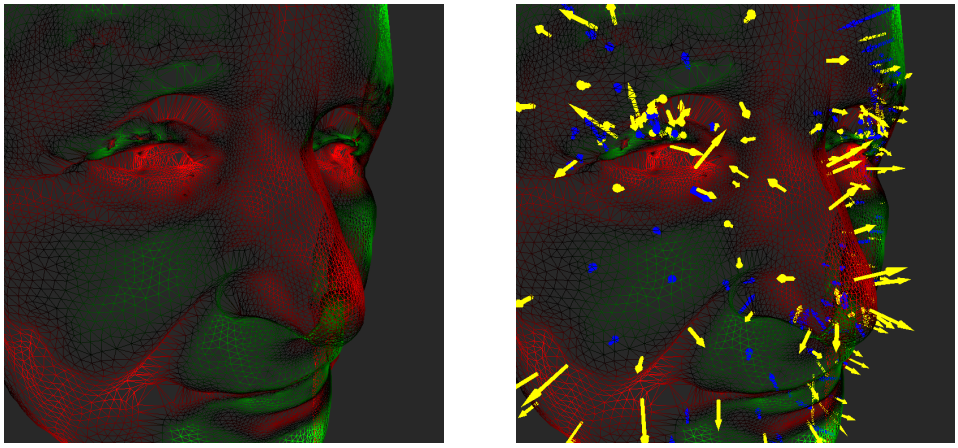
4.1.2 The Contribution of Arrows

Question 9 (see attachmentA.5.9) has shown a contribution of arrows which we did not include in our expectations (see section 1.4). The question was aimed at the absolute value of the difference and where this value was the smallest. Thresholding performed well as expected but most correct answers were given when arrows were displayed. This shows that the length of an arrow is much more suggestive than the color scale when capturing the absolute value of a difference metric.

See Fig. 4.1 for a comparison between color-based and arrow-based visualization of the absolute value of a difference metric.

4.1.3 The Contribution of Thresholding

In question 3 (see attachment A.5.3), we tried to assess the performance of thresholding in the visualization of the largest difference, a task which it was expected to excel at (see section 1.4). We found that when the threshold was set just under the largest metric value and clusters whose area was too small were excluded too[27], it was very easy for participants to identify the correct location. On the other hand, when a basic color visualization without thresholding was shown, the spread of answers was significantly larger and participants took much longer to answer. Besides this result, there are two interesting points associated with this



(a) Color-based (b) Arrow-based, combined

Figure 4.1: MeshDiff - Visualizations of corresponding vertex distance projected into surface normal

question which we have to mention here:

- The answers given for visualization A.9c clearly show that participants were unsure what is meant by “left” and “right”, even though this was explained at the beginning of the study. Because of this, we considered both “left cheek” and “right cheek” to be correct answers for this particular visualization.
- Answers provided when no visualization was shown suggest that this question is beyond the power of basic difference metrics we worked with in this thesis. The reason for this is that the term “difference” can be interpreted more globally. Humans in general are very sensitive to facial features and tend to see the difference between them globally, unlike our difference metrics which are local. This could also be observed in the answers to question 9 (see attachment A.5.9).

We will further address these and other problematic points in the following sections.

4.2 Study Improvements

Because of time constraints, we were not able to improve the study and conduct it again in order to obtain a more complete and thorough comparison between color-based and arrow-based visualizations. A more detailed study would also be able to capture the differences between various clustering methods and clustering parameters which was beyond the resolving power of our study. In this section, we propose several improvements which might help to create such a study.

[27] In our data set, areas by the edge of a mesh are not very representative of the original object they represent because of the way they are cut. At the same time, however, they are very dense and have very high and variable metric values. This results in small clusters which can be easily excluded by area thresholding.

The most obvious limitation of our study was its scale. However, several other circumstances have further hindered our findings:

- The overall presentation of the study did not attract enough attention. For the next study, we recommend to design a web application with a modern and responsive interface which would allow for the sessions to be shorter, easier and more accessible to the participants. Being presented with a modern web application also adds to the feeling that the subject of the study is modern and relevant.

- A special attention should be given to the introduction of the study. Our results, such as A.5.3, have shown that even though the way directions are given was explained in the introductory text, the participants had problems with this. This introduced error into the provided answers. We recommend to create an interactive introduction, rather than text-based, to increase its effectiveness.

- The last point related to user experience, which we believe is key, is the way questions are worded. Questions 4 (A.5.4) and 5 (A.5.5) have proven to be too complicated because of the amount of directions and concepts included in them. An example of a good question is question 7 (A.5.7).

- Apart from user experience issues which introduced error, we have also noticed a problem with the focus of the study. Our study has mixed questions related to visualizations (e.g. question 6 A.5.6) with questions related to metrics (e.g. question 3 A.5.3). On one hand, this has allowed us to capture the need for more sophisticated difference metrics, on the other hand, it has not contributed to the comparison between various visualization types which is the core of the this thesis.

- Another problem related to this is the choice of data. As was mentioned in section 4.1.3, the difference metrics we have studied are not suitable for capturing the difference between human faces which people usually tend to see. While these metrics might still be useful in certain cases, this is not very clear from our study because of the data set chosen. We recommend to use more neutral objects in order to eliminate this phenomenon.

4.3 Method Improvements

In order to improve the new visualizations presented in this thesis, a more detailed user study, as outlined above, should be conducted. However, during the course of our work we have discovered other possible improvements which are out of the scope of this thesis. We present these here.

- The clustering process used in our visualizations (see section 1.3.3), more specifically the dendrogram, allows for an interactive visualization to be created. The user would be presented with the visualization of only one cluster covering the whole mesh. By clicking on a cluster, its children in the dendrogram would replace the cluster in the visualization. This would

result in a clustering of adaptive detail depending on the specific needs of the user.

- As mentioned in section 4.1.3, difference metrics included in this thesis are not sufficient in certain cases, such as when measuring the difference between facial features. A possible next step would be to devise global difference metrics which would better correspond to the idea of shape differences that humans have. As a consequence, clustering would not be required because information would be inherently reduced.

Conclusion

In this thesis, we have addressed the problem of the visualization of the difference between two triangle meshes. A common way of approaching this problem is to first compute a difference metric, for example the distance between corresponding vertices of the two meshes. This metric is then visualized. Existing visualizations mostly encode the metrics into mesh vertex color which may lead to information loss, especially when the metrics are multidimensional. We have proposed a new way of visualizing these metrics focusing on the ability to display multidimensional information and to group similar information together in order to prevent cluttering.

We have used the following two metrics (see Fig. 1.1):

- Corresponding vertex distance
- Corresponding vertex distance projected into surface normal

The proposed visualizations are the following (see section 1.4):

Arrows Clustered metric vectors are visualized using arrows.

Cluster color Mesh vertices belonging to a given cluster are colored based on the properties of the cluster.

Thresholding We have introduced a thresholding which excludes all information which is not significant enough.

Combined Arrow-based and color-based visualizations can be combined to use the best of both worlds.

We have implemented these visualization in an experimental application called MeshDiff (see attachment A.4) which we used to demonstrate the visualizations. The application can also be used more widely by people who want to create visualizations of their own data, export them, and present them in publications.

We have also conducted a user study which helped us evaluate the new visualizations as well as existing ones. It has shown that our visualizations are superior in certain areas, especially when comparing the absolute values of a metric and when underlining the most significant differences (see section 4.1).

Future Work

Our study has also shown several deficiencies, both in its own design and in the visualizations. We have discovered three main areas of potential future improvement.

- A better organized and more thorough study would be able to further assess the newly proposed and existing visualizations. Our study did not include the comparison of various clustering settings, for example. We believe these results would help to further improve these visualizations (see section 4.2).

- Our study has suggested that existing difference metrics (which we have also used) are limited by their locality. Examining the differences between two human faces has manifested the need for more sophisticated global difference metrics (see section 4.3).
- A new adaptive interactive visualization can be built on top of our clustering method where users would choose which areas they want to be visualized in more detail. This would further help to suppress unnecessary information (see section 4.3).

Bibliography

Cahn and Hilliard. Free energy of a non-uniform system i. interfacial free energy. *The Journal of Chemical Physics*, 1958.

CGG MFF UK. *Morphome3cs II*. Charles University in Prague, Czech Republic, 2015. URL `http://www.morphome3cs.com`.

Cignoni, Rocchini, and Scopigno. Metro: measuring error on simplified surfaces. Technical report, Istituto per l'Elaborazione dell'Informazione - Consiglio Nazionale delle Ricerche, Pisa, Italy, 1998.

Paolo Cignoni, Marco Callieri, Massimiliano Corsini, Matteo Dellepiane, Fabio Ganovelli, and Guido Ranzuglia. MeshLab: an Open-Source Mesh Processing Tool. In Vittorio Scarano, Rosario De Chiara, and Nils Ugo ErraSvartholm, editors, *Eurographics Italian Chapter Conference*, pages 129–136. The Eurographics Association, 2008.

CloudCompare. *Distances Computation*, 2015. URL `http://www.cloudcompare.org/doc/wiki/index.php?title=Distances_Computation`. Retrieved on 05/04/2018.

CloudCompare. *CloudCompare*, 2018. URL `http://www.cloudcompare.org/`. GPL Software, version 2.9.1.

Du and Wang. Centroidal voronoi tessellation based algorithms for vector fields visualization and segmentation. In *VIS '04 Proceedings of the conference on Visualization '04*, 2004.

Garcke et al. A continuous clustering method for vector fields. In *VIS '00 Proceedings of the conference on Visualization '00*, 2000.

Peng et al. Mesh-driven vector field clustering and visualization: An image-based approach. *IEEE Transactions on Visualization and Computer Graphics*, 2012.

Pikora. Mobile 3d mesh viewer. Bachelor's thesis, Charles University, Prague, 2017. Available at http://hdl.handle.net/20.500.11956/86131.

Rossignac, Safonova, and Szymczek. Edgebreaker on a corner table: A simple technique for representing and compressing triangulated surfaces. In *Hierarchical and Geometrical Methods in Scientific Visualization*. Springer Science & Business Media, 2003.

Telea and van Wijk. Simplified representation of vector fields. In *VIS '99 Proceedings of the conference on Visualization '99: celebrating ten years*, 1999.

List of Figures

A. Attachments

A.1 Electronic Attachment

This is the structure of the electronic attachment of this thesis:

- `doc/` - contains this document in a PDF
- `MeshDiff/` - contains the binaries of MeshDiff
- `src/` - contains the complete source code of MeshDiff and the study program
- `study/` - contains the user study as distributed to participants

We do not enclose any test data. This data is available at request from RNDr. Josef Pelikán or from the Laboratory of 3D Imaging and Analytical Methods of the Faculty of Science at Charles University.

The code and the binaries of MeshDiff can also be found in the following GitHub repository:

- `https://github.com/honzukka/MeshDiff.git`

A.2 Parameter Description

As mentioned in section 2.2.4, the proposed visualizations can be configured by a set of clustering parameters, a set of visualization parameters and a few other parameters. We will now provide an overview of all of them.

A.2.1 Clustering Parameters

- **Cluster Count**
  - Determines the number of clusters to be retrieved from the dendrogram (see Fig. 1.4) and used for visualization. If the dendrogram is a tree, any valid number of clusters is guaranteed to cover the whole data set. If the dendrogram is a forest (see section 1.3.3), certain parts of the data set may remain uncovered by the chosen clusters.
  - Valid values: $[1, S]$ where S is the number of vertices of one of the *scenes*
- **Direction Significance**
  - Determines how large the direction weight coefficient in the error function (see Eq. 1.1) will be. See section 1.5.1 for details. It is only meaningful in combination with all the other significance parameters[28].
  - Valid values: $[0, 100]$
- **Magnitude Significance**
  - Determines how large the magnitude weight coefficient in the error function (see Eq. 1.1) will be. See section 1.5.3 for details. It is only meaningful in combination with all the other significance parameters[29].
  - Valid values: $[0, 100]$
- **Position Significance**
  - Determines how large the position weight coefficient in the error function (see Eq. 1.1) will be. See section 1.5.2 for details. It is only meaningful in combination with all the other significance parameters[30].
  - Valid values: $[0, 100]$
- **Resolution Significance**
  - Determines how large the mesh resolution weight coefficient in the error function (see Eq. 1.1) will be. See section 1.5.4 for details. It is only meaningful in combination with all the other significance parameters[31].
  - Valid values: $[0, 100]$

A.2.2 Visualization Parameters

- **Arrow Height Minimum Scale**
  - Determines the height of an arrow representing the lowest metric value present in the data set. This value will multiply the height of the default arrow (see section 2.1.5) and therefore does not represent the absolute value of the minimum arrow height.
  - Valid values: $[0.1, 10]$
- **Arrow Height Maximum Scale**
  - Determines the height of an arrow representing the highest metric value present in the data set. This value will multiply the height of the default arrow (see section 2.1.5) and therefore does not represent the absolute value of the maximum arrow height.
  - Valid values: $[0.1, 10]$
- **Arrow Width Minimum Scale**
  - Determines the width of an arrow representing a cluster with the smallest area out of all visualized clusters. This value will multiply the width of the default arrow (see section 2.1.5) and therefore does not represent the absolute value of the minimum arrow width.
  - Valid values: $[0.1, 10]$
- **Arrow Width Maximum Scale**
  - Determines the width of an arrow representing a cluster covering the whole data set. This value will multiply the width of the default arrow (see section 2.1.5) and therefore does not represent the absolute value of the maximum arrow width.
  - Valid values: $[0.1, 10]$
- **Arrow Outwards Color**
  - Determines the color of arrows pointing *outwards* (see section 1.1).
  - Valid values: Any RGB color
- **Arrow Inwards Color**
  - Determines the color of arrows pointing *inwards* (see section 1.1).
  - Valid values: Any RGB color

[28] Here is the conversion between *significance* and *weight*: Consider *significance* values $s_d, s_m, s_p, s_r \in [1, 100]$ and *weight* values $k_d, k_m, k_p, k_r \in [0, 1]$. We need $k_d + k_m + k_p + k_r = 1$, therefore $k_d = s_d/(s_d + s_m + s_p + s_r)$ and similarly for all other weights.

[29] See footnote 28.

[30] See footnote 28.

[31] See footnote 28.

- **Color Metric Outwards**
  - Determines the color hue of vertices which are assigned a metric vector pointing *outwards* (see sections 1.1 and 1.4.2).
  - Valid values: Any RGB color
- **Color Metric Inwards**
  - Determines the color hue of vertices which are assigned a metric vector pointing *inwards* (see sections 1.1 and 1.4.2).
  - Valid values: Any RGB color
- **Color Diff Threshold**
  - In absolute mode (see section 1.4.2), all vertices with an associated metric vector longer than this value will receive the brightest color.
  - Valid values: $[1, \infty)$ (In the MeshDiff UI this is limited by the diameter of the whole scene.)
- **Disabled Color**
  - The color of vertices which were excluded from the visualization by thresholding (see section 1.4.3).
  - Valid values: Any RGB color
- **Disabled Threshold Length**
  - All vertices with a shorter associated metric vector will be excluded from the visualization (see section 1.4.3).
  - Valid values: $[1, \infty)$ (In the MeshDiff UI this is limited by the diameter of the whole scene.)
- **Disabled Threshold Size**
  - All vertices which are part of a cluster whose area is smaller than this value will be excluded from visualization (see section 1.4.3).
  - Valid values: $[1, \infty)$ (In the MeshDiff UI this is limited by the diameter of the whole scene squared.)

A.2.3 Other Parameters

The rest of the parameters comprise the viewing angle and zoom of both *scenes* and also the metric type, clustering type and visualization type. They are used to configure the mesh viewer in MeshDiff and they also determine how the `DiffAlgo` class is initialized (see section 2.1.2).

- **Zoom Left & Right**
  - Determines the zoom factor of the mesh view.
  - Valid values: Not limited (The default value is 1.)

- **Rotation Left & Right**
  - Determines the viewing angle of a mesh.
  - Valid values: Any 4x4 rotation matrix
- **Metric**
  - Determines the metric type used in difference computation. (See section 1.1.)
  - Valid values: {Distance, NormalProjectedDistance}
- **Clustering Type**
  - Determines the type of clustering used in the visualization process. (See section 2.1.4.) Separate for color and arrow visualizations.
  - Valid values: {None, Simple, Signed}
- **Color Visualization**
  - Determines the type of color visualization used. (See section 1.4.2.)
  - Valid values: {None, Random, Relative, Absolute}
- **Arrow Visualization**
  - Determines whether arrow visualization is used.
  - Valid values: {Yes, No}

A.3 Parameter Loading and Storing

In this section, we describe how MeshDiff can write its configuration to a file and also read it from a file. First, we will talk about the file format. It is a very simple custom format designed to store key/value pairs and to allow for easy reading and writing. Then we will mention how the process of reading and writing is realized in the code.

A.3.1 Format

Each file is divided into sections which are delimited by a special line containing the name of the section and a customizable decoration. What follows are key/value pairs, one pair per line. Here is an example of a full configuration file:

```
---Trackball Left---
zoom=1
rotation=1,0,0,0;0,1,0,0;0,0,1,0;0,0,0,1

---Trackball Right---
zoom=1
rotation=1,0,0,0;0,1,0,0;0,0,1,0;0,0,0,1

---Metric---
metric=distance

---Clustering Parameters Arrows---
clusterCount=20
directionSignificance=15
positionSignificance=25
magnitudeSignificance=10
resolutionSignificance=0

---Clustering Parameters Colors---
clusterCount=20
directionSignificance=15
positionSignificance=25
magnitudeSignificance=10
resolutionSignificance=0

---Visualizer Parameters---
arrowOutwardsColor=1,1,0
arrowInwardsColor=0,0,1
colorMetricOutwards=1,0,0
colorMetricInwards=0,1,0
disabledColor=0.3,0.3,0.3
arrowWidthMinScale=0.5
arrowWidthMaxScale=5
arrowHeightMinScale=0.5
arrowHeightMaxScale=3
disabledThresholdLength=0
disabledThresholdSize=0
```

```
colorDiffThreshold=10

---Form Settings---
clusteringTypeArrows=None
clusteringTypeColors=None
colorVisualization=None
arrowVisualization=No
```

All lines which do not fulfill any of the following requirements are ignored:

- Starts with the decoration (in the above example this is `---`)
- Yields exactly two strings upon split at `=`

The format of keys and values is defined by the program.

A.3.2 Loading and Storing

We have written `ParameterWriter` and `ParameterReader` classes which can be used in a similar way other I/O classes are used in C#.

`ParameterReader` is initialized with a file path, a section name and optionally a decoration different to the default `---`. Then a `ReadPair()` method can be called in a loop and key/value pairs are returned as tuples. When there are no more pairs in the given section, `null` is returned. All objects of this class should be disposed of at the end of their lifetime.

`ParameterWriter` is initialized in a similar manner and provides two methods: `WritePair()` and `WriteEmptyLine()`. Both methods write to the given file and they always append to file. Upon the first call, a section of the given name is created and all subsequent calls write key/value pairs under that section. `WritePair()` can only accept arguments of type `string`. This class also implements the `IDisposable` interface.

A.4 MeshDiff User Documentation

This attachment is structured differently than the rest of the thesis because it aims to be a standalone introductory text to the purpose of MeshDiff and its structure from a user point of view.

For clarity, each section of this chapter represents an answer to a question the user may ask when consulting the user documentation. The text also addresses the user directly. At the beginning, however, we provide a short introduction of what MeshDiff is.

A.4.1 About MeshDiff

MeshDiff is a graphical program running on Windows operating systems which allows its users to interactively view two homologous triangle meshes[32] in `.ply`[33] or `.obj`[34] format and visualize the difference between them. MeshDiff has various color-based, arrow-based and combined visualizations available. Once the user has found a visualization which suits their intentions the best, there are two options of saving the visualization:

- It can be exported as a `.ply` file and subsequently loaded in any `.ply` viewer, for example in a mobile application called MorphoView and presented in Pikora [2017]
- Its configuration can be saved. In this case, MeshDiff can load the configuration and generate the very same visualization under the very same viewing angle later

A.4.2 How do I obtain the correct input data?

As mentioned above, MeshDiff requires the input triangle meshes to be homologous. Homologization of two arbitrary triangle meshes can be done in publicly available software, for example in Morphome3cs [CGG MFF UK, 2015]. The data we have used in the thesis was kindly provided to us by the Laboratory of 3D Imaging and Analytical Methods of the Faculty of Science at Charles University where it can be requested from.

It is advisable that all input triangle meshes are connected, i.e. it is possible to get from any vertex v_1 to any vertex v_2 by traversing neighboring edges. If the input mesh is not connected, certain visualizations will only consider one connected component of the mesh.[35]

A.4.3 How do I begin?

Once you have two homologous triangle meshes you wish to compare, load one of them into the left panel and the other one into the right panel:

[32] See footnotes 1 and 5.
[33] See footnote 25
[34] See footnote 24
[35] These are visualizations where *signed clustering* is used. See section 1.3.3 for details.

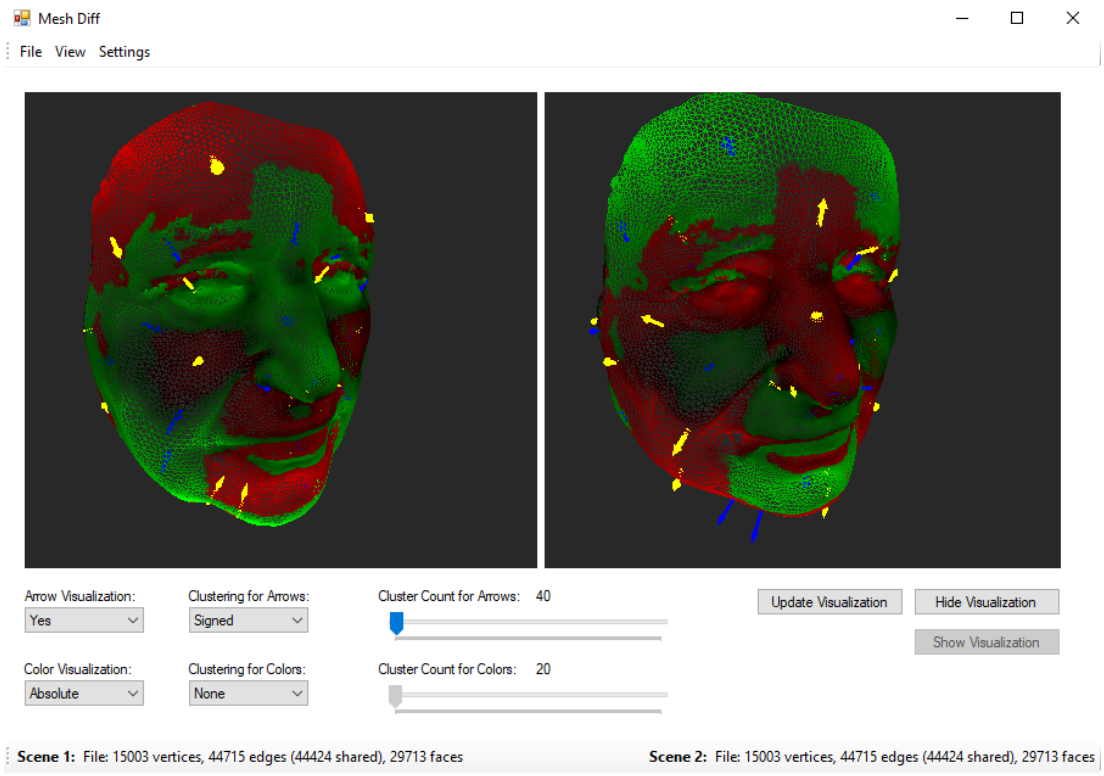


Figure A.1: MeshDiff - Main window

To load the left mesh: `File > Load Model 1`, and choose the `.obj` or `.ply` file you wish to load

To load the right mesh: `File > Load Model 2`, and choose the `.obj` or `.ply` file you wish to load

A.4.4 How do I control the view?

You can view the meshes interactively by clicking and dragging the mouse cursor over them. You can also zoom in and out using the mouse wheel.

By default, when you hover over one of the meshes while controlling the view, you change the view of both meshes at the same time. This is because the `Paired Controls` toggle is enabled. Click `View > Paired Controls` to disable it. Mesh views can be controlled separately with this option disabled.

If you want to reset the views of both meshes to their default position, click `View > Reset cam`.

There are other toggles in the `View` menu, all of which modify the view in a certain way. Here is their description:

Axes Draws the x, y and z axes into the view for better orientation

Smooth Interpolates mesh color in triangles to makes the surface look smooth. When disabled, each triangle is assigned only one color

Wire When enabled, mesh triangles are not filled which results in a wireframe rendering of the mesh

Visualization Wire A separate wire toggle for arrow visualizations

2-sided When disabled, only the side of triangles which is assigned a normal is drawn. This is useful for example when rendering half-enclosed meshes of human faces where viewing the face from the "inside" creates an unpleasant effect

GLSL Enables shaders. When shading is on, meshes look less flat

Ambient When shaders are enabled, this option adds ambient light. Ambient light is a sourceless light illuminating all parts of the mesh uniformly. In reality, this roughly corresponds to light which has been reflected many times and therefore illuminates even surfaces which are not facing a light source

Diffuse When shaders are enabled, this option adds diffuse reflections. Diffuse reflections occur in objects with a rough surface where light is reflected in many unpredictable directions

Specular When shaders are enabled, this option adds specular reflections. Specular reflections occur in polished objects with a prevalent direction of reflected light. They make the object look shiny

Phong When shaders are enabled, this option toggles Phong shading. This is an alternative shading algorithm which tries to hide the edges between triangles

A.4.5 How do I create a visualization?

Once you have loaded both triangle meshes and set the view according to your taste, you can create a visualization of the difference between the two meshes. Please, see chapter 1 for a thorough analysis of all the visualization that MeshDiff supports.

Basic configuration of a visualization can be done in the main window of MeshDiff (see Fig. A.2). The `Arrow Visualization` and `Color Visualization` drop-downs allow you to choose specific types of both visualizations. You can disable either of them by choosing `No` and `None` respectively, but it is also possible to combine them together.

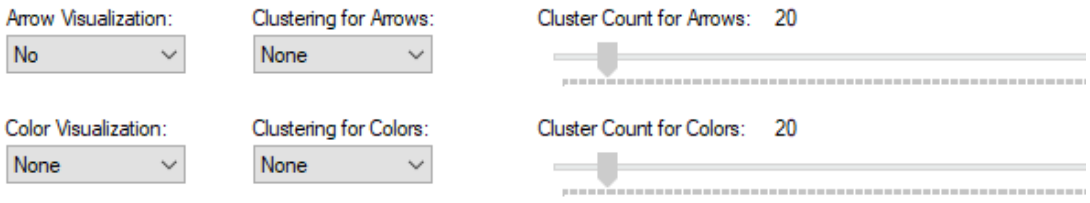


Figure A.2: MeshDiff - Creating a visualization

Once the visualization types have been chosen, optional clustering can be added to the visualization process to display more general information. Clustering can be configured independently for both arrow visualization and color visualization. The required clustering type can be set via `Clustering for Arrows`

and `Clustering for Colors` drop-downs and once a non-empty type is chosen, the corresponding trackbar will become active. The values of the trackbars determine the number of clusters which will be displayed for a given visualization type.

Please note that when selecting the `Signed` clustering, certain cluster counts do not cover the whole triangle mesh and leave black spots or show no arrows where no clusters have been chosen.

When you are ready, click `Update Visualization`.

A.4.6 How do I change clustering parameters?

If you have selected a certain clustering type for your visualization and are not satisfied with the result, MeshDiff provides a way for you to change the clustering parameters. More on what they are and what their effect is can be found in section 1.5 and attachment A.2.1.

Click `Settings > Clustering Parameters` to view the following window:

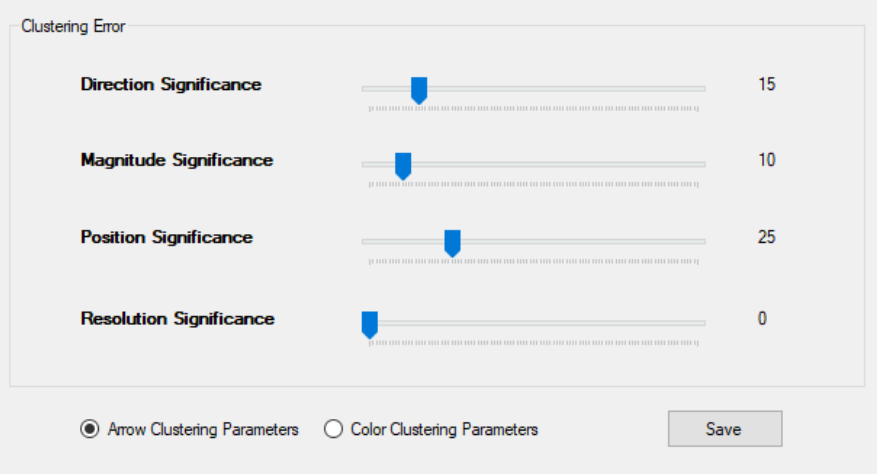


Figure A.3: MeshDiff - Clustering parameters configuration window

The checked radio button at the bottom signifies which parameters are currently being edited. The trackbar values determine the relative significance, in other words, `Position Significance` at 80 and with other values at 40 gives the same result as `Position Significance` at 40 with other values at 20.

Click `Save` to remember current values.

A.4.7 How do I change visualization appearance?

Visualization parameters, such as colors, arrow size and others can be configured by clicking `Setting > Visualizer Parameters`. The configuration window is divided into three parts. In the first part, the appearance of arrows can be configured (see Fig. A.4). The second part allows you to configure colors (see Fig. A.5a) and the third one is meant for thresholding configuration (see Fig. A.5b).

See attachment A.2.2 for a detailed description of these parameters.

Please note that the maximum range of the threshold trackbars depends on the number of vertices the currently loaded triangle meshes have. It is therefore advisable to configure these parameters after the meshes have been loaded.

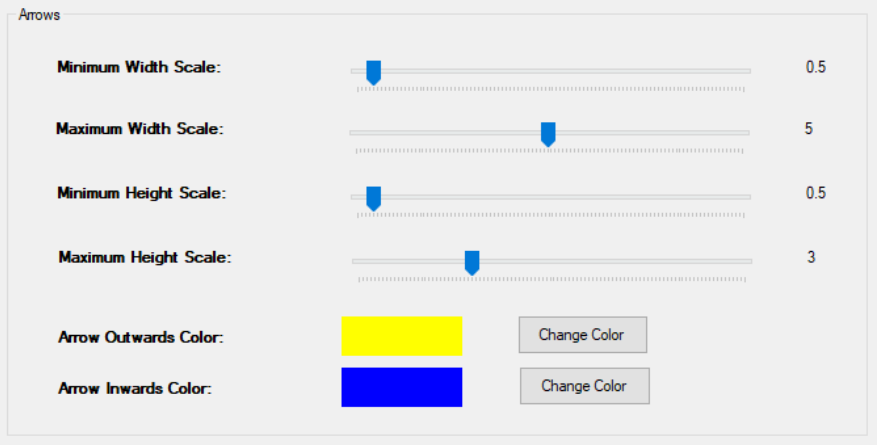
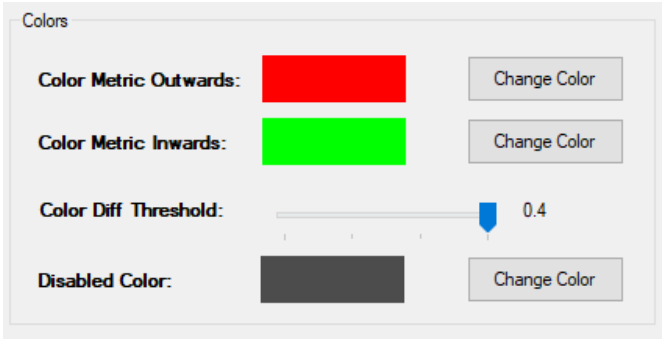
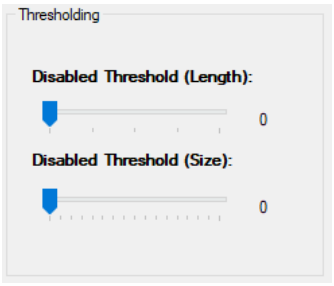


Figure A.4: MeshDiff - Visualizer parameters configuration window - Arrows



(a) Color settings



(b) Thresholding settings

Figure A.5: MeshDiff - Visualizer parameters configuration window - Colors & thresholding

Click `Save` to remember current values.

A.4.8 How do I export my visualization?

MeshDiff is able to export visualizations in the `.ply` format.

- To export the left visualization, click `File > Export Visualization 1`
- To export the right visualization, click `File > Export Visualization 2`

When you do that and your visualization contains arrows, you will be asked if you want to export arrows separately (see Fig. A.6). By clicking `No`, the complete visualization will be stored in one `.ply` file. If you click `Yes`, the underlying triangle mesh with a color visualization (if there is one) will be stored in a file called `[chosen name].ply`. The arrows will be automatically stored in a separate file called `[chosen name].arrows.ply`.

A.4.9 How do I load an exported visualization?

Because visualizations are triangle meshes stored in `.ply` files, they can be loaded using the standard `Load Model` option. However, there are two pitfalls associated with this approach:

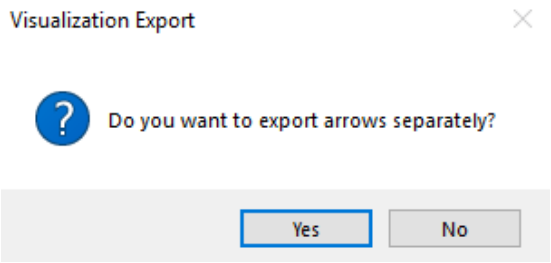


Figure A.6: MeshDiff - Export prompt

1. Arrows which are stored separately cannot be loaded into one view along with the underlying triangle mesh using `Load Model`
2. If arrows are stored together with the mesh and loaded using `Load Model`, any further attempts to generate a new visualization will also visualize the differences between the arrows which is likely undesirable

The `Load Visualization` functionality, also available from the `File` menu, at least partially solves these two issues. It opens a dialog window asking if arrows should be loaded separately. Clicking `No` will only load one chosen file, whereas clicking `Yes` will allow you to choose two files - first the underlying triangle mesh and then the associated `.arrows.ply` file.

On top of that, by using this functionality you enter a read-only mode where the `Update Visualization` button is disabled.

A.4.10 How do I save visualization configuration?

Click `File > Save Parameters` and the complete configuration of MeshDiff including mesh viewing angles and zoom will be saved to an `.ini` file.

A.4.11 How do I load visualization configuration?

Click `File > Load Parameters` and choose the desired `.ini` file. In order to utilize the configuration, triangle meshes have to be loaded separately.

A.5 User Study Results

In this attachment, the complete results of the user study are enclosed. Each page covers one question and includes the question itself, screenshots of all four associated visualizations and a table with the distribution of answers and normalized average answer times for each visualization. We will now explain how we computed the normalized average answer times.

Each participant has a different overall speed of answering. Consider participants p_i where $i = 1, 2, \ldots, n$, questions q_j where $j = 1, 2, \ldots, m$ and the answer times for each participant and question $t(p_i, q_j)$. In order to eliminate these differences, we compute the average answer times $\bar{t}(p_i, q)$ for each participant separately and created a normalization coefficient c_i where

$$\frac{1}{c_i} = \frac{\bar{t}(p_i, q)}{\max_j \bar{t}(p_j, q)}.$$

Once the answer times are normalized, we need to compare them among different visualizations. To achieve that, we compute the average normalized answer times for each question and visualization. Because of the way participants are assigned to visualization groups (see Fig. 3.1), we decompose the set of all participants into equivalence classes V_1, V_2, V_3, V_4 and compute the averages using the following formula:

$$\widehat{t}(p, q_j, V_k) = \frac{\sum_{i;p_i \in V_k} (c_i \cdot t(p_i, q_j))}{n}$$

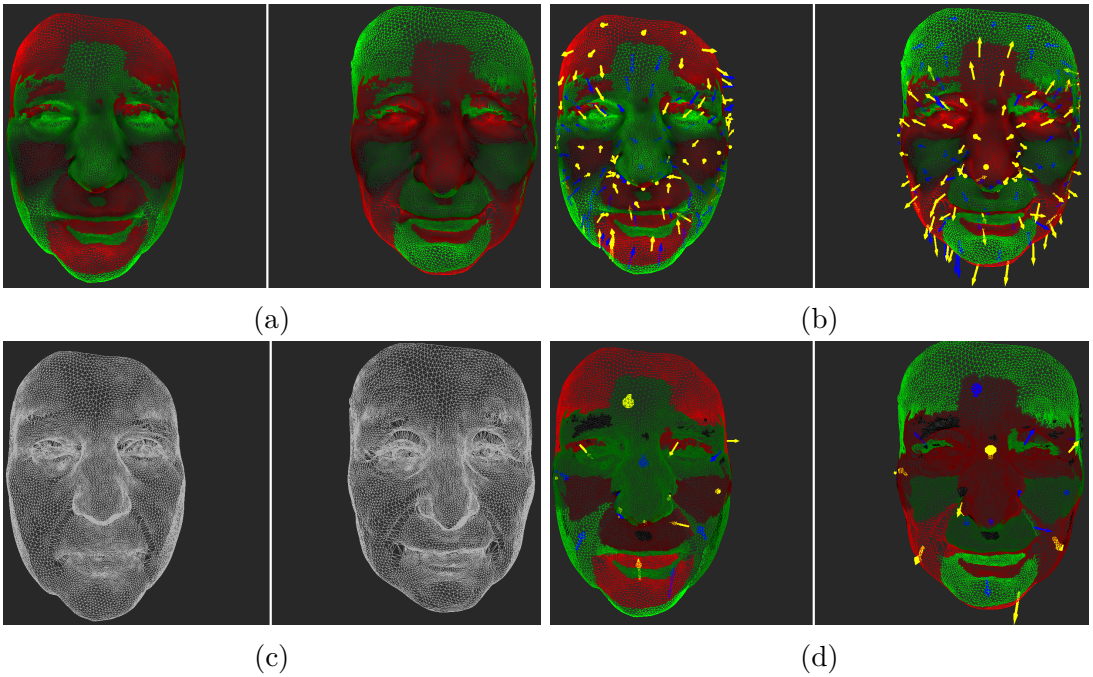
This implies that the normalized average times included in the tables are relative and therefore do not represent any standard units such as seconds or minutes.

Each question is also accompanied by the answer we thought to be correct prior to the study and a visualization we expected to be the best for finding such an answer. We provide a brief commentary related to section 4.2 in cases where our expectations were not met.

The following page begins with the results of the first question.

A.5.1 Question 1

Which face has a larger nose?



(a) (b) (c) (d)

Figure A.7: Visualizations shown for question 1

Expected best visualization: A.7b

Expected answer: *N/A*

Collected answers:

Visualization	A.7a	A.7b	A.7c	A.7d
$\widehat{t}(p, q_1, V_k)$	20.46	26.19	20.26	21.56
Answers				
Left	7	8	6	4
Right	4	2	1	2
NotSure	2	1	0	0
Total	**13**	**11**	**7**	**6**

Commentary: The majority of participants thought the left to be larger. This shows their interpretation of the term "larger" (Fig. A.7c) and that the visualizations favor this interpretation. We did not choose an expected answer because while the left nose is thicker, the right nose is longer.

A.5.2 Question 2

Does the chin stick out more to the front in the right face than in the left face?

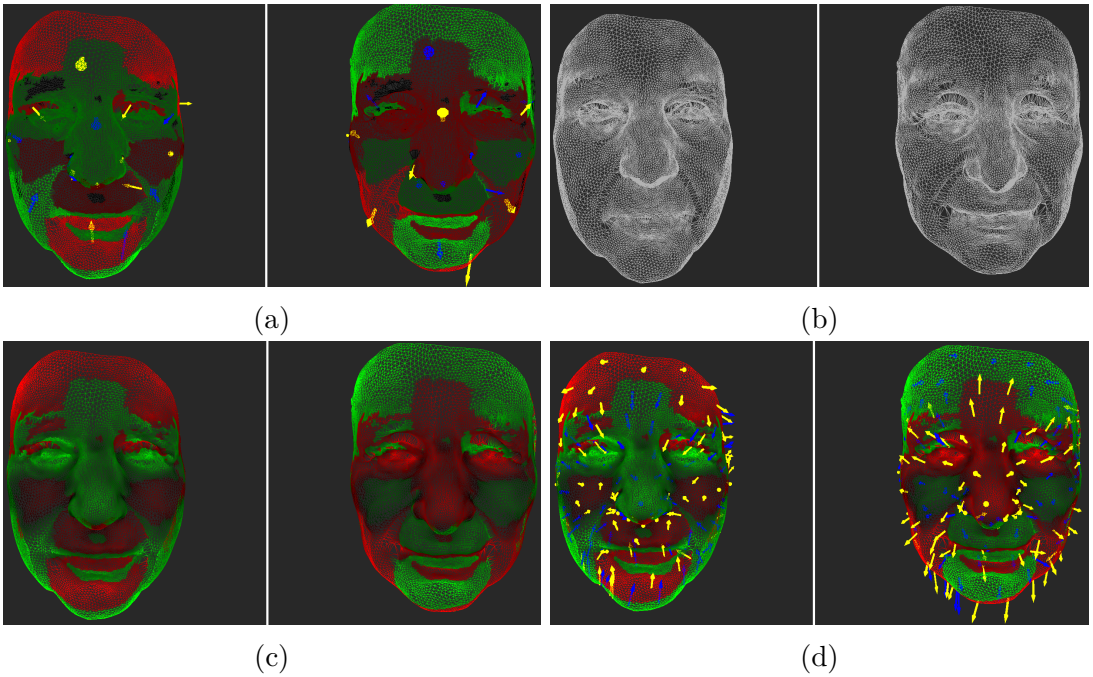


Figure A.8: Visualizations shown for question 2

Expected best visualization: A.8d

Expected answer: *No*

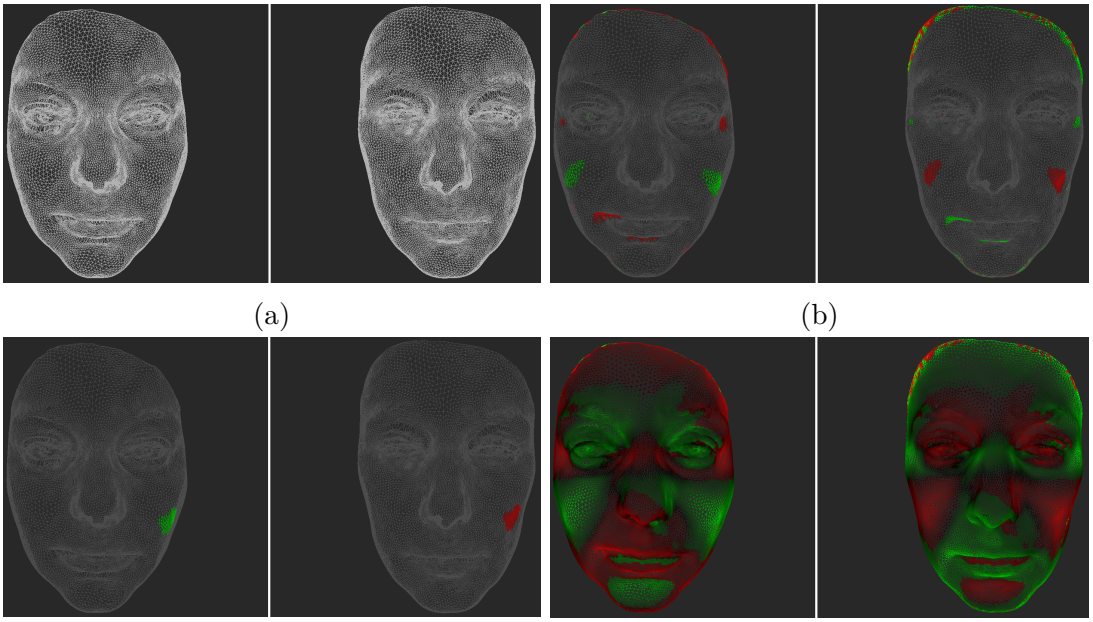
Collected answers:

Visualization	A.8a	A.8b	A.8c	A.8d
$\widehat{t}(p, q_2, V_k)$	46.62	43.08	61.56	38.90
Answers				
Yes	10	5	6	4
No	3	4	1	2
NotSure	0	2	0	0
Total	**13**	**11**	**7**	**6**

Commentary: The participants have agreed that the answer should be "Yes". We assume that when only colors were shown, the difference seemed quite large in favor of the right face. When arrows were shown, we believe that some participants thought there was no difference, while others have seen a slight difference in favor of the right face. Our original view was that there was close to no difference related to the question.

A.5.3 Question 3

Where is the most significant difference between the two faces?



(a) (b) (c) (d)

Figure A.9: Visualizations shown for question 3

Expected best visualization: A.9c

Expected answer: *Right Cheek*

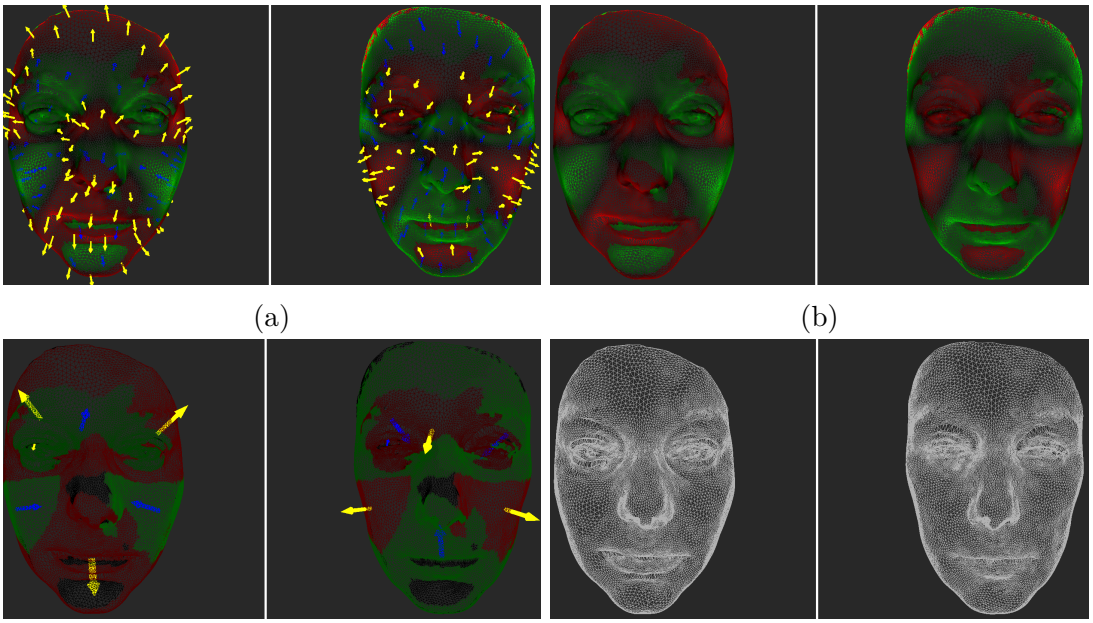
Collected answers:

Visualization	A.9a	A.9b	A.9c	A.9d
$\widehat{t}(p, q_3, V_k)$	41.87	37.36	31.29	58.84
Answers				
LeftCheek	2	1	3	1
Forehead	5	4	0	1
RightCheek	2	4	2	1
Nose	1	0	0	0
NotSure	2	2	1	1
Chin	1	0	0	0
Mouth	0	0	1	2
Total	**13**	**11**	**7**	**6**

Commentary: Based on the answers provided for visualization A.9c, we assume that the participants have confused "left" and "right" in this question. Visualization A.9b did not exclude areas close to the edge of the mesh where a lot of error is accumulated by the way the mesh is cut.

A.5.4 Question 4

When examining the differences moving from the left face to the right face, what is their main direction in the area below the nose?



(a) (b) (c) (d)

Figure A.10: Visualizations shown for question 4

Expected best visualization: A.10c

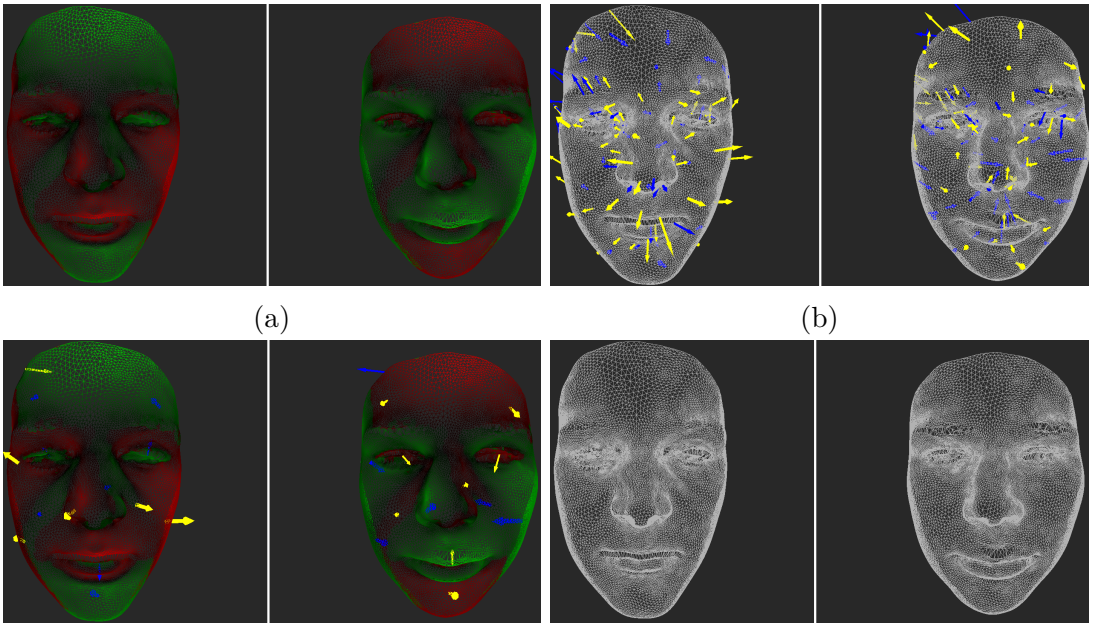
Expected answer: *Down*

Collected answers:

Visualization	A.10a	A.10b	A.10c	A.10d
$\widehat{t}(p, q_4, V_k)$	48.19	53.41	47.80	55.13
Answers				
Down	5	2	2	1
Right	1	0	0	1
In	2	2	1	0
NotSure	2	1	2	2
Left	1	1	0	0
Out	2	5	0	1
Up	0	0	2	1
Total	**13**	**11**	**7**	**6**

A.5.5 Question 5

Does the left cheek stick out more to the front in the right face than in the left face?



(a) (b) (c) (d)

Figure A.11: Visualizations shown for question 5

Expected best visualization: A.11c

Expected answer: *No*

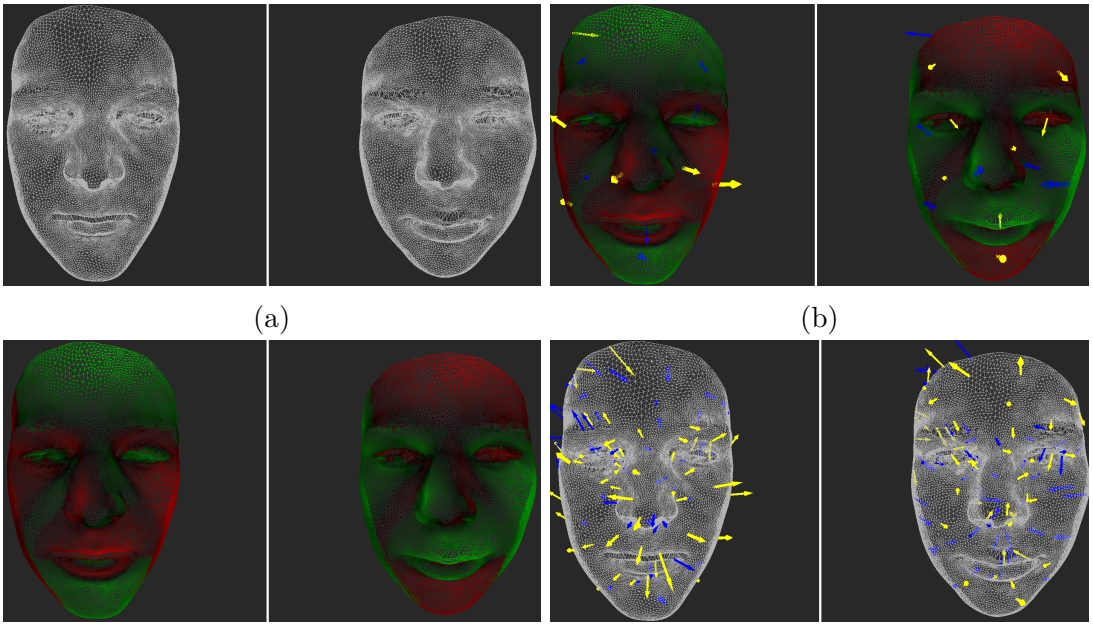
Collected answers:

Visualization	A.11a	A.11b	A.11c	A.11d
$\widehat{t}(p, q_5, V_k)$	40.74	43.85	49.87	31.33
Answers				
Yes	8	5	6	3
No	5	6	1	3
Total	**13**	**11**	**7**	**6**

Commentary: We have based our expected answer on the thin green stripe on the left face in color visualizations. The opposing answers might have stemmed from the fact that "left" and "right" were confused again or that a larger area was understood as a cheek and the difference was interpreted more globally.

A.5.6 Question 6

Which face has a longer nose?



(a) (b) (c) (d)

Figure A.12: Visualizations shown for question 6

Expected best visualization: A.12b

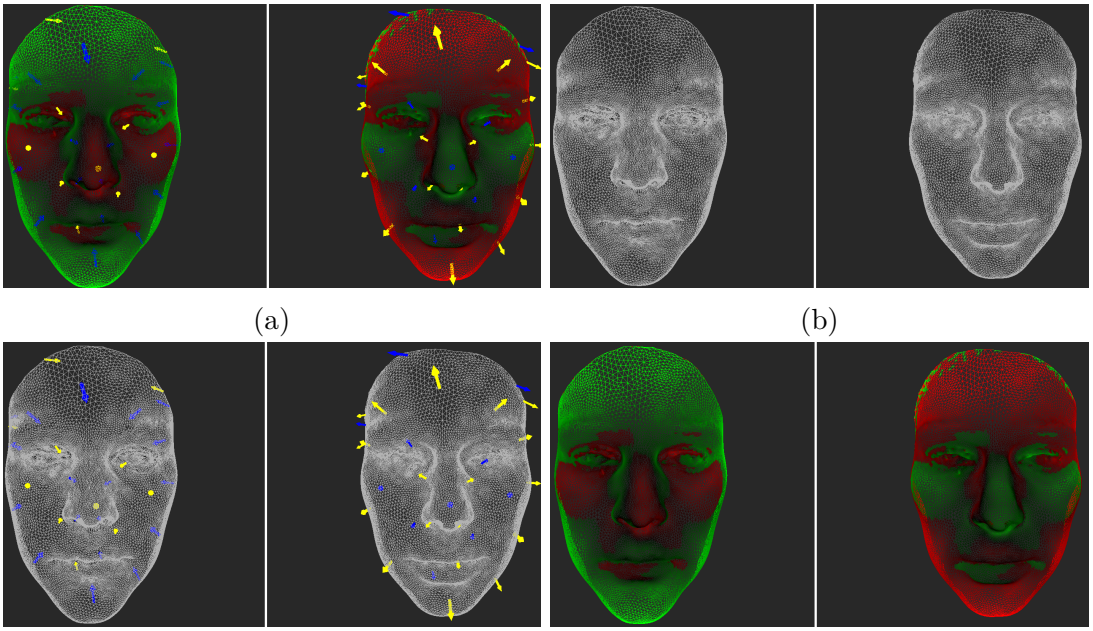
Expected answer: *Left*

Collected answers:

Visualization	A.12a	A.12b	A.12c	A.12d
$\widehat{t}(p, q_6, V_k)$	33.37	32.34	23.85	29.71
Answers				
NotSure	2	1	0	0
Right	3	5	3	1
Left	8	5	4	5
Total	**13**	**11**	**7**	**6**

A.5.7 Question 7

Which face has larger cheekbones?



(a) (b)

(c) (d)

Figure A.13: Visualizations shown for question 7

Expected best visualization: A.13a

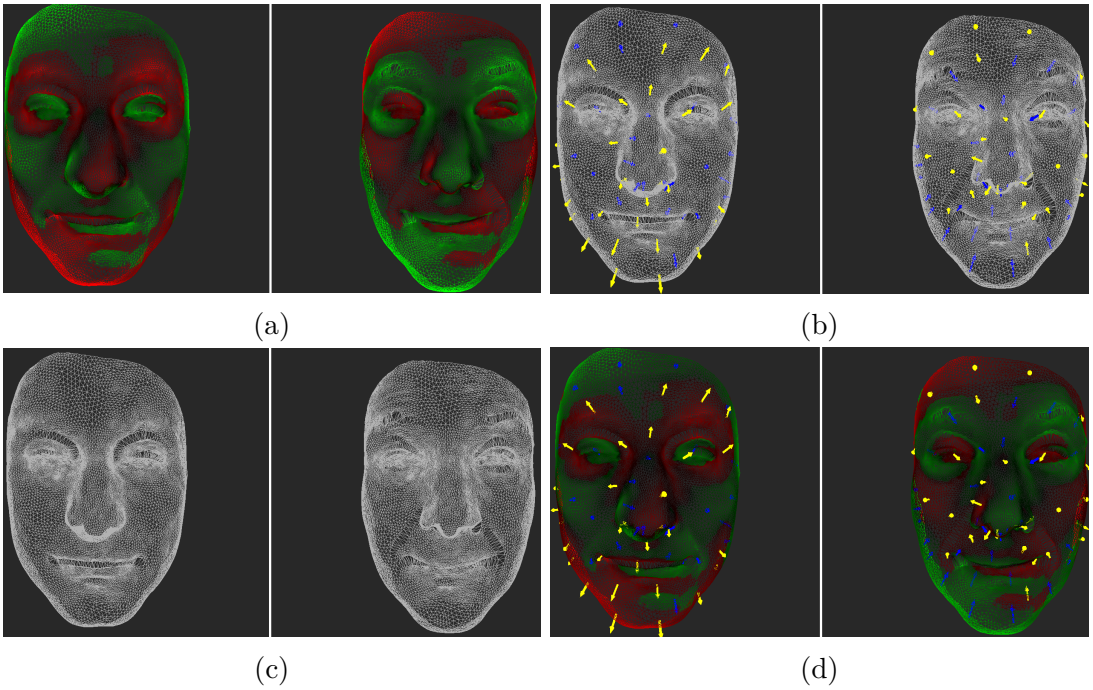
Expected answer: *Right*

Collected answers:

Visualization	A.13a	A.13b	A.13c	A.13d
$\widehat{t}(p, q_7, V_k)$	16.11	24.59	18.47	15.01
Answers				
Right	11	4	6	5
Left	2	4	0	1
NotSure	0	3	1	0
Total	**13**	**11**	**7**	**6**

A.5.8 Question 8

Which face has a larger eyebrow bone?



(a) (b) (c) (d)

Figure A.14: Visualizations shown for question 8

Expected best visualization: A.14d

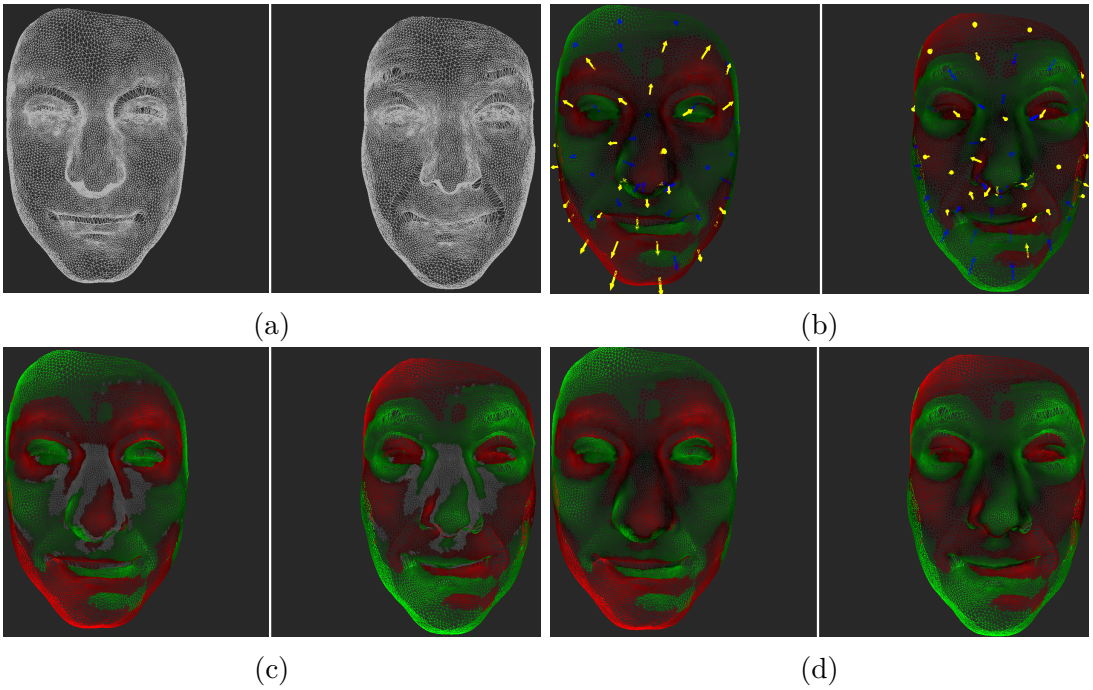
Expected answer: *Right*

Collected answers:

Visualization	A.14a	A.14b	A.14c	A.14d
$\widehat{t}(p, q_8, V_k)$	17.00	16.25	24.09	16.03
Answers				
Right	12	8	4	5
NotSure	1	0	0	0
Left	0	3	3	1
Total	**13**	**11**	**7**	**6**

A.5.9 Question 9

Where are the two faces the most similar?



(a) (b)

(c) (d)

Figure A.15: Visualizations shown for question 9

Expected best visualization: A.15c

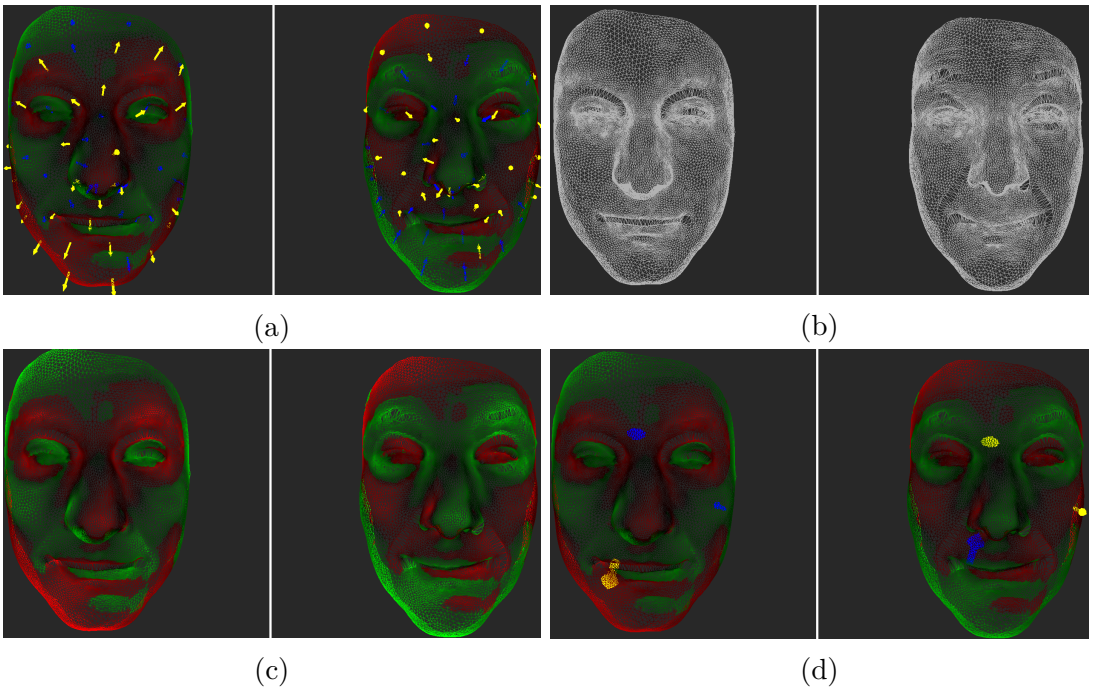
Expected answer: *Nose*

Collected answers:

Visualization	A.15a	A.15b	A.15c	A.15d
$\widehat{t}(p, q_9, V_k)$	47.23	51.02	36.77	42.21
Answers				
NotSure	2	1	1	2
LeftCheek	2	0	0	0
Chin	6	0	1	0
RightCheek	3	2	1	2
Mouth	0	2	1	0
Nose	0	5	3	1
Forehead	0	1	0	1
Total	**13**	**11**	**7**	**6**

A.5.10 Question 10

When examining the differences moving from the left face to the right face, what is their main direction overall?



(a) (b) (c) (d)

Figure A.16: Visualizations shown for question 10

Expected best visualization: A.16d

Expected answer: *Down*

Collected answers:

Visualization	A.16a	A.16b	A.16c	A.16d
$\widehat{t}(p, q_1 0, V_k)$	50.42	33.90	48.06	53.28
Answers				
NotSure	4	2	2	1
Down	3	1	1	2
In	2	1	1	0
Out	3	4	2	0
Up	1	2	0	2
Left	0	1	0	0
Right	0	0	1	1
Total	**13**	**11**	**7**	**6**