FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Parameter Settings Exploration in Visualisation by Using a Semi-automatic Process

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Visual Computing

eingereicht von

Manuel Hochmayr
Matrikelnummer 0627715

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Ing (Master Degree) Peter Mindek

Wien, 01.05.2015

(Unterschrift Manuel Hochmayr)

(Unterschrift Betreuung)

Parameter Settings Exploration in Visualisation by Using a Semi-automatic Process

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Visual Computing

by

Manuel Hochmayr
Registration Number 0627715

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Ing (Master Degree) Peter Mindek

Vienna, 01.05.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Manuel Hochmayr
Daringergasse 28/2/17 1190 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Manuel Hochmayr)

Abstract

Parameters and the process of setting them play a major role in the world of computer based visualisation, no matter whether it is a visualisation of information or of volume data.
Finding suitable parameter values can take up most of the time in the visualisation process and users have to sensibly adjust a large number of parameters. Finding a useful parameter value distribution for achieving the desired visualisation result can be a cumbersome process which also depends on the user's speed and experience. The purpose of this master's thesis is to find a new and faster way to reach an appropriate parameter value distribution resulting in the desired visualisation.
For this master's thesis a prototype is developed which guides the user through a semi-automatic process of adjusting parameter values, which finally results in the desired visualisation of a scientific volume. Using this prototype enables the users to explore a large number of different parameter values within only a few iterations steps and a short amount of time. In order to do so we move away from the classic approach of setting parameters by adjusting sliders or combo boxes.
The idea of this thesis is to combine concepts that were already used in volume visualisation into a prototype. Our main strategy is to present pre-rendered images of the volume with different parameter values to the users. The images that are closest to the target visualisation can be selected and new images, similar to these, are shown. After some iterations of this process the users should have reached a visualisation that meets their expectations. The basis of our approach is a spreadsheet user interface.
Further we make use of the concept of high-level parameters, which are a combination of low-level parameters, like the specular exponent, to one single parameter, like contrast. The advantage of this concept is to have parameters which are more understandable to the users. We move away from the concept of displaying every single image in the spreadsheet interface, having multiple pages. Instead we use kMeans++ or DBScan with an automatic method to choose the distance parameter ε to cluster the images by similarity. This results in only the cluster centres, which are images, being presented to the user in the spreadsheet interface for exploration. Additionally, Locally Linear Embedding (LLE) is used to map single images into a global coordinate system. As a second new approach we use the distance between the images within the coordinate system as a similarity measure for kMeans++ and DBScan. To provide a fast calculation of the Locally Linear Embedding, which includes the nearest neighbours, the distance matrix and the Eigenvalues of the images, we use CUDA. The selection process consists of two different steps: exploration and refinement. Depending on the cluster size of the selected image, a re-clustering of the sub cluster is done if the user has reached the end of the cluster due to having explored all

images and not achieving the desired final image. Thus a new set with varied parameter values is created and used to render new images. In contrast to the initially created set, the newly created one takes into account the explored parameter values from the images chosen by the user. This means that the range - in which the values of the single parameters are varied - is limited by the minimum and maximum value the parameter received during the before made exploration. Our tests showed that that by combining all these techniques it is possible to explore many different parameter values for high-level parameters in a very short time, and to achieve visualisations equal to those created by setting parameter values manually. In a short test our approach enabled two users, who are rather inexperienced in the field of volume visualisation, to create similar visualisations in fewer steps than by setting parameter values manually.

Kurzfassung

Parameter und das Setzen ihrer Werte spielen eine wichtige Rolle in der computerbasierten Visualisierung, egal ob es sich dabei um eine Informations- oder Volums-Visualisierung handelt. Die richtigen Einstellungswerte für die jeweiligen Parameter zu finden, gehört zu den zeitintensivsten Vorgängen während eines Visualisierungsprozesses. Parameter müssen, um das gewünschte Ergebnis zu erhalten, genau adjustiert werden. Diese Aufgabe muss von Benutzern/-Benutzerinnen im Normalfall manuell gelöst werden. Der Prozess, Parameter Werte zu setzen, kann für diese sehr mühsam sein und ist außerdem abhängig von ihrer jeweiligen Erfahrung. Ziel dieser Masterarbeit ist es, einen neuen und schnelleren Ansatz zu finden, der es ermöglicht, die gewünschten Parameter Werte für verschiedenste Visualisierungsalgorithmen zu erreichen, ohne diese einzeln manuell setzen zu müssen.
Für diese Masterarbeit wurde ein Prototyp entwickelt, welcher Benutzer/Benutzerinnen in einem halb-automatischen Prozess beim Setzen von Parameter Werten unterstützt und zum Schluss zur gewünschten Darstellungen eines Volumens führt. Der Prototyp ermöglicht es ihnen, eine Vielzahl an verschiedenen Parameter Werte in wenigen Schritten und innerhalb kürzester Zeit zu erkunden. Um dies zu erreichen sind wir vom klassischen Weg, die Parameter mit Hilfe von Slidern und Comboboxen einzustellen, abgerückt.
Unser Idee ist es, verschiedene Konzepte, welche schon im Bereich der Volumens Visualisierung verwendet wurden, in einem Prototypen zu vereinen. Wir wollen Usern vorberechnete Bilder zeigen, wobei jedes mit anderen Parameter Werten erzeugt worden ist. Das Bild, das den Vorstellungen der Benutzer/Benutzerinnen am ehesten entspricht, kann von ihnen ausgewählt werden und neue Bilder, ähnlich dem Ausgewählten, werden dann wiederum den Usern gezeigt. Nach einigen Iterationen sollten die Benutzer/Benutzerinnen dann ein Bild haben, welches der gewünschten Visualisierung entspricht. Zum Anzeigen der Bilder verwenden wir eine strukturierte Benutzeroberfläche, ähnlich einer Tabelle, ein sogenanntes Spreadsheet User Interface. Weiters benutzen wir noch das Konzept von High-level Parametern. Dabei werden einzelne Parameter, wie zum Beispiel der Einfluss des Glanzpunktes beim Shading, zu einem für die Benutzer/Benutzerinnen verständlichen Parameter, wie etwa Kontrast, kombiniert. Weiters wollen wir vermeiden, jedes einzelne dieser erzeugten Bilder dem Benutzer zu zeigen. Daher verwenden wir entweder kMeans++ oder DBScan mit einer automatischen Methode zum Finden des Distanzparameters ε, um die verschieden Bilder anhand deren Ähnlichkeiten in Cluster zusammenzufassen. Am Ende werden nur die Cluster-Zentren, also die Bilder, die am nächsten zu den errechneten Cluster-Mittelpunkten sind, den Benutzern gezeigt. Zusätzlich verwenden wir Locally Linear Embedding (LLE), um die einzelnen Bilder in ein globales Koordinatensystem abzubilden. Als zweiten neuen Ansatz verwenden wir die Distanz zwischen den Bildern in die-

sem Koordinatensystem als Ähnlichkeitsmaß für kMeans++ oder DBScan. Um eine schnelle Berechnung dieser Koordinaten zu ermöglichen, welche die Berechnung der ähnlichsten Bilder zu einem ausgewählten Bild, eine Distanzmatrix und die Eigenwerte eines Bildes beinhaltet, verwenden wir CUDA. Der Auswahlprozess der Bilder durch den Benutzer lässt sich in zwei verschiedene Schritte unterteilen – Exploration und Verfeinerung. Abhängig von der Größe des ausgewählten Clusters wird dieser entweder neu geclustert oder, wenn zum angezeigten Cluster-Zentrum keine weiteren Bilder mehr gehören weil das Zentrum das letzte Bild in diesem Cluster repräsentiert, wird ein Set mit verschiedenen Parameter Werten erzeugt. Im Unterschied zum vorher erzeugten Set haben die Parameter, deren Werte variiert werden nicht mehr den vollen Variationsbereich. Dieser wird anhand der vorher von den Benutzern/Benutzerinnen ausgesuchten Bilder und deren Parameter Werten eingeschränkt. Unsere Tests zeigten, dass es durch die Kombination all dieser Techniken möglich ist, viele verschiedene Parameter Werte in kürzester Zeit zu erkunden und Bilder zu erzeugen, die denen mit manuell gesetzten Parameter Werten ähnlich sind. In einem kurzen Test war es zwei, im Bereich der Volums-Visualisierung eher unerfahrenen Benutzern mit Hilfe unseres Ansatzes möglich, ähnliche Visualisierungen in weniger Schritten zu erzeugen, als es ihnen mittels manuellen Setzens der Parameter Werte möglich war.

Contents

CHAPTER 1

Introduction

1.1 Motivation and Problem Statement

Visualisation has become a major topic in computer graphics over the past years. There are different types of visualisation, like information visualisation or volume visualisation. All have one thing in common: the need to set parameter values. Such values influence how the final visualisation is produced. However, in many visualisation algorithms there is not only one parameter which value has to be adjust. Users have to sensibly deal with a large number when trying to create the desired visualisation. A typical user could be an illustrator or animator. Within both professions, many different parameters are needed to achieve a result, like a volume rendering which looks like an hand drawn illustration or a fire animation which has realistic behaviour. However, these users are not always familiar with parameter values, names - if these refers to the ones from the implementation - and how the parameters are related to each other, if using a new software or different visualisation algorithm.A user's approach to setting parameter values is often no more than simple trial and error. This means that the user adjusts values randomly and sees what has changed in the image. After the trial and error phase the user knows what the influence of the single parameter to the image is. According to this knowledge the user refines every single parameter value until the imagined result for visualisation is achieved [1]. Therefore, one of the most time consuming and difficult tasks in doing a visualisation is finding suitable parameter values for achieving the desired results. Normally users already have an idea of what the final visualisation should look like but the user interface does not support them properly in choosing the right parameters and their values. Standard user interfaces in volume visualisation software, for example, provide sliders for every single parameter. By using such sliders the user can change the actual value of the parameters and therefore the intensity of their influence to the displayed volume. Sometimes they have a history function in the form of a simple undo button [2]. As already mentioned, this kind of user interface is not easy to work with for non-expert users and does not provide any further functionality for supporting users in finding suitable parameter values and combinations to achieve the desired final image. Another problem is that there is no support from the interface for the user when navigating through the different stages of adjusting the parameters, which lowers the speed of reaching the desired result.

1.2 Aim

The aim of this thesis is to explore how an image-centric method can be utilized for efficient specification of parameter values for visualization algorithms. This means that the parameter values should not be set manually by the user in traditional ways, such as by simply adjusting sliders, combo boxes or spin boxes. Instead the process of setting parameter values should be simplified by avoiding these typical control elements. Additionally parameters are often named by their technical identifiers which are not so easy for users to understand. We want to find a way to make parameters more understandable to a non-expert user. Finally, setting the parameters' values, if the user is a non-expert or if many values have to be set, takes a lot of time. By taking this into account, we want a method where we present the user many different settings for parameter values in a short time to assist him/her to find the desired values for all the parameters. In order to demonstrate the feasibility of our new approach, a prototype of this method should be implemented in volume visualisation software.

1.3 Contribution

Our thesis has four contributions:

1. The main contribution of this thesis is a new way to set different parameter values in a visualisation process. This is done by combining the concept of high-level parameters [3] with kMeans++ [4] or DBScan [5] clustering, where each of the clustering algorithms uses Locally Linear Embedding [6, 7], short *LLE*, coordinates of pre-rendered images. Each of the latter should represent a different setting of parameter values. The *LLE* coordinates are then taken as input to calculate the Euclidean distance to determine similarity between them.

2. In order to accelerate the calculation of the coordinates and of kMeans++ clustering, we implement parts of the algorithms in CUDA to make use of parallelism.

3. DBScan also has one parameter which normally has to be set by the user. This parameter determines the distance between images in one cluster. We automatised the concept proposed by Ester et al. [5] for the setting of this parameter.

1.4 Overview

In chapter 2 an overview of the related work is given. It reviews the work done so far in user interface design and parameter space exploration for visualisation, and provides background information on high-level parameters, the clustering algorithms kMeans, kMeans++ and DBScan, and Locally Linear Embedding. Finally, background information on displaying a large number of images through design galleries or grid like interfaces is given. In chapter 3 the design of the prototype is described. It contains detailed information about the pipeline used to generate images, calculate their Locally Linear Embedded coordinates, and cluster, display and explore them. Chapter 4 gives an overview of the implementation of the prototype in VolumeShop. It describes the technology used and explains the implementation of the pipeline as editor and render plug-in in VolumeShop [8, 9]. The results are discussed in chapter 5. First the test methodology and the volumes are described before talking about the different tests and their results. Chapter 6 concludes the thesis and provides an outlook to the future.

CHAPTER 2

Related work

In this chapter the techniques which are relevant for the thesis are explained. First we start with a short introduction on what has been done so far for parameter space exploration and what techniques are already used. Second we continue with how parameters can be combined followed by the third part of this chapter which focuses on the transfer-function for volume-rendering. Finally we focus on clustering and similarity.

2.1 Visualisation and Setting Parameters

Simply rendering a volume is not sufficient when doing a visualisation. Only after the parameter values are set for the visualisation the needed data can be displayed in a sensible way [2]. The process of finding parameter values is, as Van Wijk et al. wrote, a time consuming one that requires expertise and effort [10, 11]. The traditional approach of setting parameter values in most of today's graphical user interfaces is to use buttons, sliders and combo-boxes [2]. Users with no expertise in setting parameter values for a visualisation experience difficulties in managing their complexity [12]. We want our approach, based on this knowledge, to avoid having common GUI elements, like sliders or combo-boxes, when it comes to changing the parameter values.
Parameters for an algorithm are set, in most cases, heuristically and used for everything that has to be approximated or has a condition; therefore, for all that is uncertain [13].The questions when changing parameters are as follows: How does the visualisation change? How stable is the parameter? In which range is it kept stable [14]? Further there is the already mentioned uncertainty, meaning the lack of information. The IEEE VisWeek 2012 Tutorial on Uncertainty and Parameter Space Analysis in Visualization dealt with these topics. There idea was to combine uncertainty in visualisation and parameter space analysis, which they thought could be essential for future algorithms. Uncertainty in visualisation occurs when different methods of data processing, rendering algorithms and filtering are applied to a data set, for example, to a volume [15]. Hege [16] suggested methods like fuzzy sets or interval analysis to quantify the uncertainty. When doing a visualisation today, the approach of expert users is algorithm-centric

as opposed to domain experts whose approach is data-centric.This means the mapping from data to the final image is not specified by the expert; he/she only defines the features he/she wants to see. Therefore the task of future software would be to have an optimisation process to find the algorithms and their parameter values, the latter of which do the mapping automatically for the user [13].
By avoiding common GUI elements, as described before, we want to reduce the uncertainty for the user when setting parameters by assisting him/her when exploring different parameter settings. Before we can do this we need knowledge of the different ways parameters can be set and how they are related.

2.2 Different Ways of Setting Parameter Values

Design Galleries were introduced by Marks et al. [17]. The purpose of using Design Galleries is to help the user in finding suitable parameter values. The user has a grid like interface from which he/she can select images; images which were generated by varying a list of parameters. Marks et al. demonstrated the use of Design Galleries for setting parameter values for lightening in image rendering, setting the colour in a transfer-function of volume rendering, animation in a particle system and for articulated-figure animation.
Grids are a good way of presenting data to users because of psychological factors [18]. Jankun-Kelly and Ma used this cognisance of grids for their spreadsheet like interface. A two dimensional interface, where each dimension represents a different parameter having varying values, is displayed for exploring the data parameter space [19]. Figure 2.1 shows the spreadsheet interface used by Jankun-Kelly and Ma for determining the opacity and colour of a volume rendering.

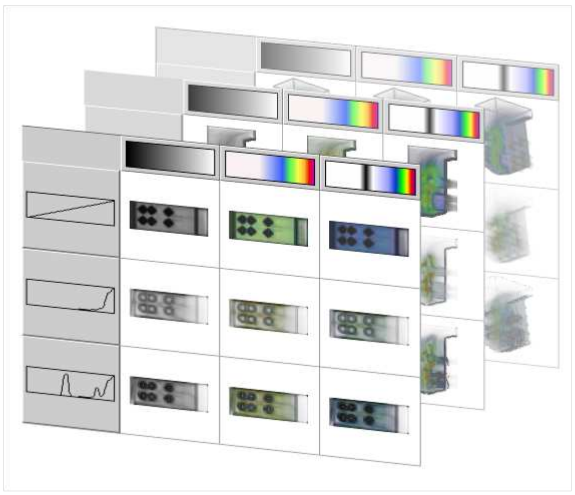
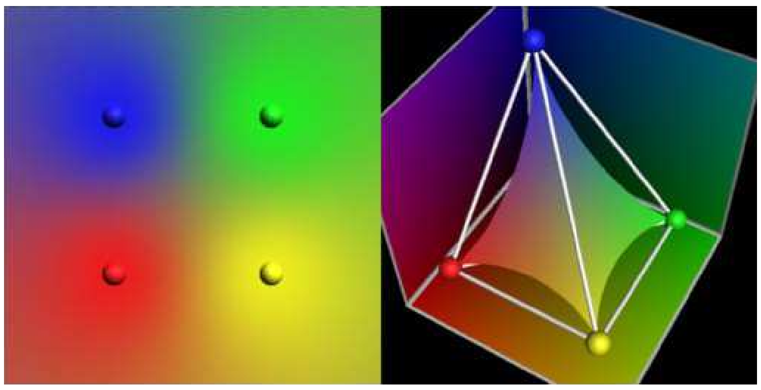


Figure 2.1: Spreadsheet interface where the Y axis represents different opacities and the X axis different colour combinations for a volume rendering [19].

Pre-sets are a common method for supporting novice users in finding suitable parameter values quickly or for providing an initial setting for experts. Van Wijk et al. [10] presented a pre-set based method for setting parameter values. Their approach is to define a setting as a weighted sum of several pre-sets. However their method again had only one parameter, which is the weight of the pre-set. In order to change the parameters, they introduced a pre-set controller. Each pre-set and the actual selected one is represented as a labelled symbol in a plane. Now the user can change the weight by simply clicking anywhere in this plane. Depending on the selected point's distance to the pre-set symbols, the new weight is determined. As an extension for the controller, they added a continuous function in parameter space to find a range of parameter values [10]. Figure 2.2 shows the pre-set controller and what happens when moved by a user.


(a) The pre-set controller for colour in 2D and 3D space. The user can move the shading colour dots in order to change the influence of the pre-set [10].

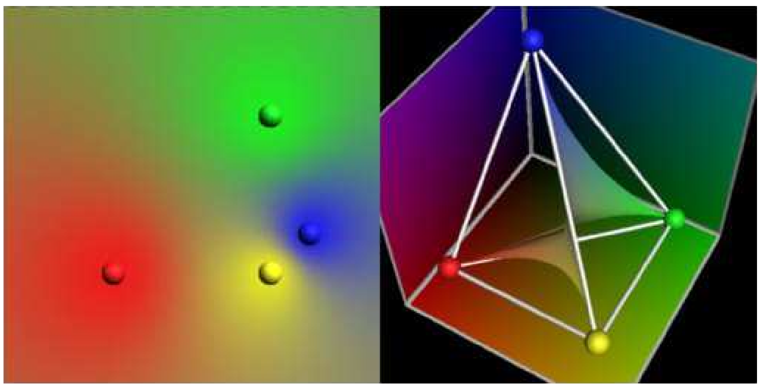

(b) Changing the pre-set by using the controller [10].

Figure 2.2: Pre-set controller and its' use

Another approach for finding parameter values and settings for physically-based simulations was introduced by Bruckner and Möller [11]. They used a result driven visual approach to navigate through the parameter space. At the beginning, a subset of parameter space, selected by the user, is randomly sampled and produces a set of parameter vectors. For every combination, a simulation consisting of multiple time steps on a volumetric grid is made. To determine similar states in the different simulations they evaluate the spatio-temporal similarity. Clustering is then used to group segments. Finally an easy to understand user-interface allows exploration of the different parameter settings. Figure 2.3 shows the method proposed by Bruckner and Möller.
Our approach is based on the concept of design galleries by using the spreadsheet or grid interface to display images. We do not use the idea of having two dimensions where each dimension represents a different parameter, as Jankun-Kelly and Ma [19] did for their grid-Like interface approach. We adapted the approach of producing a set of parameter vectors from Bruckner and Möller [11], to only generate different random parameter values which are then used to quickly generate images.

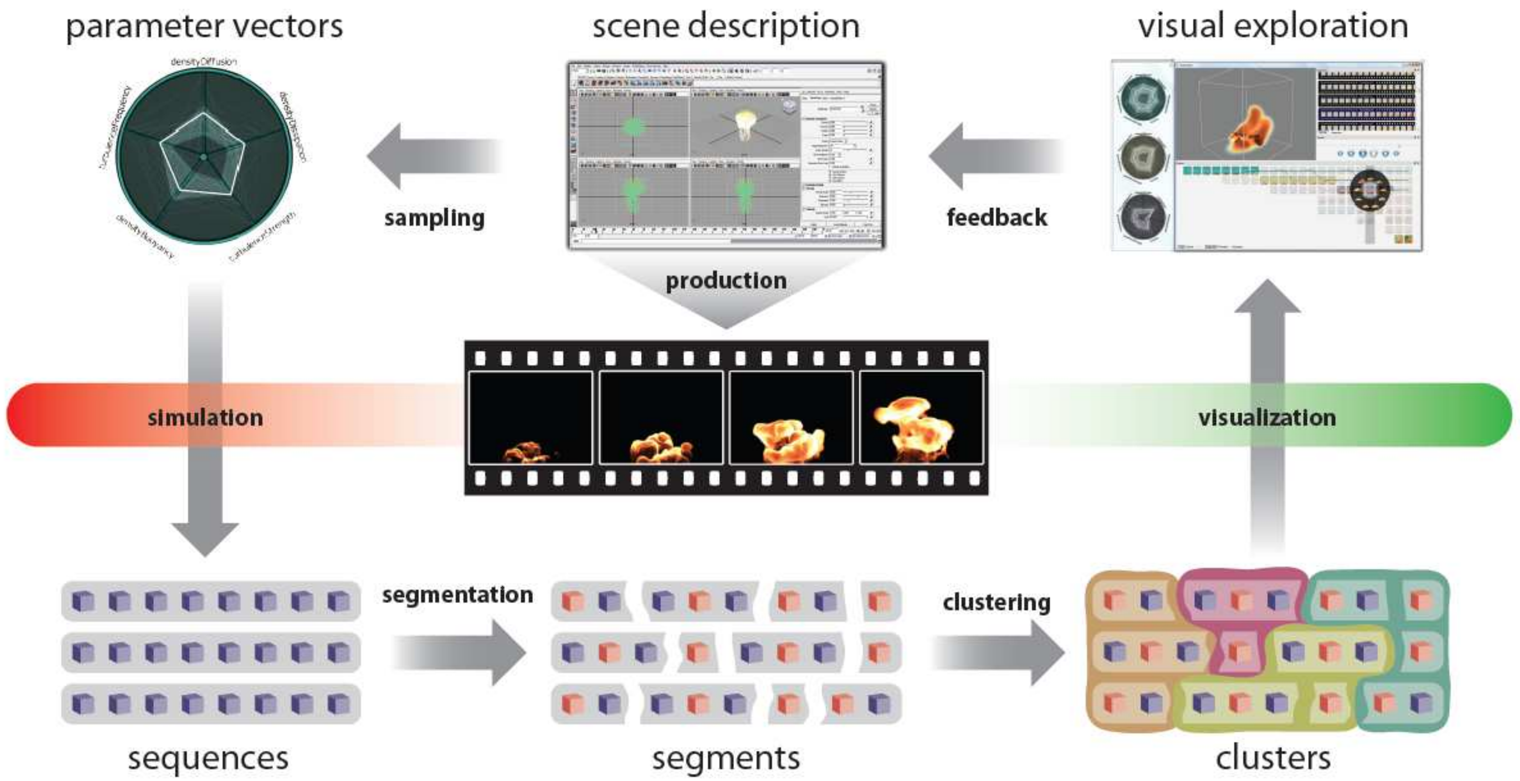
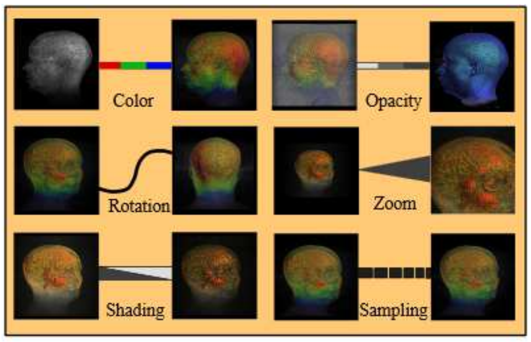


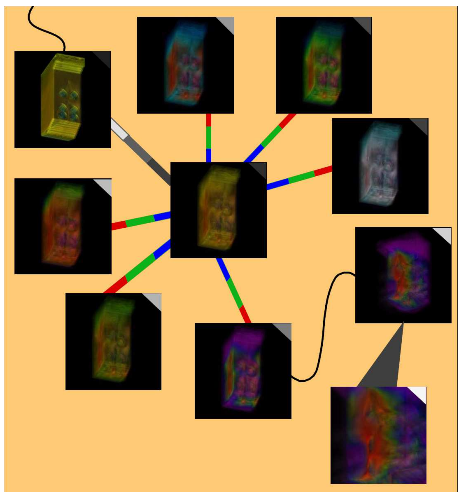
Figure 2.3: The process of finding a parameter values and settings using the approach proposed by Bruckner and Möller [11].

2.3 Combining Parameters

To gain knowledge about the relationship of the rendering parameters and the final image, Ma introduced the concept of image graphs [20]. The idea behind an image graph is not only to represent the results but to also visualise the process of achieving them. A node in the graph represents an image in combination with the parameters used to create it. An edge can have six different shapes, as can be seen in figure 2.4a. It connects two nodes and shows the rendering parameters which were changed between them. Figure 2.4b shows an example of an image graph where the exploration of a dataset from the combustion engineering simulation of an industrial furnace is shown. First the user tried different colour transfer functions on the dataset, represented as coloured edges, before rotating and zooming to get the final visualisation.



(a) The different edges where each represents the change of a different rendering parameter between two nodes [20].



(b) Image graph of the exploration of a dataset from a combustion engineering simulation of an industrial furnace [20].

Figure 2.4: The image graph proposed by Ma [20].

Bhagavatula et al. [3] introduced a concept for summarising parameters for volume rendering algorithms into one single parameter which they call high-level parameter. Normally a user has to set many different parameters manually and one after another, like the amount of ambient light, the amount of directed light or the scale of gradient enhancement. Bhagavatula et al. observed an intuitive relationship between certain low-level parameters when changing their values. For example, when adjusting the four different low-level parameters, which are influencing the contribution of non-boundary regions, the silhouette for contribution of non-silhouette regions, the scale of the gradient enhancement and the scaling in silhouette regions, the volumes sketchiness was increasing or decreasing.In order to find relationships between low-level parameters and to define high-level ones, expert knowledge was needed. Bhagavatula et al. defined different high-level parameters by trial and error and then conducted a user study to determine whether these parameters were influencing the volume the way they were supposed to. After having defined the high-level parameters, a mapping between them and those of the low-level was needed. They used a decision tree to map the value of the high-level parameter to the single low-level parameters. This tree also captured the relationships between the two types of parameters. To allow the user to change the influence of a high-level parameter a simple user interface was provided where each high-level parameter was represented by a single slider to increase or decrease its influence. Every time the user changed a high-level parameter, the volume was rendered and displayed. The result of each rendering is used as reference input for the next high-level parameter.
For our approach we use the concept from Bhagavatula et al. [3] which combines different low-level parameters to one single high-level parameter because they are more understandable. They are named by properties of the presented image which can be easier understood by the user.

2.4 The Transfer-Function - Parameters and Automatic Setting

There are not only low-level parameters which can influence a visualisation. A special parameter in a volume rendering is the transfer-function. An example can be seen in figure 2.5. Volume data consists of voxels, where each voxel is composed of one or more values. A voxel is a spatial location where a density value of the scanned or manually created material is given. To map density values to optical properties, like opacity and colour, a transfer-function is used. A way of changing a transfer-function is to have a two dimensional coordinate system. The X axis of the coordinate system corresponds to the density values, and the Y axis determines their opacity. The user sets control points, which assign an opacity and RGB value to a density value. Those points are then connected via a curve, representing the alpha gradient for the density values in between. Along the X axis a colour range, defined by RGB values, can also be set. Typical examples for 2D transfer-function are the trapezoids and the paraboloid shapes [12, 21, 22].

Editing this function is a difficult task for the user because it maps the data values to different optical domains, like data range, opacity and colour. König et al. presented a new user interface to specify a transfer-function. It provides tools for each of the before named domains which enables the user, depending on his/her experience, to define the transfer-function [23]. First the data range is defined. For the entire range of the data values, thumbnail images are

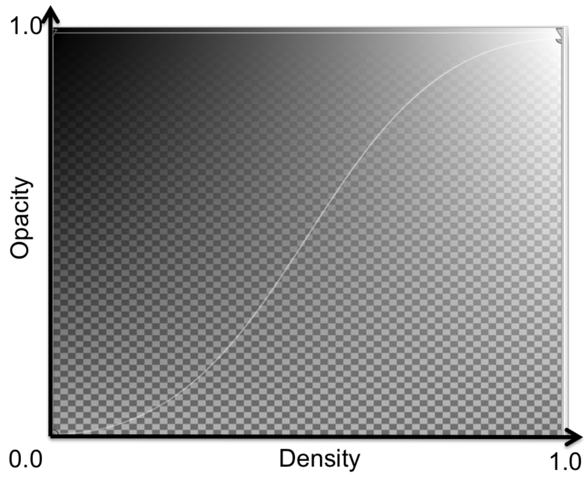


Figure 2.5: Example of a transfer-function for a volume rendering. The X axis represents the dense values. The Y axis determines the opacity or alpha level and the influence of the colour of the dense values.

rendered and displayed to the user. If the user is interested in a certain range and wants to see additional similar thumbnails around this range, he/she can zoom in.To find certain materials more easily, a histogram of the data values is displayed under the thumbnails, where peaks give hints of the material. Second the colour of the transfer-function is determined by assigning it to each selected peak from the step before, one after another. The hue and saturation of possible colours is represented in a bar. The third and final step is to determine the opacity range of the transfer-function. The before assigned colours to the different peaks are combined to a single visualisation, where each single peak receives an opacity value. This is done by using a simple user interface. Figure 2.6 shows the approach of König et al. [23] The upper part (1a. in the image) represents the first step of the approach, flowed by the second step which is the choosing of colour (2a. in the image), and the third step which is the setting of the opacity (3a. in the image). To have a larger view of the selected thumbnail and achieve real time feedback, an additional window within which the volume is rendered by using direct volume rendering, is provided.

Bruckner and Kühschelm [24] developed a framework called *The Gallery* to set a transfer-function for direct volume rendering. This approach was a image centric one in that it generated galleries for different datasets. They defined a collection of transfer-functions, like box, tent and ramp; all three of which are a common pattern when designing a transfer-function. One of the selected patterns is shifted along the data axis to produce different images, which are displayed in the gallery. After the images are generated, the task of the user is to define the transfer-function in three steps. The first step is to select images out of the generated galleries. These images represent the different opacity transfer-functions. The second step is to define the colour which is then used for the opacity functions to do a final selection of images. Using this strategy Bruckner and Kühschelm wanted to handle complex settings, for example, for multiple transfer-functions.

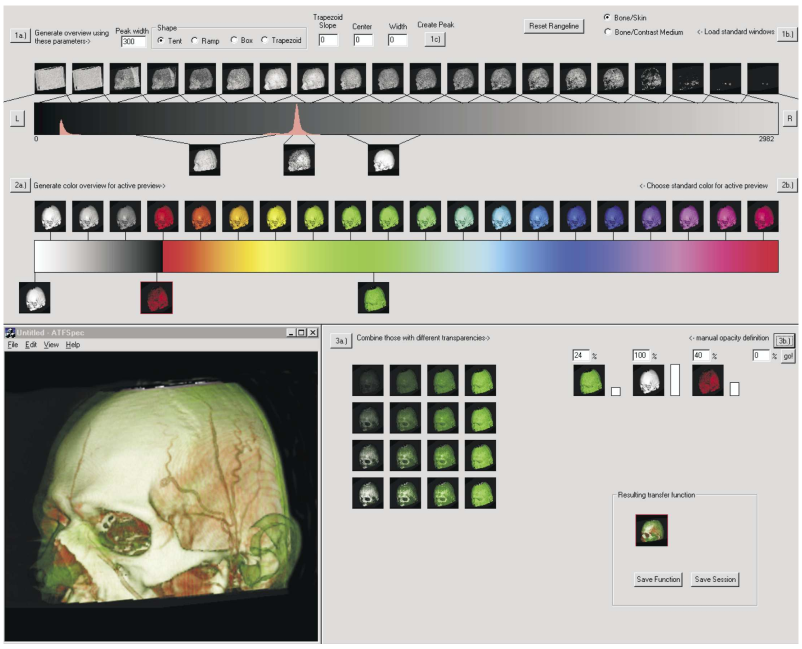
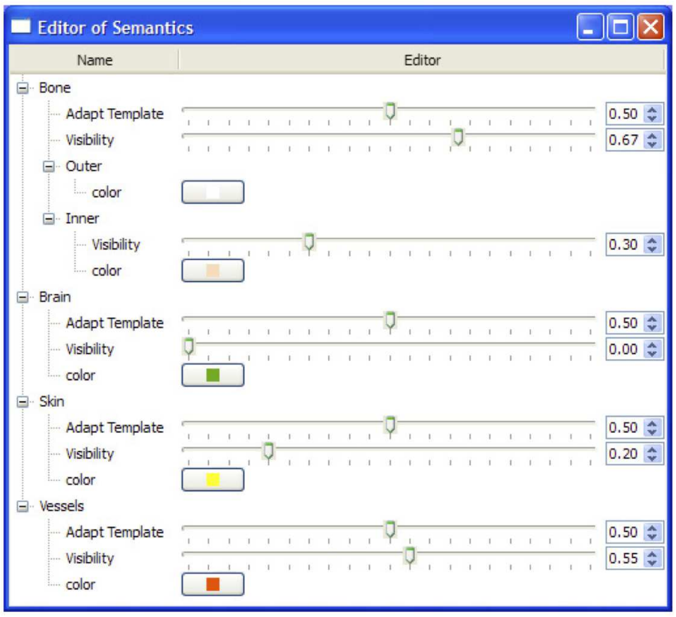


Figure 2.6: The user interface for mastering a transfer-function proposed by König et al. [23].

Salama et al. [12] presented a framework which allows non-expert users to select a high-level transfer-function priorly defined by an expert. Their idea was to take a concept from computer animation: the combining of parameters, called low-level parameters, to one simple high-level parameter and therefore hiding the complexity of parameter assignment from non-expert users.The designed semantic transfer-function models by regarding each transfer-function as an array of *n* floating point values where the array is influenced by pre-defined semantic parameters. There aim was to create different semantic models applicable to different data sets. In creation of the models, the intention was for the process to be done in cooperation between a computer scientist and a domain expert, like a medical doctor, for those data sets coming from a CT or MRT.The first step is to find the required semantic parameters and their respective weighting. They suggested using reference data which *"should statistically represent the range of possible data sets for the desired examination purpose"* [12]. The next step is to split the relevant structures of a data set into single components - for example, splitting the human body into bones, skin and blood vessels - to create a transfer-function model. The final step is to apply the created model to each of the data sets and to adapt it accordingly to fulfill the requirements of what should be displayed. The final result is a parameter array for each reference data set where the principal component analysis (PCA) was also applied to create the final semantic parameters. The non-expert user changes the influence of each of the semantic parameters by sliders. An example of the user interface for an angiography CT data set can be seen in figure 2.7. The number next to the slider represents the actual value of the semantic parameter. By conducting a user study they were able to demonstrate that semantic models can be used to make a transfer-function more comprehensible to a non-expert user and that the complexity of the visual parameter assignment can be hidden.



Name	Editor
Bone	
Adapt Template	0.50
Visibility	0.67
Outer	
color	⬜
Inner	
Visibility	0.30
color	🟧
Brain	
Adapt Template	0.50
Visibility	0.00
color	🟩
Skin	
Adapt Template	0.50
Visibility	0.20
color	🟨
Vessels	
Adapt Template	0.50
Visibility	0.55
color	🟥

Figure 2.7: Graphical user interface to change the influence of the semantic parameters to the data set [12].

Finally Opitz [25] presented a method to support the design of a transfer-function by using mean shift clustering and LH histograms. First he generated a LH histogram of the data to which then applied the mean shift clustering in order to find a centre for each bin which is required to build the histogram. In a third step he constructed the transfer-function in the following way: By clicking the point he/she is interested in within the volume, the corresponding cluster to this point is automatically selected by using the LH histogram.For each point assigned to the selected cluster their corresponding position in the transfer-function is highlighted by the cluster specific colour. Figure 2.8 shows an example of the LH histogram and the corresponding colouring of the Chapel Hill CT head data set on the right side. Opitz showed with his thesis that LH histograms in combination with mean shift clustering are capable of automatising and making the creation of transfer-function easier.
As our approach is implemented for a volume rendering software we also wanted to simplify the use of the transfer-function. We regard the transfer-function as an array of parameters in the same way as Salama et al. [12]. In two steps we want the user to set the transfer-function and then the colour. The different variations of the transfer-function are once more presented to the user as generated images in the grid interface. Again this is described in more detail in the design section.

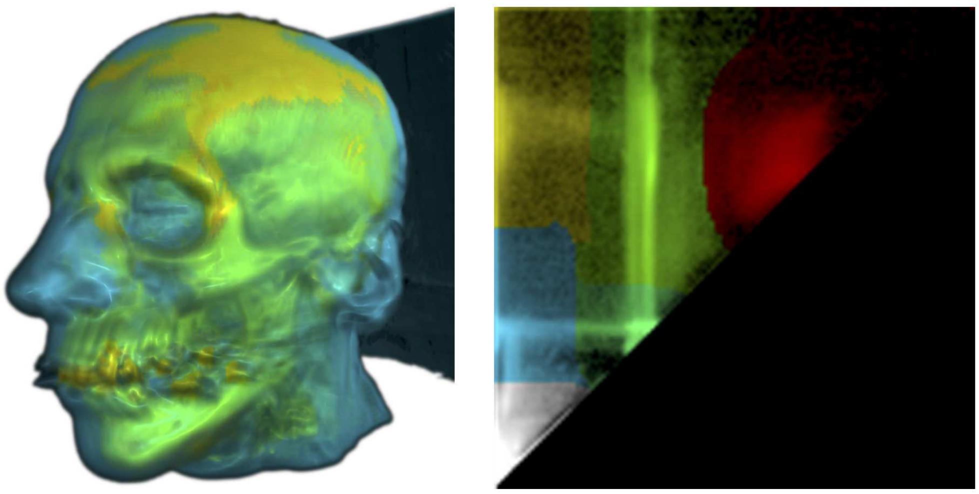


Figure 2.8: The final colouring of the clusters applied to the volume using the method of Opitz [25].

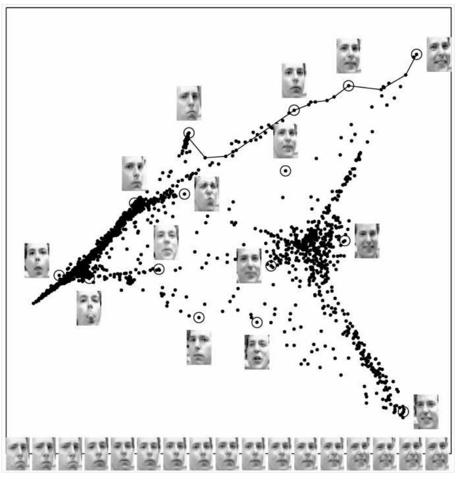
2.5 Clustering and Similarity

Clustering is a common method used to unite similar results, especially images. According to a survey made by Berkhin [26] in 2002 *"the kMeans clustering algorithm is by far the most popular clustering tool used in scientific and industrial applications"*. Even if it is one of the most popular, it has a disadvantage: the number of cluster centres, *k*, has to be determined first before running the algorithm. The quality of the result of the clustering depends on how appropriate the chosen *k* is. If *k* is too big or too small, the algorithm may not perform well, which results in wrong cluster assignments. Arthur and Vassilvitskii [4] improved the kMeans clustering algorithm by adding a method for better distribution of the *k* start centres. They referred to this method as kMeans++ clustering. By applying this improvement kMeans++ performs two times better than kMeans [4].

Another approach was introduced by Ester et al. [5]. They presented DBScan, i.e., Density Based Clustering, as a new algorithm for clustering data that requires two input parameters.The advantage of using DBScan is that it can discover clusters of arbitrary shape in contrast to kMeans++ which produces only squared shaped clusters.
In order to determine clusters, both algorithms need a measurement for similarity of images. Therefore an important question is how to determine similarity by use of a machine. In a visualisation process one often has multidimensional data which has to be compared based on similarity [27]. Ankerst et al. [27] proposed a method to effectively order the dimension of the multidimensional data by its dimension similarity. For measurement they used the Euclidean distance and a Fourier-based method.

According to Seung and Lee [28] the brain distinguishes between different images by deter-

mining their manifolds and comparing them. However, it is not yet known how these manifolds are represented in the brain. The method of Locally Linear Embedding - short *LLE* - is an algorithm to map high dimensional data into a low dimensional single global coordinate system. The algorithm was introduced by Roweis and Saul [6, 7]. One of the advantages of *LLE* is that it is capable of learning the global structure of the underlying non-linear manifolds generated by images. Furthermore *LLE* coordinates are invariant to rotation, rescaling and translations [6, 7]. Figure 2.9 shows two examples of a set of images represented as *LLE* coordinates and their distribution in a two dimensional coordinate system.
In order to not glut the user with many different images and to make exploration of a big amount of images within a short time possible, we make use of kMeans++ or DBScan clustering algorithm. Both algorithms use *LLE* coordinates of generated images as input to calculate the Euclidean distance. We do this to see whether *LLE* in combination with the two clustering algorithms is working properly.


(a) Faces which are mapped to 2D *LLE* space [6].

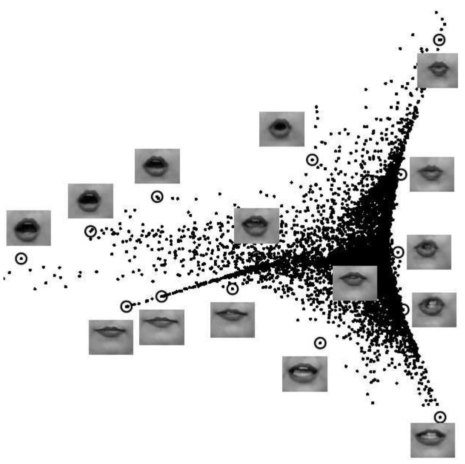

(b) Changed the pre-set by using the controller [10].

Figure 2.9: Examples of mapping different images to *LLE* space. Representative images are shown next to the circled data points.

CHAPTER 3

Design of the Parameter Selection Advisor - ParSAd

In this section, the individual processing steps which form the basis of our approach are described. We start with the general description of our approach and give a figure of the pipeline. Based on the pipeline we describe the different stages we run through to find the final parameter values.

3.1 Approach

With our approach we want to achieve that a user is able to find suitable parameter values for many different parameters of a visualisation algorithm without having to set them manually with common GUI elements. Our idea is to automatically generate a large number of images with the algorithm and slightly different parameter values. This set of images is presented to the user who then selects those images closest to the visualisation he/she imagined. Having a large number different images, we do not want to present all of them to the user at once. The idea is to cluster the images by their similarity and to only display one image out of each cluster. This limits the number of displayed images from which the user can select but finding an image reflecting the final desired solution with the few displayed images becomes very unlikely. Therefore we have to repeat the process and show the user new images similar to the already chosen ones. For this we can take already existing images from the selected clusters or create new ones with parameter values similar to those of the selected images. This is done iteratively until the user is satisfied with the result of the visualisation algorithm.
To realise this approach we want to try a combination of three different techniques. The first is a grid-like interface [18] used as the user interface where the images are displayed. In contrast to the original idea we do not use the two dimensions of, where each dimension represents a different parameter having varying values as already mentioned in the related work section.We simply use it as an easy interface to select images from. The second is clustering. We decided to

use two common algorithms: kMeans++ [4] and DBScan [5]. We chose kMeans++ because it is fast and we wanted to see whether using kMeans++ to cluster an arbitrary amount of images instead of using it for clustering an image itself produces good results. kMeans++ is known for only finding convex clusters [26]. Therefore another clustering algorithm was needed which was also able to find clusters with arbitrary shapes. One of them is DBScan, a density based clustering algorithm. These types of clustering algorithms are flexible in terms of their shape [26]. Both clustering algorithms have the disadvantage of having input parameters which influence their results. For kMeans++ it is k and for DBScan it is the number of minimum points in a cluster and ε, which defines the maximum distance to the other points in the cluster. For both we have a concept that allows the user to no longer have to care about these parameters. k gets a fixed value. For determining ε we try a new automatic approach which is based on the original idea of Ester at al. [5] to set ε manually.This may improve the results of the DBScan algorithm, and the user needs no domain knowledge about the distribution of the full amount of images. It is described in more detail in this chapter. Third, when clustering, we wanted to use different input for calculating the Euclidean distance between the images instead of their RGBA values. Therefore we again tried something new and used *LLE* in combination with the two clustering algorithms.

3.2 Area of application

Our method is designed for users working in the field of visualisation, simulation and/or animation that use different visualisation algorithms where many parameter values have to be set to reach the desired final result. In the field of medical visualisation, for example, illustrators have to deal with many illustrative visualisation algorithms to make data that has been created in a CT or MR visible in a way that a human being can understand. In the field of animation, physic simulations, like fire, fluids, particles, deformation or collisions, are very important to achieve a good looking and realistic result. These simulations mostly use mathematical formulas with different parameters which values can be individually varied.An animator may be an expert in using his/her software but has no knowledge at all about the influence of the different parameters on the formulas.
Therefore we designed our method in a way so that it can be adapted for all mentioned fields. By taking these areas into account, our method is applicable for visualisation algorithms with many different parameters - which influence each other - where fast exploration is required. We do not want to confront the user with any details of parameter settings except the name of the high-level parameters. Instead we intend to provide a simple to use application which still enables the user to achieve their desired result. Furthermore it can be utilised to visualise an initial result very quickly, i.e., within a few steps, without any knowledge of the parameters. To allow for single parameters to still be set, we also offer standard control elements which are per default hidden in order to not overwhelm the user.

3.3 Detailed Concept

To demonstrate that our idea is working we implemented a prototype for a volume visualisation application. The final pipeline for achieving the aim of finding parameter values without setting them manually, as defined in section 3.1, consists of six different steps. A schematic presentation of said pipeline is shown in figure 3.1. The first step is to automatically generate images where each image has different, randomly chosen parameter values. The second step is to map them to 2D coordinates.The closer the coordinates of two images are, the more they are similar to each other. This behaviour is used for the third step, which is to cluster the images based on their similarity to each other. As a fourth step we select one image from each cluster and display them to the user. For this the images closest to the cluster centres are taken. The fifth step requires user input in that the user has to select one or more images which is (are) closest or have anything in common with the visualisation he/she imagined (step 5b), or already represents it (step 5a).If the selected image(s) is (are) not the final one(s), we differ between two cases. The first case (step 6a) is when the displayed image(s) is (are) the only image(s) in the cluster, meaning the cluster consists of one image. Therefore we can not display further images to the user without generating new ones. In contrast to the initial generation, we have to refine the range in which the parameter values are varied in order to get images similar to those selected by the user. The second case is if the selected image(s) belongs to a cluster (belong to clusters) containing similar images. These images are than all taken and clustered again, which takes us back to step 3 in the pipeline. We repeat all the steps until the final visualisation is found by the user. The following sections give detailed explanation for each of the steps in the pipeline.

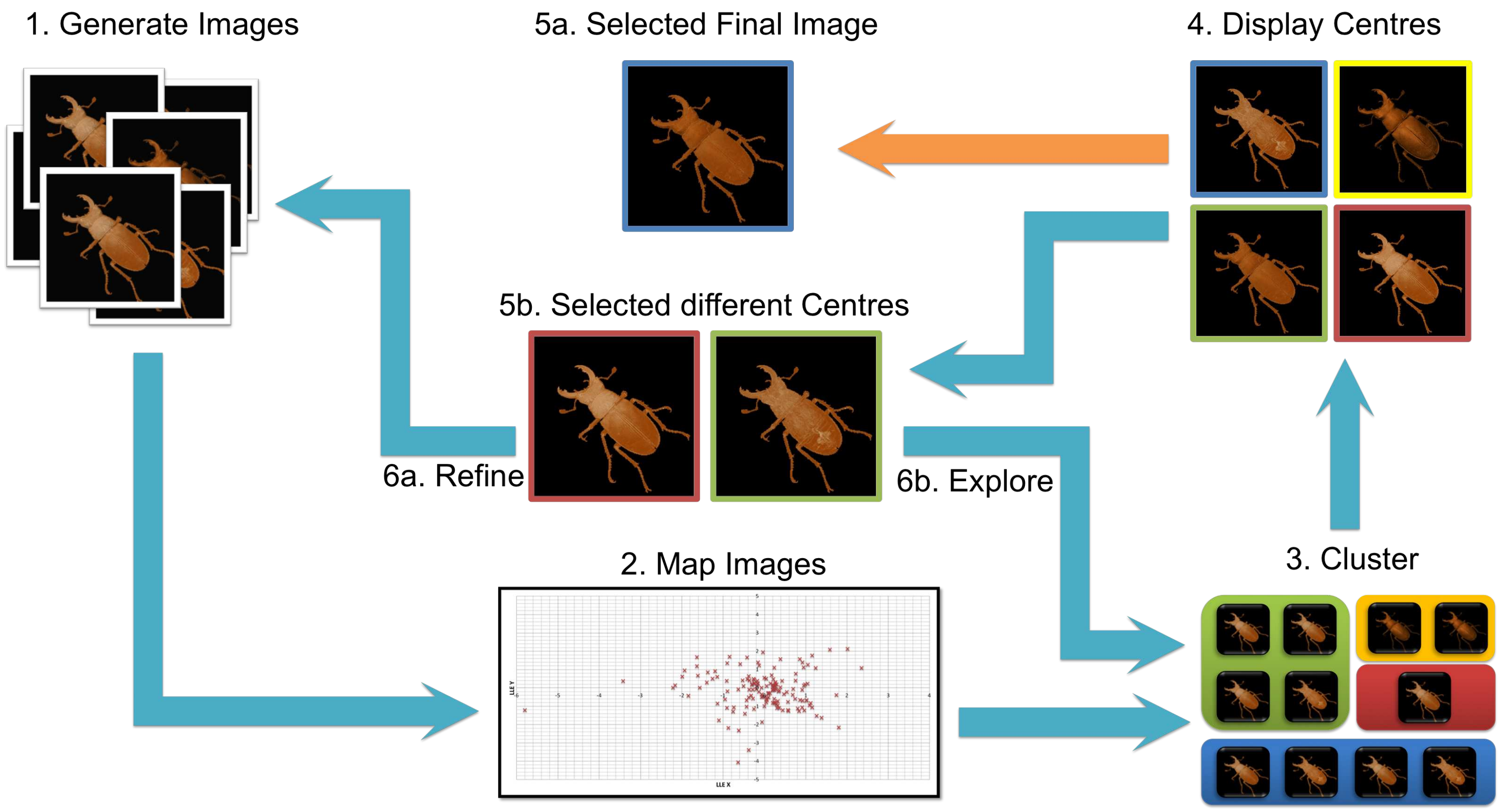


Figure 3.1: The pipeline, consisting of six different steps. The turquoise arrows represent the closed circle. It starts with generating the images followed by mapping them to 2D coordinates. After that we cluster display the images. If the user continues exploring clusters a re-cluster or refine could be necessary. Depending on whether one of the displayed images already show the final visualisation, the user can select this image and the pipeline stops at step 5a and is represented by the orange arrow.

3.4 Image Generation

For our approach we automatically generate different images by randomly varying the values for all parameters. This generation of random parameter values represents the first step in our pipeline and is necessary for creating a set of different images which is later presented to the user. Our intent is to vary the values for every parameter when generating a new image to ensure broad coverage of different parameter values. But sometimes it is not necessary to vary all of them because the user is already satisfied with the visualisation and only wants to change a few details, like the contrast or the level of detail. It seems to be an obvious step to group parameters which influence the visualisation in exactly that way. Therefore for our implementation, we use the concept of high-level parameters, representing a group of parameters with an understandable name, introduced by Bhagavatula et al. [3].

3.4.1 High-Level Parameters

The concept, which was introduced by Bhagavatula et al. [3], groups parameters according to their influence on volume rendering. It combines parameters - called low-level parameters - of a volume rendering algorithm, for example, shading intensity, diffuse shading or specular exponent, to single parameters - called high-level parameters. All the values of the low-level parameters which belong to a high-level one are changed immediately when its influence to rendering is increased or decreased. Bhagavatula et al. used a decision tree to map the value of the high-level to the values of the low-level parameters. Further the combination is not a random one; it depends on how the combined low-level parameters influence the outcome of the volume rendering as described in the previous section on related work, 2.3. The advantage of using this concept is that high-level parameters have easy to understand names, like contrast or opacity. These are terms already know by most users who intend to use our approach.

Our idea is based on this approach. The high-level parameter *All* represents our initial approach to vary the values of many different parameters at once. This allows use of our system without much knowledge of parameters and their values, and a visualisation aim is still able to be reached.
We use modified pre-defined high-level parameters in our implementation to change the visualisations appearance, like opacity or sketchiness. They are based on those by Bhagavatula et al. [3]. Instead of using a decision tree to change each high-level parameter, and in order to make our approach fast and different from the one of Bhagavatula et al., we randomly vary the low-level parameter values as described in section 3.4.3. Table 3.1 shows the list of the low-level parameters used for our approach. The low-level parameters for volume rendering are those proposed by Rheingans et al. [29] and by Bruckner et al. [8]. The low-level parameters proposed by Bruckner et al. [8] influence the volume rendering differently or equally to the ones of Rheingans et al. [29]. Depending on that behaviour, we replaced or added low-level parameters to the proposed high-level parameters by Bhagavatula et al. [3] and looked at whether the high-level parameter had the desired behaviour. After this evaluation we defined our high-level parameters as can be seen in table 3.2.

Low-level parameter	**Influence**	**Type**	**Range**
Sample distance	The step size of the ray used for ray-casting	Continuous	0.5 to 2.0
Level	Specifies with window how the data values are mapped to the output range	Continuous Float	0.0 to 1.0
Window	Specifies with level how the data values are mapped to the output range	Continuous Float	0.0 to 1.0
Shading	Applying shading	Categorical Boolean	True or False
Cell shading	Applying toon or cell shading	Categorical Boolean	True or False
Shading intensity	Intensity of the used shading	Continuous Float	0.0 to 1.0
Diffuse shading	Amount of diffuse shading	Continuous Float	0.0 to 1.0
Specular shading	Amount of specular shading	Continuous Float	0.0 to 1.0
Specular exponent	Shininess of the material	Continuous Float	0.0 to 128.0
Gradient shading	Controls the influence of the gradient magnitude on the shading intensity - lower values will cause homogeneous regions to remain unshaded	Continuous Float	0.0 to 1.0
Gradient opacity	Influence of the gradient magnitude on the sample opacity - lower values will result in transparent homogeneous regions	Continuous Float	0.0 to 1.0
CoNbR	Changing the opacity of regions which are not part of a boundary in the volume	Continuous Float	0.0 to 1.0
Gradient scale	Scaling of the gradient enhancement	Continuous Float	0.0 to 100.0
Fall-off	Tightness of enhancement Fall-off	Continuous Float	0.0 to 10.0
CoNsR	Contribution of non silhouette regions	Continuous Float	0.0 to 1.0
Silhouette scaling	Silhouette scaling	Continuous Float	0.0 to 100.0
Silhoutte Fall-off	Enhancement of the silhouette Fall-off	Continuous Float	0.0 to 10.0

Table 3.1: The different low-level parameters and their effect on the volume rendering.

High-level parameter	**Influence**	**Belonging low-level parameters**
Contrast	Changes the contrast of a volume therefore it can get darker or lighter	Shading, Shading Intensity, Diffuse Shading, Specular Shading, Specular Exponent
Detail	Determines the amount of details that can be seen in the volume.	Shading, Specular Shading, Specular Exponent, Gradient Shading, CoNsR
Opacity	Makes a volume more or less opaque	Shading, Gradient Shading, Gradient Opacity, CoNbR, Gradient scale, Fall-off, CoNsR
Flatness	Makes the volume look more or less three dimensional	Shading, CoNsR, Gradient Shading, Diffuse Shading, Level, Window
Sketchiness	Makes a volume look more or less hand-drawn	CoNbR, CoNsR, Silhouette scaling, Gradient scale, Silhoutte Fall-off, Fall-off
All	Varies the values for all low-level parameters at once.	All low-level parameters from table 3.1
Transfer-function	Varies the alpha value A (opacity) of the RGBA set.	Set of RGBA values assigned to density values

Table 3.2: The different high-level parameters, their influence on the volume rendering and their related low-level parameters. They are based on the approach of Bhagavatula et al. [3] with adaptations for our own approach to include the low-level parameters of Bruckner et al. [8], written in italics in the table.

3.4.2 Transfer-Function for Volume Rendering

The transfer-function consists of a variable number of control points which are normally set by the user. In an attempt to simplify the generation process of the transfer-function we want to use the same approach as with the low-level parameters. We consider every control point as a low-level parameter which opacity value is varied.This is similar to the approach of Salama et al. [12] who regards the transfer-function as an array of *n* floating point values. As an initial transfer-function we use two control points. The first control point has an opacity and density of zero and the second one an opacity and density of one. As initial colour for these two control points we take black for the first point and white for the second. The points are then connected via a sigmoid function curve which is our initial pattern for the transfer-function. Different to the approach of Bruckner and Kühschelm [24], we pre-defined only one pattern. It changes within the exploration of the function. The reason for the two points at the beginning is to keep the number of control points small and to cover a wide range of dense values lying between the two control points. When refining the transfer-function the number of control points can increase and different curve types can occur. This is described in more detail in section 3.8.2.

Colouring the Transfer-Function
Other parameters of the transfer-function are the red, green and blue RGB values which a user can set by clicking on arbitrary points along the upper X axis. This allows the user to choose a colour for the respective values.The RGB values' range is between zero and one or zero and 255, depending on the implementation. The RGB values can be considered as three different parameters with values that are permuted, as is the case with other low-level parameters.However colour is very specific because it is quite sensitive to changes in the RGB values. Even slightly different RGB values can result in very different colours. This leads to a huge number of possible colouring combinations which would take too much time during pre-processing and which are impossible to display.
As we still want to simplify and automatise the setting of parameters, we also tried to find a simple solution to assist the user in the colouring process. We decided to let him/her select the colours himself/herself and to only automatise the assignment of colours to the density values. This makes it possible to avoid presenting the user with images for every possible colour, and also speeds up the process. Next we take the selected colour and generate different images where the colour is used for a certain range of density values. We limit this range by control points which again limit the number of colours a user can choose.Our general approach is to vary the values for different parameter combinations and let the user choose the images which have parts in common with the final visualisation or represent it exactly.This is also applicable for colour. Analogue to the initial approach, we vary the range of the density values to which the user selected colour is applied. The images generated, where each has different colouring, are shown to the user and he/she chooses the image which displays parts of, or represents the final colouring.
An example would be taking a transfer-function with three control points, as can bee seen in figure 3.2. The user chooses a colour, which is step one in the figure. At the beginning we do not know whether the density values in the lower, the middle or in the higher range should have this colour. For the second step we assign the colour to all three control points, which also means that all density values get this colour assigned in combination with the opacity value. The result is one image the user can select. This is because in the first step we do not have a second colour to vary the first with. The user can chose this image and stop the colouring process, replace the actual colour or he/she selects the image and therefore the actual colouring of the control points (step three in figure 3.2 indicated by a turquoise rectangle).In step four a second colour is chosen by the user. We again generate images with it. In contrast to the initial generation, we use the new selected colour to set the colour for the first control point. The second and third control point keeps the before chosen colour. With this setting we generate an image. A second image is generated by keeping the old colour for the first and third and taking the new colour for the second control point. We repeat this for the last control point, where it gets the new colour. The first and second control point keeps the old colour (step number five in figure 3.2). Three different images are generated, representing the different colouring possibilities. The user has

the same possibilities as before: To continue or to stop the colouring process or to choose a new actual colour. If he/she continues, an image has to be selected (step number six). By selecting an image the user tells the program that the control point with the second chosen colour is now correctly coloured and no longer need to be taken into account if a new colour is chosen.For our example it is control point number two in step six. The next step is that the user has to choose a colour. With the newly chosen colour (step eight) we generate two different images the user can select from. One where the first control point has the initial colour and the third the new colour, and another where the first control point has the new colour and the third one the initial colour (step seven in the figure). Depending on the finally selected image we have a colouring for all three control points (step nine in the figure). This approach provides us with the possibility to offer different colourings by varying the colours for the different control points. If we have to colour 10 or more points it is possible that some combinations look similar or equal. To make it easier for the user to find the wanted colouring and to be consistent with when other high-level parameters are set, we combine these images in clusters which the user can select and explore.

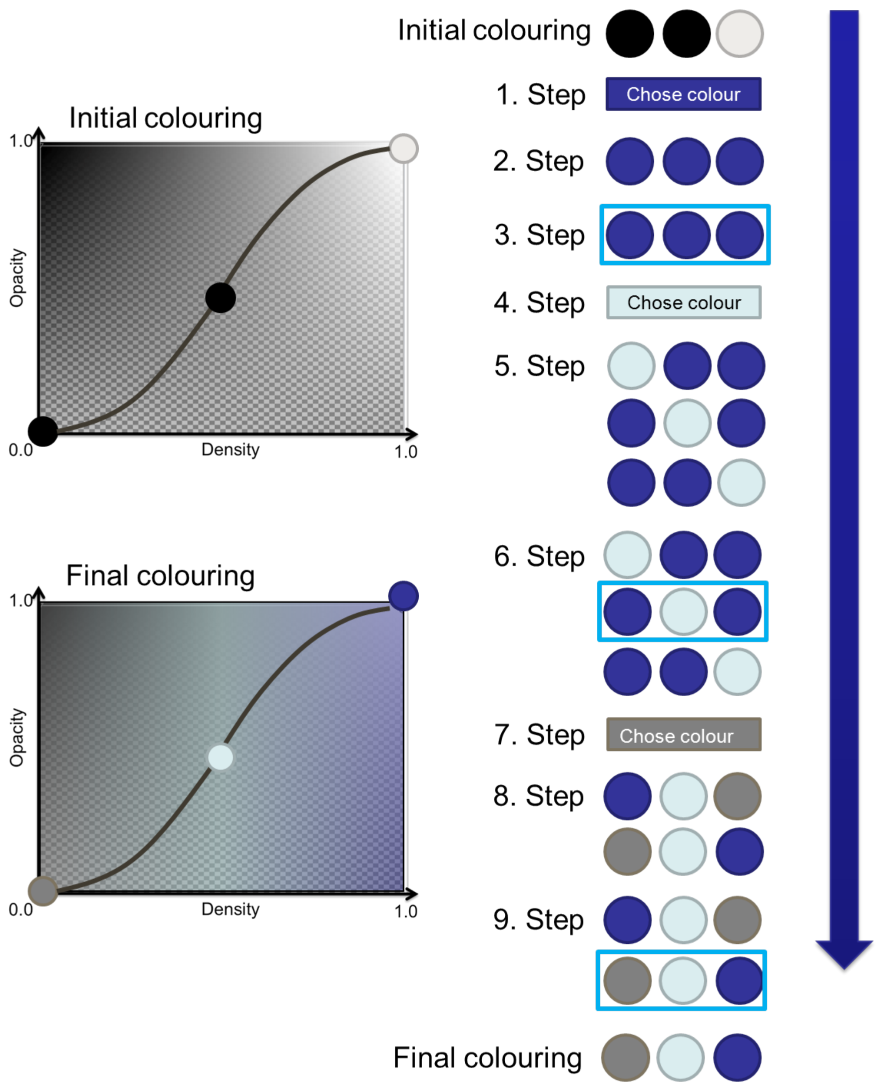


Figure 3.2: The colouring process for a transfer-function having three different control points. The colours red, green and blue are assigned to them. The red rectangle represents the chosen colour combination.

3.4.3 Generating Parameter Settings for Low-level Parameters

If the user chooses one of the high-level parameters, images with different parameter-setting values have to be generated. To cover many possibilities we use the same approach as Bruckner and Möller [11]. Random values are generated to set the low-level parameters for the selected high-level parameter.The number of values being generated depends on the number of low-level parameters which were used for the high-level parameter.One set of values represents one parameter setting to be used to generate an image. The advantage of using random values is that they are easy and fast to generate. In most of the cases they cover a sufficient number of different settings. Furthermore by using them, the number of generated images can be easily adapted at any time.
To cover the values for many different low-level parameters, where each can have a different range, we use random percent values between 0% and 100% which are than mapped to the range of the low-level parameter. Further this makes it easier when the range of a low-level parameter is adapted during the refinement.

3.4.4 Generating Images with the Random Value Sets

To generate, i.e., render, the images, we take the random values sets as mapping for the low-level parameter values, as mentioned before. They are again used as input for the rendering algorithm. Our intent is to save the rendered images and use them as input for the mapping (step 2 in the pipeline as shown in figure 3.1). However even if the parameter values are different, they can lead to equal images. If two images equal each other for all RGB values, we remove one for performance reasons.

3.5 Map Images to LLE Space

Calculating the *LLE* coordinates and clustering are the next two steps in the pipeline. In order to not overwhelm the user with hundreds of different images, we use kMeans++ and DBScan to combine similar images to clusters. Both use similarity, based on the Euclidean distance between two images, to determine whether an image belongs to one cluster or not. To determine similarity between two images, often their RGBA-values, where A is the alpha value of an image, are used as input for the Euclidean distance. Instead we use the method of Locally Linear Embedding [6] and the resulting *LLE* coordinates to calculate the Euclidean distance. This is done because we have high dimensional data as generated images. The advantage of*LLE* is that it is capable of learning the global structure of the underlying non-linear manifolds generated by the image [6]. These manifolds are most likely used by the human brain to determine similarity between real objects, images, faces, etc. [28]. Therefore we consider *LLE* to be a good approach for determining similarity between images. We use the algorithm proposed by Roweis and Saul [6] to calculate 2D *LLE* coordinates for each of the generated images, which is explained in the following section.

3.5.1 Locally Linear Embedding

LLE is based on simple geometric intuitions. It supposes that the data, sampled from some underlying manifold, used as input, consists of N real valued vectors, $\overrightarrow{X_i}$, where each has a dimensionality D [6]. For our approach N defines the number of images and D the resolution of an image. $\overrightarrow{X_i}$ contains the images in RGBA pixel values between 0.0 and 1.0. For example, an image the size of 256x256 pixels has a D dimension of one and contains $256 * 256 = 65.536$ entries.Expression 3.1 shows the definition of the vector $\overrightarrow{X_i}$ where $\overrightarrow{I_i}$ represents an image.

$$\overrightarrow{X_i} = \{\overrightarrow{I_1}, \overrightarrow{I_2}, ..., \overrightarrow{I_n}\} \tag{3.1}$$

LLE expects that each data point - for our approach, each image - lies on or close to a locally linear patch of the manifold. This leads to each image being capable of reconstruction from its neighbours by linear coefficients. Those coefficients characterise the local geometry of these patches and can therefore be used for reconstruction [6]. To reconstruct each image from its neighbours, K neighbour images have to be found.As proposed by Roweis and Saul, we use the Euclidean distance as measurement to determine K neighbours for each image. To measure the reconstruction errors, the cost function in expression 3.2 is used. The function adds up the squared distances between all the images and their reconstructions where W_{ij} is the contribution of the i^{th} reconstruction image to the j^{th} image.

$$\varepsilon(W) = \sum_i \mid \overrightarrow{X_i} - \sum_j W_{ij}\overrightarrow{X_j} \mid^2 \tag{3.2}$$

To compute the weights W_{ij} for each image, we have to minimise the cost function by meeting two constraints, and solve a least-square problem. The first constraint is that each image has to be reconstructed from its K neighbours. If $\overrightarrow{X_i}$ is not one of the neighbours of $\overrightarrow{X_j}$ then $W_{ij} = 0$. The second is that the sum of the rows of the weight matrix has to be one. $\sum_j W_{ij} = 1$. This constraint makes the images invariant to translation. The invariance to rotation and rescaling is given by expression 3.2. The weights used for reconstruction characterise intrinsic geometric properties of the data that are invariant to translation, rotation and rescaling. This implies that the same weights W_{ij} that reconstruct the i^{th} image in D dimensions also reconstruct its embedded manifold coordinates in d dimensions, where d is the dimensionality of the manifold which is supposed to be $d << D$ [6, 7]. Under the predetermined constraints we minimise the cost function, representing the reconstruction error:

Normally a Lagrange multiplier is used to enforce the constraint $\sum_j W_{ij} = 1$ and to minimize the error. To simplify and to make the algorithm more efficient we avoid using the Lagrange multiplier. Roweis and Saul proposed to solve the linear system of equations, $\sum_i C_{ij}W_j = 1$ [6,7]. C_{ij} is the local covariance matrix which is constructed by $C_{ij} = (\overrightarrow{X} - \overrightarrow{Xk_i}) \times (\overrightarrow{X} - \overrightarrow{Xk_j})$ where $\overrightarrow{Xk}$ contains the nearest neighbours of the actual image. When using this approach it is necessary to rescale the weights after solving the linear system so that the sum equals one. This yields the same result as taking the inverse of the covariance matrix. All the steps made so far construct a neighbourhood preserving mapping. The final step is to map the high dimensional $\overrightarrow{X_i}$ to the low dimensional vector $\overrightarrow{Y_i}$, representing global internal coordinates on the manifold.

Next the embedding cost function, expression 3.3, has to be minimised.

$$\Phi(Y) = \sum_i \mid \overrightarrow{Y_i} - \sum_j W_{ij}\overrightarrow{Y_j} \mid^2 \tag{3.3}$$

The embedding cost function is, as the previous one 3.2, based on locally linear reconstruction errors. In contrast to equation 3.2, the weights W_{ij} are fixed while the coordinates of $\overrightarrow{Y_i}$ are optimised. To minimise the embedding cost function, a sparse $N \times N$ Eigenvector problem is solved. The bottom d non-zero eigenvectors are an ordered set of orthogonal coordinates centred on the origin [6, 7]. For our approach we take the first two components discovered by *LLE*, meaning $d = 2$, to map the images to a simple 2D coordinate system for easier and faster calculation of the Euclidean distance. The two coordinates represent the X and Y values in the coordinate system.

3.6 Clustering Images in Groups

After calculating the *LLE* coordinates for each image, we want to check their similarity and combine similar images to clusters. As already mentioned, this step in the pipeline is important because we do not want to overwhelm the user with hundreds of images where he/she can chose from. We want looking similar images to be represented by a single one.

3.6.1 kMeans/kMeans++ Cluster Algorithm

The first clustering algorithm we want to use for our approach is kMeans [30]. kMeans clustering is often used to cluster segments within images. In contrast, we utilize it to cluster whole images. KMeans has only one parameter: *k*. It represents the number of clusters to which the data will be partitioned. The result of the kMeans clustering algorithm is that all data points belong to the cluster where they have the smallest distance to the cluster's centre. The disadvantage of setting *k* manually becomes an advantage for our approach and is also the reason why we chose kMeans. It provides us with the possibility to define the number of cluster centres. To get rid of the problem of a wrongly chosen *k* - as kMeans is known for bad initialisation of *k* which can lead to bad overall clustering - and to avoid running kMeans several times with different values for *k* we use kMeans++ [4] to choose an appropriate *k*. This approach has the advantage of finding a better distribution of the starting centres when first running the algorithm with a given *k*, and is twice the time faster than kMeans.
For our approach, *k* is a fixed value, determined by the number of images being displayed to the user in the gird-like interface. We do this to avoid having the user set *k* manually, and to limit the number of images being displayed. The algorithm starts with randomly choosing one image. The X and Y *LLE* coordinates from this image then represent the initial start centre. Next a new centre, therefore another image, is chosen randomly, where the probability for being chosen increases with the square of the distance between the X and Y coordinates of the actual chosen image and the already determined cluster centres. This step of choosing a random image and determining whether it can become a cluster centre or not is repeated until *k* centres are found.

The images are partitioned by assigning them to clusters where they have the shortest Euclidean distance to the centre. After all images are assigned to the clusters, a new centre for each cluster is calculated. We do this by taking the X and Y *LLE* coordinates of the images assigned to one cluster and calculate the mean for each cluster which represents the new cluster centre. The steps of assigning new images to clusters for the calculating of a new centre for each of the clusters are repeated until no image is assigned to another cluster.

3.6.2 DBScan Cluster Algorithm

One of the already mentioned disadvantages of kMeans++ is that it only finds convex cluster structures. It is not possible for example to find a cluster with a *round*, *S* or *Y* shape (see figure 3.3). Therefore we chose DBScan as the second clustering algorithm to find those clusters with arbitrary shapes.

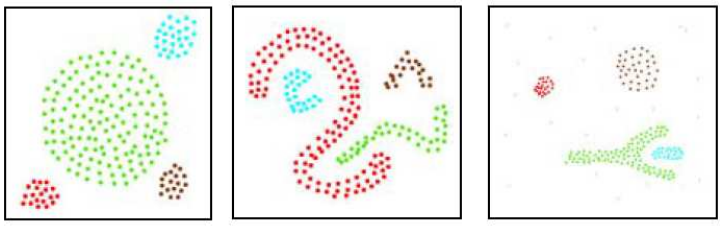


Figure 3.3: An example of the different cluster shapes which can be discovered by DBScan [5].

DBScan needs two input parameters. The first is the minimum number of images, p_{min}. A cluster must have at least one image to be treated as a cluster by the algorithm. The second is ε which defines the maximum distance an image can have to its neighbours within a cluster. Again we take the *LLE* coordinates to calculate the distance between the images.
For our approach we start the DBScan algorithm by choosing an arbitrary image from all the images generated. Then the neighbourhood of this image is determined by adding every image to this image's neighbourhood if it lies within the ε distance. The image and its neighbourhood form a cluster if more than p_{min} images are assigned to the neighbourhood. If the actual image's neighbourhood has less than p_{min} images assigned then the image is marked as noise. Still it can be added to a cluster if it is within the ε range of another image, which is already part of a cluster. If an image becomes part of a cluster, its ε neighbourhood images are also assigned to the cluster. The process continues until no further images can be found. In the next step a new unvisited image is processed.

One of the disadvantages of DBScan is that a user needs domain knowledge to set ε. Normally he/she has to make a rough estimation of the distribution of the images in *LLE* space to set an appropriate value for ε manually so as to get appropriate results. Another disadvantage of the algorithm is that it marks images which are not reachable in a distance ε from any other image as noise. These images are no longer considered by the algorithm and do not belong to any cluster. To achieve an appropriate number of clusters and to avoid many images being marked as noise, we try to implement an automatic method base on the manual approach of Ester at al. [5], which attempts to determine a good ε. Additionally the number of minimum points in a cluster is adapted by this determination process of ε. By using this method we want to see whether or not we can achieve better results for the DBScan algorithm.

Determining ε Automatically

Our approach to estimating an optimal ε is based on the heuristic of finding it manually, as proposed by Ester et al. [5]. Their basic idea is to generate a k-distance graph. This means that for every point in the set of points, the distance to its k^{th} nearest neighbour is calculated. The distance values of all points are sorted in descending order and taken as input for the graph. The X axis of this graph represents the sorted points, and the Y axis represents the distance values. An example of this graph can be seen in figure 3.4 where a k of four was used. Experiments indicated that a $k > 4$ is not necessary and does not significantly differ from the 4-distance graph [5].

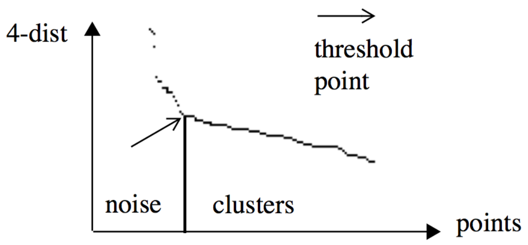


Figure 3.4: The sorted 4-distance graph for a set of points [5].

Having the graph, the aim is to find the *threshold point*. This point represents the maximum k-distance value in the thinnest cluster. It has the property that all points with a distance value greater than the *threshold point's* distance are considered as noise. All the other points with smaller distances will be assigned to a cluster. The *threshold point* is the first *valley* in the graph [5]. The *valley* or *threshold point* is indicated by the arrow in figure 3.4. After the *threshold point* was found the number of minimum points,p_{min}, is set to k, and ε gets assigned the k-distance value.

We want the user interface to be as easy to use as possible. Therefore we do not want to have a k-distance graph, and want to select ε and p_{min} automatically by estimating the *threshold point*. The idea is to start with obtainment of the distances for every image to its *k* neighbours in *LLE* space, and sort them in descending order. From this set of distance values we take the minimum and maximum distances and calculate the difference between them. This is the maximum absolute difference between any two of the distances in this set. To estimate the valley we add a certain percentage to the maximum absolute difference to the minimum distance. We estimate the percentage to be 10%. The estimated ε may not be accurate enough and lead to a wrong number of clusters, which is not suitable for our user interface (see section 3.7.1). Therefore we run the DBScan algorithm with the before chosen ε and check the number of clusters. If this number is too big or too small we slightly change ε. We check whether increasing ε by

5% would lead to a better distribution of the clusters and less images marked as noise. If this is not the case we decrease ε by 5% and again check the distribution and the noise. We run the DBScan algorithm repeatedly with changed ε until we achieve a suitable number of clusters which is determined by the grid size of the user interface.

3.7 Display Centres

The next step, step 4, in the pipeline is to display images to the user for selection. When running the kMeans++ algorithm we already obtained X and Y *LLE* coordinates for each of the clusters, representing their centres. The images having the smallest Euclidean distance in *LLE* space to the calculated clusters centres are chosen for display.
For clusters generated by DBScan, it is a little bit more difficult. The algorithm is dense based and no centres are determined by the algorithm itself. Therefore we have to find the centre for each cluster manually by calculating the arithmetic means between the minimum and maximum coordinates of all images within a cluster. The image lying closest to this point is taken as the cluster centre being displayed to the user in the grid-like interface.

3.7.1 Grid-Like User Interface

For our approach we want a user interface which is easy to understand and use. As a grid or a spreadsheet like interface is a good way to present data to users [18, 19] we us it for displaying and selecting the images. Our grid interface has three rows and three columns. A total number of nine cluster centre images can be presented to the user at once. As the image could be too small for the user we want a view where he/she can see an enlarged version of the image with the parameter set with values of the selected image. Further we want to give the user the possibility to view the image in more detail. In the case of our prototype which is for a volume visualisation we want the image which the user selects to be rendered with the actual parameter values in a separate view, i.e., the render view. The user can then rotate, scale and translate the volume within. A schematic illustration of the interface can be seen in figure 3.5. The right hand side shows the grid interface, where each cell shows an image representing a cluster centre. On top of it, the high-level parameters can be selected. On the right hand side the actual selected image appears in the view. We do not want to show the actual parameter settings for the image in our interface so as not to confuse the user by displaying the many different low-level parameters. The user has the possibility to manually add a new view for the low-level parameters where the values can be set manually if necessary. As it is the aim of this thesis to not set parameters manually, we hide this view by default. There are two possible ways to navigate through the grid interface. The first includes selection of one or more images, exploration of the clusters of the selected images or refinement of parameter values.This is described in detail in section 3.8.2. The second is backward navigation which allows the user to navigate backwards in the exploration history. An additional option to the exploration view is the high-level parameter editor. It should offer expert users, who are able to determine parameter combinations and their influence to the volume, the option to define their own high-level parameters or change pre-defined ones. The editor allows the user to change the distribution of the random values and the number of images being generated where each can have different parameter setting values.

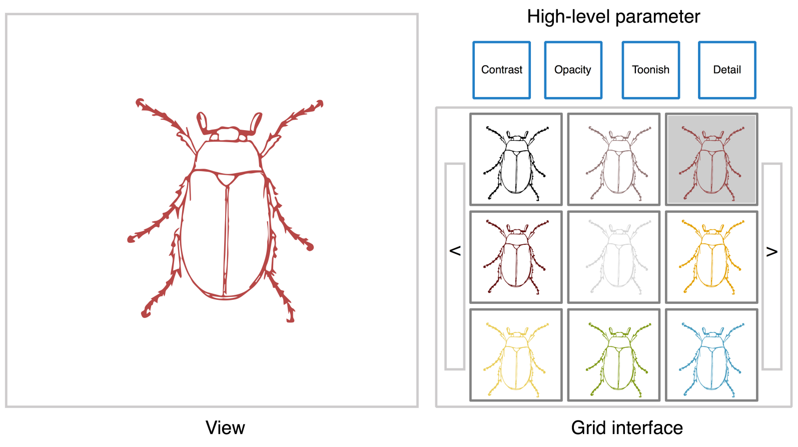


Figure 3.5: Schematic illustration of the user interface.

3.8 Select an Image/ Images

The idea for selecting images is that after the cluster centres are displayed, the user selects those centres that look approximately like the envisioned final visualisation. This does not only include images that match the target visualisation as a whole but also such images, where only some parts fit better. This may mean that desired parts of the volume are visible or better to identify. For example, we want to highlight the teeth of a skull volume. In order to do so we select images where the teeth are clearly visible. Furthermore we imagine the root of the tooth being slightly opaque. Therefore images which fulfill this criterion are selected as well. After one or more images are selected we offer the user three possibilities: to explore, to refine or to select the final image.

3.8.1 Explore Generated Images

To explore and refine are important parts of the pipeline. These two steps allow the user to explore different parameter values and also to refine them without knowledge of the underlying low-level parameters, and without using any sliders or combo boxes.

The exploration process starts after the generation, mapping and clustering of the images for a selected high-level parameter (steps 1 - 3 in the pipeline in figure 3.1). Nine cluster centre images are presented to the user. He/she selects the appropriated images as described in section 3.8. We expect that the selected images represent the different clusters, where each of the clusters consists of similar images to the selected ones. The idea is to collect all the images contained in the clusters. If the total number of collected images is smaller or equal to the grid

size, we display all of them to the user and he/she can again select the appropriate images. If the total number is greater than the grid size, we cluster all the images again and display the new cluster centres to the user.
It is possible that a cluster only consists of one image which is the image being displayed. This can occur for several reasons. One reason is that, as already mentioned, the total number of images is smaller or equal to the grid size. Therefore we already displayed all images available to the user and new ones have to be generated. Another reason could be that a cluster only consists of one image. This happens for kMeans++ when an image representing a centre was randomly chosen and no other image was assigned to this centre. For DBScan this only happens when p_{min} is set to one and no other image can be reached in ε distance. If one of these reasons is the case, and if the user selects one of the displayed images and wants to explore the similar images in a cluster, a refinement is made.

3.8.2 Refinement of Low-Level Parameter Values

When exploring the different images there are two possibilities. The first is that the user finds the wanted result of his/her visualisation in the generated set of images. Therefore no further steps are necessary. The second possibility is that the user explored all the images and he/she is still not satisfied with the result. If this is the case we have to generate new images which are similar to the already selected ones. We could again vary the full setting range for each belonging low-level parameter to the selected high-level one and increase the possibility of obtaining images equal to those which were already dismissed by the user, or we make use of the already selected images and their parameter setting values. Therefore we collect all the parameter values for all low-level parameters which were present during the before made exploration. For each of the low-level parameters the minimum and maximum parameter value out of the collected data is determined. These values indicate the new range in which the parameter becomes varied when new images are generated. If the distance between the minimum and maximum value is too small, then the low-level parameter will no longer be varied. The intent of this approach is to limit the variation range for each before varied low-level parameter and therefore refine them.

Refinement of the Transfer-function
By using our approach it should be possible to refine the transfer-function as well. We want to refine the opacity of each of the control points and also try to limit the range of the dense points which get this opacity.
First we want to refine the opacity for each control point. We collect the different Y axis positions (their opacity values) which they had during the before made exploration of the images. As for the other high-level parameters, we determine the minimum and maximum value out of the collected data for each control point. These two values are the new variation range. If the distance between the minimum and maximum is too small, the control point will no longer be varied and be seen as a fixed point.
Second we try to achieve other curve shapes and also want to refine the opacity values for dense points lying between two fixed point. The idea is to add new additional control points between them. The number of new control points depends on the distance between the two fixed points

and also the minimum and maximum variation range. Both depend on the opacity values of the fixed points.

3.8.3 Changing High-level Parameters

Within the exploration process it is possible to change the high-level parameter. Certain high-level parameters have low-level parameters in common. Our first intent was to keep the parameter values of low-level parameters which both an already chosen and a new selected high-level parameter have in common. Parameters do influence each other depending on the actual value and the combination. Therefore we can not assume the outcome of a convergent behaviour if we keep the parameter values of low-level parameters which high-level parameters have in common.Thus we want to offer the user two possibilities which he/she can choose from when changing high-level parameters within an exploration process. The first possibility is to vary all low-level parameters in their full range; even those which two high-level parameters have in common. The second possibility is to keep the value of overlapping low-level parameters and to only vary the ones which two high-level parameters do not have in common. The user can try both options and take the one that he/she gets better results with.

CHAPTER 4

Implementation

In this chapter the implementation, based on the approach in chapter 3, is described. In the first two sections we describe the technology used and the program for which the prototype of our approach – use of a plug-in - was implemented. We continue describing the different steps of the pipeline from 3 in figure 3.1 from the implementation side. First we start with generation of the images, which are the input for Locally Linear Embedding, i.e., the mapping, followed by the clustering, displaying and exploring/refining of the different images.

4.1 Technology

Our approach was implemented for VolumeShop in Visual Studio 2008 as a two plug-in solution. The programming language used was C++ in combination with the Qt framework for the user interface, OpenGL for rendering, and CUDA [31] and CULA tools [32] for parallel mathematical calculations.

4.2 VolumeShop

VolumeShop is an interactive software for volume illustration [8,9]. It is plug-in based and offers five different main types of plug-ins which contain sub plug-ins which a user can use to build up a project.The following list shortly describes the different types:

- *Editor* plug-ins offer various elements to interact with the displayed volume, like setting parameters or the transfer-function.
- *Interactor* plug-ins add different mouse controls, like rotation, translation and scaling. The camera types which define the field of view and the near and the far plane.
- *Importer* plug-ins are responsible for loading different volume data type,s like RAW, .dat, etc.

- *Exporter* plug-ins offer different export types which the user can choose from to save his/her final result as an image or video file, for example.
- *Render* plug-ins are responsible for generating the final image which is then displayed in the *View*. The *View* is the canvas where the volume is displayed. An example for one of the *Render* plug-ins is the one for direct volume rendering.

For PasSAd we use a combination of six different plug-ins. Four of them are provided by VolumeShop. These plug-ins are necessary for our approach to add basic functions, like mouse control and camera control. In the following list we give a short description of the four plug-ins.

1. The *CompositorSimple* plug-in executes all the following plug-ins in a sequence and provides an interface for them to exchange data.
2. The *ColorTransferFunction* plug-in handles the transfer-function and serves as input for the direct volume rendering algorithm to set the opacity and colour of dense points.
3. The *InteractorPerspectiveCamera* plug-in is used for setting up a correct perspective view for the volume.
4. The *InteractorViewingTrackball* plug-in is used so that the user can rotate, scale and translate the displayed volume in the *View* with the mouse.

The two plug-ins which implement our approach are *EditorParsad* and *RendererParasad*. They are of the type *Editor* and *Render*. We had to use two plug-ins because only in the *Editor* plug-in were we able to overwrite the standard GUI and replace it with our own interface.By using the *Render* plug-in, we gain access to OpenGL. An advantage of implementing it as a plug-in is that ParSAd can be added to every project in VolumeShop as it implements a complete volume renderer and graphical user interface.

The *EditorParsad* plug-in implements the grid interface in Qt, the high-level parameters, and is linked to the plug-ins which are responsible for loading of the volumes and the handling of all user input. The *RendererParasad* implements the actual generation of the images, the mapping to *LLE* coordinates and the two clustering algorithms. Further, it is responsible for saving the images temporary in the RAM or to a hard-disk. We save this combination of plug-in as a pre-set. Pre-sets can be used in VolumeShop to save an actual project, containing all the necessary plug-ins. To use our approach, a user simply has to load this pre-set. However, only the *EditorParsad* plug-in is visible to the user. The actual layout can be seen in figure 4.1. It consists of two different linked views: the render view, where the volume is rendered with the parameter values of the selected image, and the exploration view, where the different cluster centres are displayed in the grid interface.

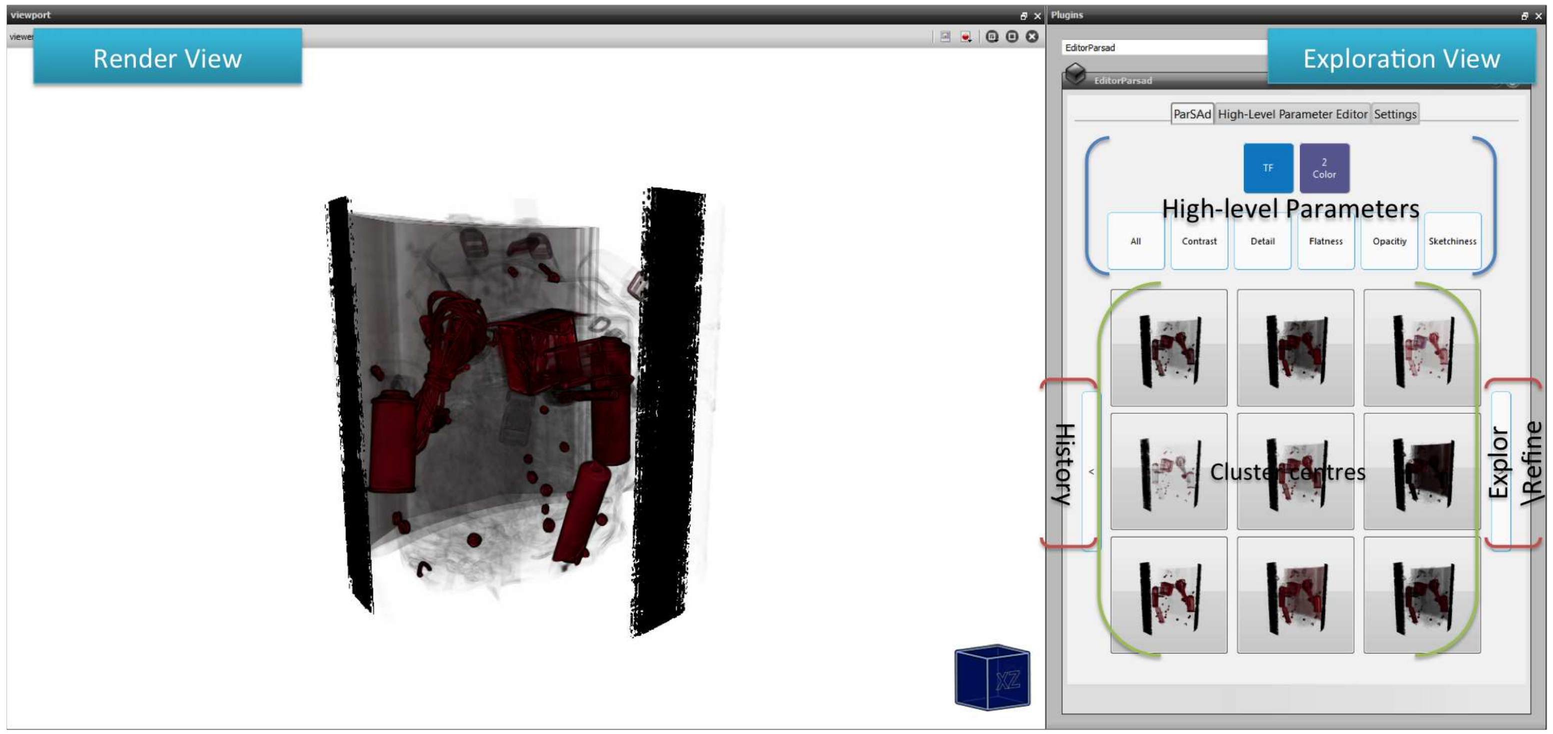


Figure 4.1: Screen-shot of the render and exploration view.

4.3 The ParSAd Pipeline and Classes

For the implementation we follow the pipeline from figure 3.1 as described in chapter 3 of this thesis. Before we describe in detail each step in the pipeline from the implementation side, we give a short overview of the different classes of each plug-in.
The steps taken from the generation of images, to the mapping of them to *LLE* coordinates, and to the clustering of them are all done in the *RendererParasad* plug-in.The implementation of the plug-in consists of five different classes (see figure 4.2).The main class of this plug-in is `RendererParasad`. It is responsible for the plug-in being loaded and the other classes instantiated.It implements the interface to make the plug-in capable of communication with other plug-ins. The actual rendering in OpenGL of images also takes place in this class. The generated images are passed to the `ImageHandler` class. It saves the images and makes them available to the `LLE`, `KMeans` and `DBScan` classes. The *LLE* coordinates for each image are calculated in the `LLE` class which uses the `LLE CUDA` class to transfer heavy calculations to the GPU. We describe this in more detail in section 4.5.1. In the classes `KMeans` and `DBScan` the different clustering algorithms are implemented. We tried to implement the classes which calculate the `LLE` coordinates and the clustering so that they could be reused in any other visualisation framework. Both classes take images as input. Therefore another framework only has to provide pre-generated images to reuse our classes.

The last steps in the pipeline - displaying of the cluster centres, and selecting, exploring and refining of image(s) - are implemented in the six classes of the *EditorParsad* plug-in (see figure 4.3 for the class diagram).Further it handles the user input. The main class is `EditorParsad`. As for the `RendererParasad` it is responsible for loading the plug-in, instantiating the belonging classes and enabling the plug-in to communicate with other plug-ins. It instances the `GeneralSettings` class which manages all the important settings including the low-level and the high-level parameters for ParSAd. The `Varyer` class is used by the `HighLevelParameter` and the `TFParameter` class to generate the random values for the low-level parameters. The `GeneralSettings` class is not only responsible for saving the settings; it also manages the communication between the `Parameter`, `HighLevelParameter` and `TFParameter` class.

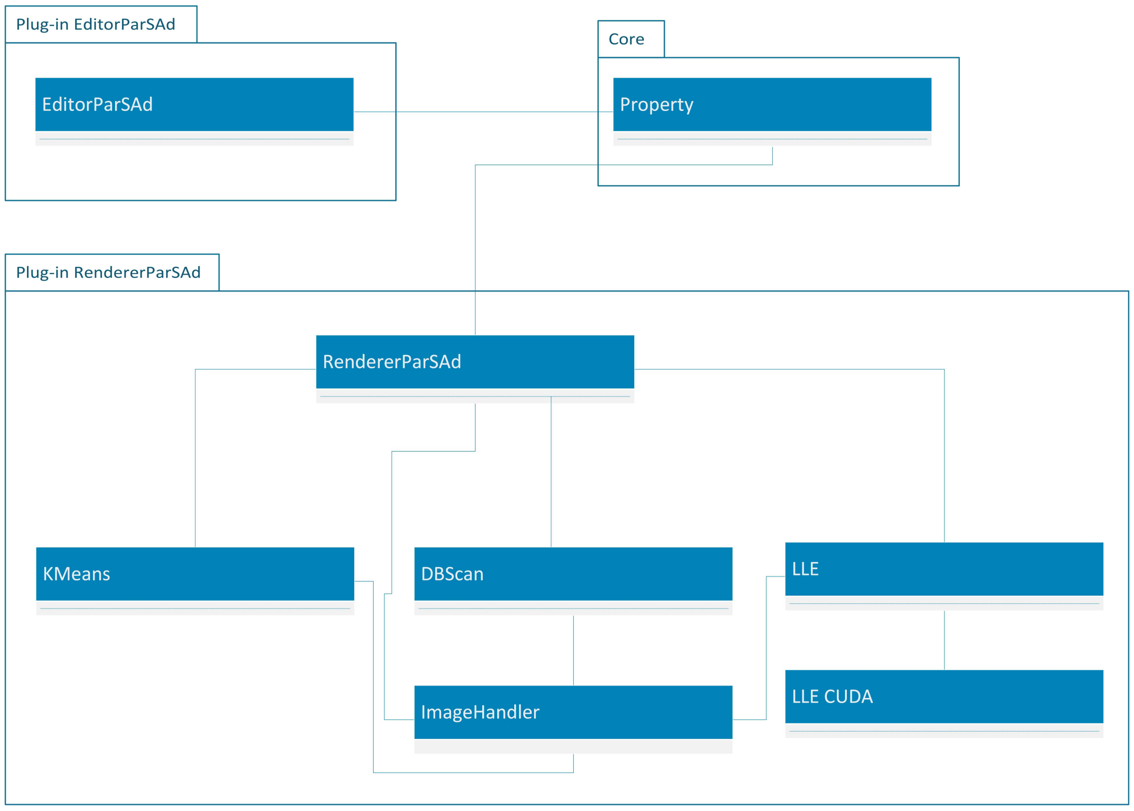


Figure 4.2: Abstract class diagram of the renderer plug-in showing the different classes and how they are related to each other, shown by the blue line between the classes.

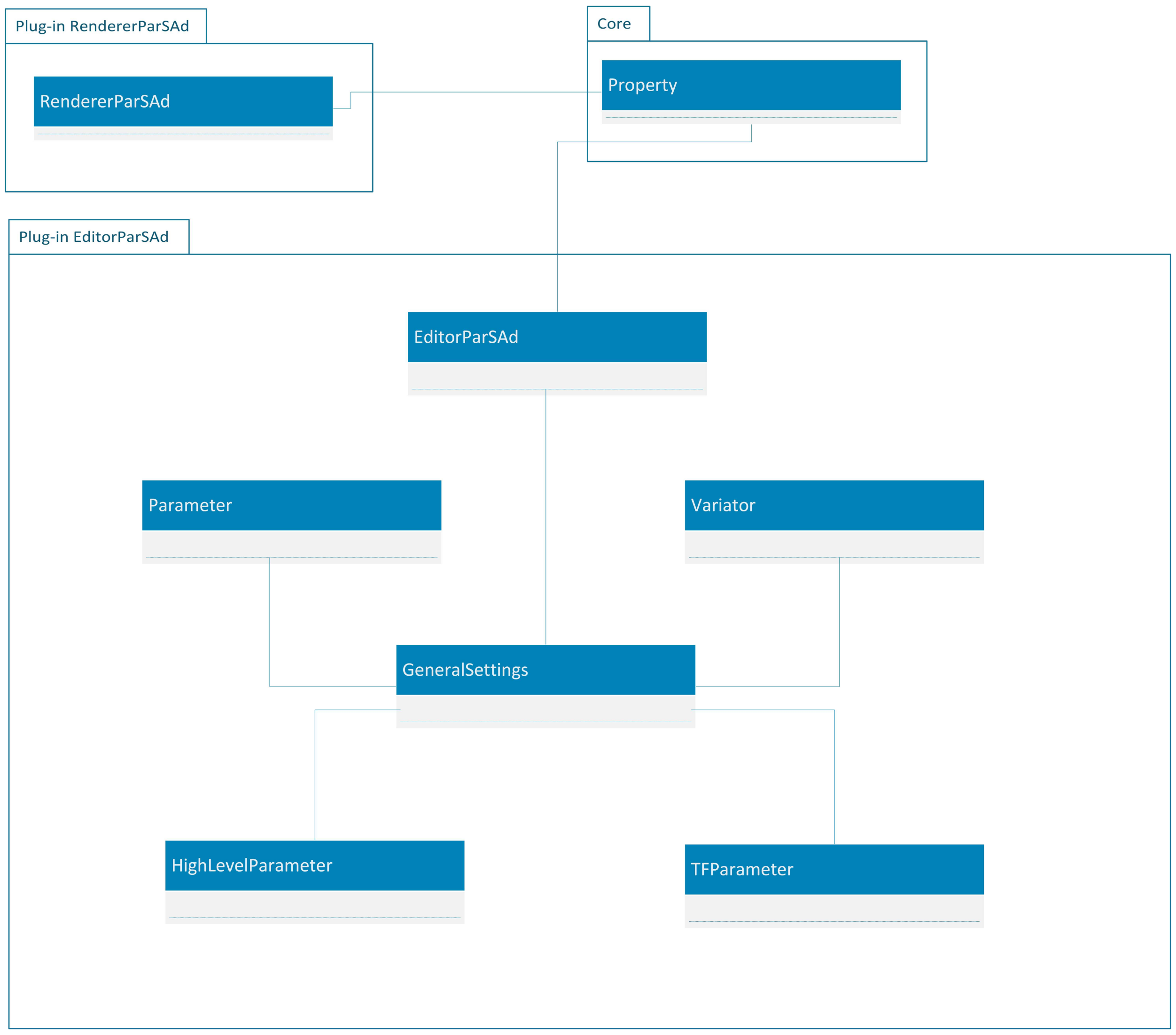


Figure 4.3: Abstract class diagram of the editor plug-in showing the different classes and how they are related to each other, shown by the blue line between the classes.

4.4 Generate Images

To generate images we first have to load the Volume via the importer plug-in. Our intent is to find the parameter values which would correspond to the intended final visualisation of the volume. To achieve this we first start with generating different images, where each has different parameter values. We use high-level parameters - which are described in detail in section 3.4.1 - to determine which low-level parameter's values are varied. Depending on this we render images of the volume with the varied values as input for the direct volume rendering algorithm.

4.4.1 Varying Low-Level Parameter Values

We want to generate many different images with different parameter values when the user chooses one of the high-level parameters. We have two requirements for the algorithm which generates the different values. The first is that we want to get random numbers between 0 and 100. The second is that we want to generate an arbitrary number of different random value sets which do not contain equal numbers.
To generate random numbers and to fulfill the first requirement we us the `srand` function of `C++`. The advantage of this function is that it can take a seed as argument. To make the numbers distinctive enough for one value set we pass the number of seconds since 00:00 hours, Jan 1st 1970 UTC as seed to the `srand` function. For the second requirement we introduced a *diversion* value in combination with a hash-map to quickly compare the actual generated value set with all already generated ones. The *diversion* value can be a number between 1 and 100 and defines the variation of all the value sets. For example, if the *diversion* value is set to five, the variation between the values in all sets is 5%. The hash-map contains all the generated value sets. The advantage of using a hash-map is that entries can be easily accessed by a key. We us this key to quickly compare an actual created value set with all the already created ones. The key for each value set in the map is a string which is compounded of the single values of the actual created value set where each is divided by the *diversion* value. When the key is created, we try to save the value set in the map. If an entry already has this key we dismiss the actual created value set and create a new one. We repeat this until a defined number of value sets has been created. Figure 4.4 illustrates key generation and insertion into the hash-map. We have three different value sets with the values (0,18,56,33), (1,19,57,32) and (56,99,0,3). Every single value is now divided by the diversion value of five. We get two equal hash-keys for the first two sets. Therefore the second set can not be put into the hash-map because there already exists an entry with that key.

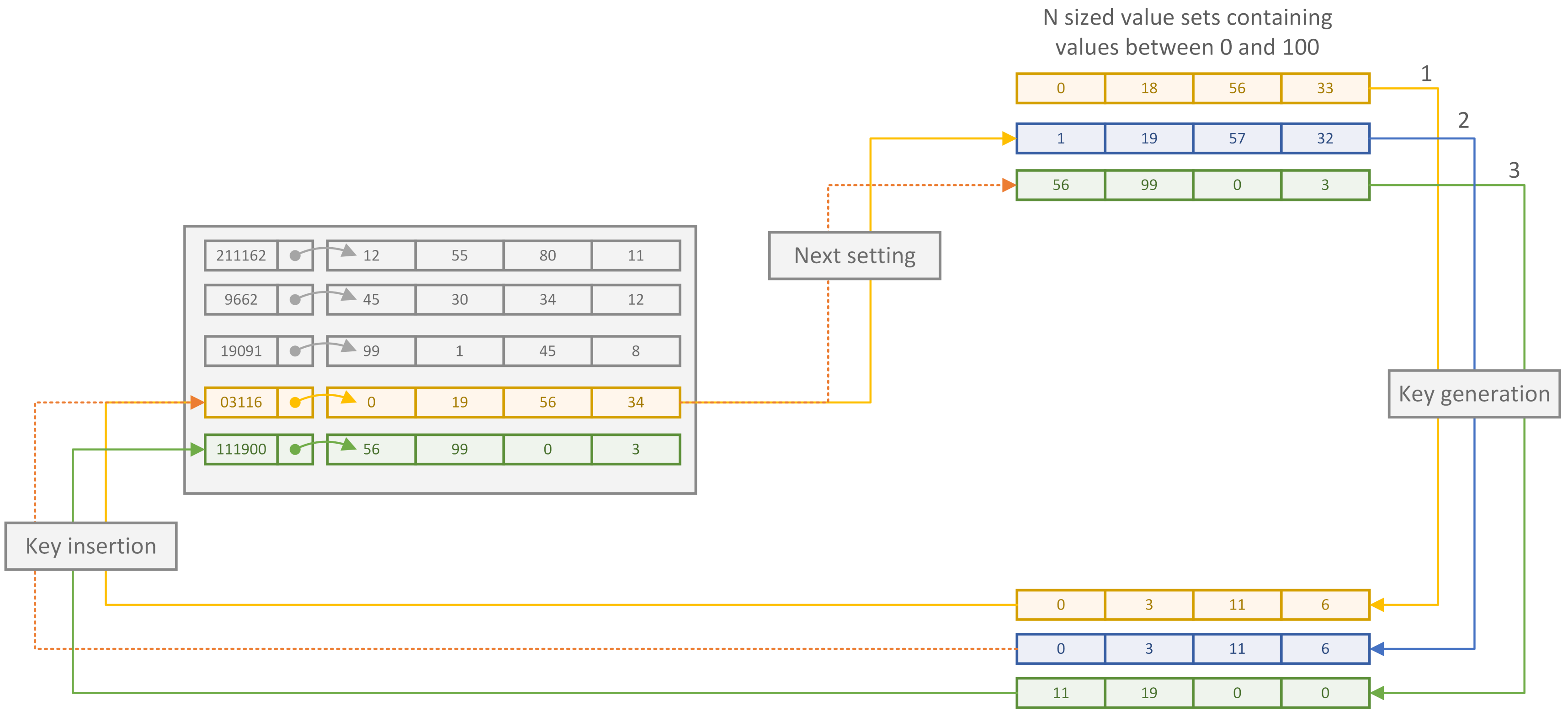


Figure 4.4: Example for generating a key for a value set and adding it to the hash-map.

4.4.2 Plug-in Properties and Inter-Plug-in Communication

The implementation of our approach in VolumeShop - as already mentioned - is comprised of two different plug-ins. All the high- and their low-level parameters including their value sets are saved in the *EditorParsad* plug-in. The initial idea was that the *EditorParsad* plug-in would pass the actual value sets for a selected high-level parameter as object to the *RendererParasad* plug-in which uses them for rendering the different images.However, it is not possible that plug-ins exchange objects, which do not belong to the core implementation of VolumeShop. Therefore we had to find another way of passing data - like the value sets - between the plug-ins and came up with using the internal property format of VolumeShop which can be used to pass double values and images between plug-ins. The disadvantage of using a property is that we need one for every single low-level parameter and for every image we want to exchange. An example for setting the property for the *Shading intensity* parameter, having a maximum of 1.0, minimum of 1.0 and an initial value of 1.0, can be seen in listing 4.1.

```
GetPlugin().GetProperty("Shading Intensity").require(Variant(1.0f,0.0f,1.0
    f)).addObserver(&m_modObserverUpdate);
```

Listing 4.1: Properties for the *Shading intensity* low-level parameter

To link two properties with each other and to exchange data types - like float, boolean, integer and images - between them, we had to create a property with the same name and type in each of the plug-ins.One in the *EditorParsad* plug-in and a second one in the *RendererParsad* plug-in. If one property is assigned, for example, a new value for a low-level parameter in the editor plug-in, the linked property in the rendering plug-in has this value assigned as well and vice versa. The same is done for exchanging value sets and RGBA values between the *EditorParsad* plug-in and the *ColorTransferFunction* plug-in.

4.4.3 Rendering Images

We explained how the value sets are generated and how the two plug-ins can exchange data. Finally we are able to generate and render images.The sequence in figure 4.5 shows how the rendering of the images works.

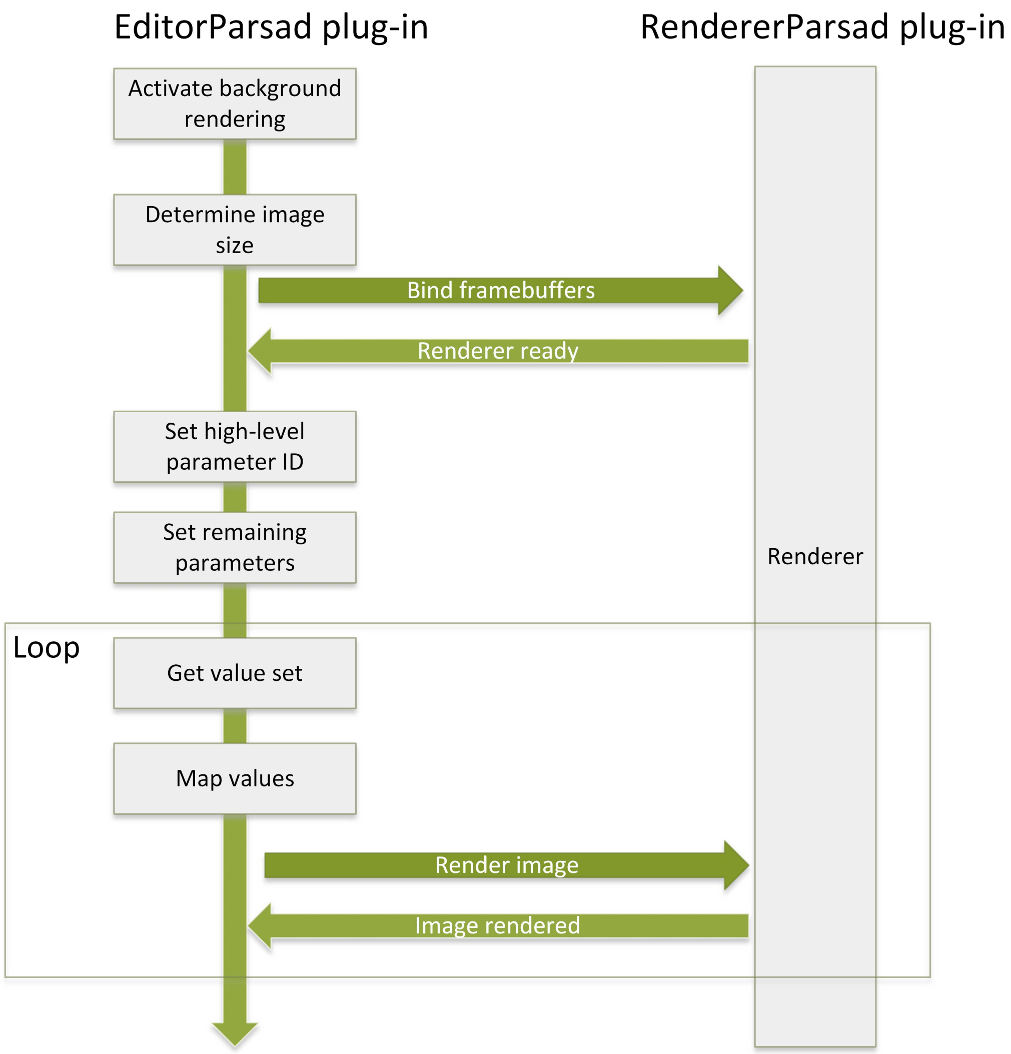


Figure 4.5: Sequence of rendering different images for each value set of the selected high-level parameter.

The rendering is initiated by selecting a high-level parameter. Before rendering, whether a volume is loaded and the size of the images to be generated are checked. When a high-level parameter is first instantiated, an arbitrary number of value sets, where each set represents a different image, is created. Each of these sets for the selected high-level parameter is used as input for the direct volume rendering algorithm. This is why the exchange of data between plug-ins is important. Every time the low-level parameter in the *EditorParsad* is changed, so is its property. Therefore it automatically passes the values of each changed low-level parameter to the property of the *RendererParsad* plug-in which then uses the values as input for the rendering algorithm. Before we set the low-level parameter values belonging to the chosen high-level parameter, we have to map the entries of the random values in the value set to the range of each low-level parameter. The mapping is done by using the equation 4.1. To obtain the actual mapping value, val_{act}, we calculate the distance between the minimum, p_{min}, and the maximum p_{max} value the low-level parameter can have and take the value set entry, vec_{entry}, as a percent measurement of the distance.

$$val_{act} = p_{min} + (vec_{entry}/100 * (p_{max} - p_{min}) \tag{4.1}$$

Until now we have only set the low-level parameters assigned to a high-level parameter. If the high-level parameter does not contain all low-level parameters needed as input for the rendering algorithm, we have to also assign values to the low-level parameters which are not part of the high-level one. We differ between two cases for these. The first case is that the values have never been set by using any of the high-level parameters, therefore these low-level parameters are in their initial state. If this is the case, we assign initial values from table 4.1 to them. These values lie, for most of the low-level parameters, in the middle of their total range. We do this because low-level parameters influence each other depending on their values. As we do not know which values the assigned low-level parameters to the selected high-level parameter get, we can not predict a good initial value for the others. Additionally we want to keep the values of the not assigned low-level parameters equal for every value set we use so as to generate images. We only want to show the user the influence of the actual varied low-level parameters on the volume.

Finally the rendering is done and the images are saved as `Image` object in the `ImageHandler` class. The rendering continues until all generated value sets for the selected high-level parameter are used to render an image.

Rendering Images Using the Transfer-Function

The rendering of images for which the transfer-function was selected as a high-level parameter works almost exactly as described before. In contrast to the other high-level parameters, we have to check whether we have to render an image where the position of the control points in transfer-function are varied or only their colour is changed. For varying the control points it works as with the other high-level parameters. The value sets have the size of the number of control points. The values the set contains are mapped again using equation 4.1 to the range in which a control point can be varied on the Y axis. The result of the mapping is passed to the

Low-level parameter	**Initial Value**
Sample distance	0.5
Level	0.5
Window	1.0
Shading	True
Cell shading	False
Shading intensity	1.0
Diffuse shading	0.875
Specular shading	0.125
Specular exponent	32.0
Gradient shading	0.975
Gradient opacity	1.0
CoNbR	0.1
Gradient scale	1.0
Fall-off	2.0
CoNsR	0.1
Silhouette scaling	1.0
Silhoutte Fall off	0.25

Table 4.1: The different low-level parameters and their initial values.

ColorTransferFunction plug-in where the control point is set. When applying a value set, if the user would open this plug-in he/she would see the change of the control points directly in the transfer-function.
For setting the colour we do not use value sets. This is done by using the `QColorDialog` which is opened by the *EditorParsad* plug-in. It is a simple colour chooser provided by the Qt framework. The user simply clicks on the colour he/she it wants and the RGB values are passed from the *EditorParsad* plug-in to the *ColorTransferFunction* plug-in via linked properties. Then the colouring is done as described in chapter 3, section 3.4.2.

4.5 Map Images to *LLE* Coordinates

The next step is to calculate the *LLE* coordinates of the different images. This is done, as proposed by Roweis et al. [6], in three steps:

1. The images' neighbours are determined by calculating the Euclidean distance between all the images.
2. Calculation of the weight-matrix.
3. Finally, the eigenvalues, depending on the weight-matrix, are calculated to determine the *LLE* coordinates.

Parts of the *LLE* algorithm, like calculating K neighbour images, calculating an entry in the weight-matrix and solving the Eigenvalue problem, are very CPU-intensive. Therefore we came up with the idea of using CUDA to parallelise these single steps for calculating the *LLE* coordinates to minimise the pre-processing time.

Figure 4.6 shows the calculation of the *LLE* coordinates for the images by using CUDA. First, using CUDA, we start with comparing all images with one another to obtain all neighbours.After, K neighbours are taken to create the neighbourhood matrix for a single image. CUDA was again used to accelerate this calculation. Out of this matrix the covariance matrix is created, which is necessary to solve the linear system. This is done in parallel by using the CULA tools library [32]. Finally the results of this calculation are taken to create the final entry in the global weight matrix. The steps from creating the neighbourhood matrix to calculating the entry in the global weight matrix are done for every single image. When the global weight matrix is filled with all the entries from each image, as described before, CULA tools are again used to solve the Eigenvalue problem for this matrix. As a final result we get the *LLE* coordinates for all the images.

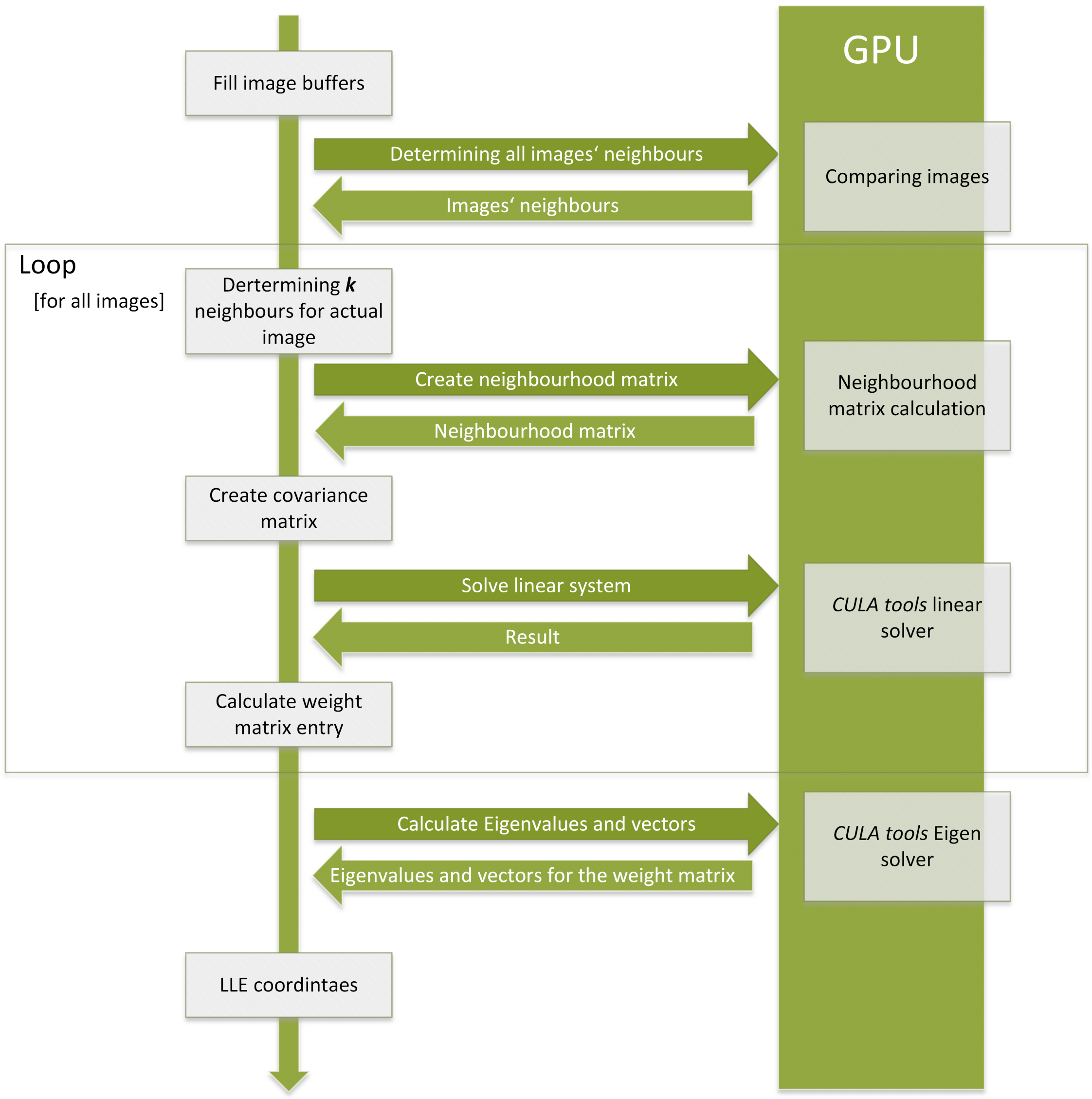

GPU
Fill image buffers
Determining all images‘ neighbours
Comparing images
Images‘ neighbours
Loop
[for all images]
Dertermining *k* neighbours for actual image
Create neighbourhood matrix
Neighbourhood matrix calculation
Neighbourhood matrix
Create covariance matrix
Solve linear system
CULA tools linear solver
Result
Calculate weight matrix entry
Calculate Eigenvalues and vectors
CULA tools Eigen solver
Eigenvalues and vectors for the weight matrix
LLE coordintaes

Figure 4.6: The process of calculating the *LLE* coordinates of all the images by using CUDA.

4.5.1 Image Comparison in Parallel

Before starting to calculate the *LLE* coordinates of a single image, K neighbour images - which the actual image can be reconstructed with - have to be found. To obtain these neighbours for one image, the Euclidean distance between all the images has to first be calculated. For this step we calculate the distance between each pixel and not their *LLE* coordinates. Our approach is designed for an arbitrary number of images. To calculate the distances between all of them can take a lot of time if using an iterative approach where every single pixel is compared with the same pixel in the other image. Therefore we use CUDA in parallel to accelerate this process. To do so we came up with the following implementation:

The idea is to have two arrays with image objects as entries. On array represents the main array and the other the buffer array. After initializing the size of the two arrays, which depends on the number of images and the memory space on the GPU, we calculate the distances between the images as follows:

1. The main array is filled with images from the before generated ones. The number of images in the array depends on the GPU memory size. The images the array is filled with are removed from the total set of images to avoid double calculation. Next the main array is uploaded to the GPU and all the distances between the images in the array are calculated. Instead of looping through all the pixels of the two images to calculate the Euclidean distance between them, we separate an image in different blocks. In CUDA a block is an accumulation of threads, whose maximum number depends on the block size.For example, having a block with the size of 32 $times$ 32 results in a total number of 1,024 threads.Hence a block is like process and different blocks can be executed in parallel. We want each thread to calculate the Euclidean distance of a pixel between two images. To get the total Euclidean distance between two images, the concept of reduction proposed by Marks et al. [33] is used and adapted for our approach to sum up all the distances between threads and the blocks.

2. After calculating the distances between the images within the main array, we fill the buffer with images coming again from the total set of generated images. These images will not be removed as was done for the main array. The distances between all the images in the main array and the buffer array are calculated. This step is repeated until all the distances between the main array and remaining images in the set are calculated.

3. Finally the main array is voided and we begin again at step 1, filling the main array with images. These three steps are repeated until the distances between all the images are calculated.

Almost equal images are removed during the distance calculation if their distance is smaller than a given threshold.

4.5.2 Calculating the Final *LLE* Coordinates

Now we have the distance to all neighbours for every image. Depending on the defined K for *LLE*, the neighbourhood matrix for each image is created. It has K rows. Every row represents the K^{th} neighbour image subtracted from the actual image. To do the subtraction in parallel we use again CUDA. An image is treated as one dimensional vector and we do a simple vector subtraction where for every entry in it a thread is started. The result of the subtraction is again a vector which becomes a row in the neighbourhood matrix.
After the neighbourhood matrix is created, we have to calculate the covariance matrix out of it. This is done by using the matrix and vector classes of the *Eigen-library* [34]. To solve the linear system, CULA tools are used, which do the calculation on the GPU to accelerate the process (see listing 4.2).

```
\\Filling the Eigen-matrix with the rows of the k-distance Matrix
for(int a = 0; a < image->kdistanceMatrix.size();a++){
        for(int b = 0;b < image->kdistanceMatrix[a].size();b++){
                covariance(b,a) =  image->kdistanceMatrix[a].at(b);
        }
}
\\Creating the covariance matrix
Eigen::MatrixXf centered = covariance.rowwise() - covariance.colwise().
    mean();
covariance = centered.adjoint() * centered;
\\Solving the linear system
culaStatus s = culaSgels("N",N,N,NRHS,covariance.data(),N,solution.data(),
    N);
\\Normalize solution
solution = solution/ solution.sum();
\\Returns solution as column for calculating the entry in the weight
    matrix.
weightM->col(image->number) = solution;
```

Listing 4.2: The calculation of the covariance matrix and solving of the linear system using CULA tools and setting the column for the global weight-matrix from the `solveLinearSystem()` method in the `LLE` class

The solution of the linear system is returned as column for calculating an entry of the global weight matrix which is shown in listing 4.3.

```
for(int i = 0; i < images[0]->neighbourNrIndex.size();i++)
{
        //Index of the actual image
        int jj =images[0]->neighbourNrIndex[i];
        //Subtracting the weights of the actual image
        //from the entries at (ii,jj) in the global
        //weight matrix and vice versa (jj,ii)
        M_M(ii,jj) = M_M(ii,jj) - w(i);
        M_M(jj,ii) = M_M(jj,ii) - w(i);
        //Saving the neighbour position to square the actual entry.
        neighbourhoodIndecies.push_back(jj);

}
//Adding the squared weight of the actual image
// to the global weight matrix
for(int a = 0; a < neighbourhoodIndecies.size();a++)
{
 for(int b = 0; b < neighbourhoodIndecies.size();b++)
 {
        M_M(neighbourhoodIndecies[a],neighbourhoodIndecies[b]) = M_M(
            neighbourhoodIndecies[a],neighbourhoodIndecies[b]) + (w(a) * w
            (b));
 }
}
```

Listing 4.3: The calculation of the global weight matrix

The steps for creating the neighbourhood matrix, solving the linear system and entering the results in the global weight matrix are repeated for every image. The final result is the global weight matrix. This matrix is used to calculate the bottom three Eigenvalues by again using CULA tools. Finally the last two Eigenvalues are used as *LLE* coordinates for the image.

4.6 Cluster

After the image's *LLE* coordinates are calculated, we cluster them by using kMeans++ or DB-Scan.The clustering itself is implemented in the *RenderParsad* plug-in. The final centre images are rendered and passed to the *EditorParsad* plug-in to be displayed to the user in the grid interface. In the following sections we describe the implementation of kMeans++ in CUDA and the implementation of DBScan including the automatic determination of ε.

4.6.1 kMeans++

Determining the cluster start centres is the first step of the kMeans++ algorithm. We use the method as described in section 3.6.1. These cluster centres found by the improved selection of the start centres are then used for the first run of the clustering algorithm. To determine the Euclidean distance for each image to its k neighbours in *LLE* space, CUDA is again used to obtain all distances at once and to determine the images' cluster centres in one step.

Clustering and Distance Calculation of Images in Parallel for kMeans++
A 1, the actual centre, to n-1, the remaining images excluded the actual centre, comparison is implemented in CUDA, to calculate the distances for every image in the total image set to the cluster centres.The *LLE* coordinates of all the images are uploaded to the GPU. It returns an array of distances between one centre and all the other images. This means by using CUDA the distances for one cluster centre can be calculated at once. After repeating the distance calculation for every centre, new means are calculated for every cluster. Then the clustering runs again and the number of images which change clusters is counted and saved as a percentage number. If this number is smaller than a given threshold, the cluster centres and their assigned images are saved. However, if the percentage number is higher, kMeans++ continues running until the number of changing images is under the given threshold. Finally the clusters are saved and the centre images returned to the *EditorParsad* plug-in to be displayed to the user.

4.6.2 DBScan

The second clustering algorithm we implemented is DBScan. In contrast to kMeans++ it is implemented to use the CPU for distance calculation because the performance of a single run using an iterative approach on the CPU of the DBScan algorithm is fast enough. For DBScan the struct `DensePoint` and `Cluster` in listing 4.4 was used.

```
struct DensePoint
{
        std::string id;
        float x;
        float y;
        bool visited;
        int cluster;
        bool noise;
};

struct Cluster
{
        int nr;
        DensePoint* center;
        std::vector<DensePoint*> belongingImages;
};
```

Listing 4.4: The sturcts used for DBScan clustering.

For every image a `DensePoint` is created. The X and Y coordinates of the point correspond to the *LLE* coordinates of the actual image. Depending on whether the user chooses the automatic method to determine ε or not, ε is defined, otherwise a fixed ε is passed to the clustering algorithm. Finally the images are clustered as described in section 3.6.2. Every cluster is represented as `Cluster` object, as can be seen in listing 4.4. This struct eases the process when doing a new clustering with the images assigned to a selected centre /selected centres.

Determining ε automatically

The ε parameter of DBScan determines, as already mentioned in the design section, whether a certain image is assigned to a cluster or not. However, selecting a good ε - which means that the number of clusters is bigger than one and the number of noise images is smaller than 50% of the total number of images - is a difficult task if the user has no knowledge of the approximate distances between all the images.

We start with the creation of the k-distance graph as proposed by Ester at al. [5]. We loop through all the `DensePoints` and calculate the distances between them.The k neighbours with the shortest distance to the `DensePoint` are saved in a global map which is finally sorted by the shortest distance.

The difficult part is finding an approximation of the first valley in this graph automatically. We use the sorted graph to get the maximum and minimum distance values. Then we do the calculation of the first valley as supposed in chapter 3, section 3.6.2. Our first guess for the percentage to add to the minimum distance from the maximum total distance was between 10 and 20 percent. We implemented the algorithm first, then we tested it and determined a sensible value in a trial and error process. A value of 15% was found to deliver a number of clusters closest to our specification. If the first approximation of the first valley did not produce the wanted results we try an iterative more slow approach which takes the approximated first valley as input value. Every distance value in the k-distance graph is used to determine the final ε by running the DBScan algorithm with it and counting the number of clusters created and the number of noise images. To determine whether the actual value for ε is good or bad, we set a criterion that a cluster should not contain more than 30% of the total number of images. If all ε are tested and none fulfill the criterion, we take the ε closest to the given criterion. Next a *for loop* is started, decreasing the criterion by 2.5%.This loop continues until the criterion reaches 60%. If a good ε is still not found, we take again the one which approximates the criterion best.

4.7 Display Images as Centrers

The *EditorParsad* plug-in consists of three different tabs. The first tab, which can be seen in figure 4.7, represents the main view of our approach and is called *ParSAd*. It displays the grid, with nine cells in combination with a right and left arrow button. Each cell is a button on which the image is displayed. If the user hovers over one of the buttons, the image is rendered and displayed in the render view. If the user wants to select an image he/she simply has to click on the button. If an image/ images is/are selected, the right arrow button will be enabled. The user can continue the exploration of the assigned images to the selected centre(s). The left arrow button enables the user to navigate backward in the exploration process.
Above the grid, the high-level parameter buttons are positioned. The number of buttons changes dynamically if the user adds or removes a high-level parameter. Per default, five high-level parameters are available as described in section 3.4.1, table 3.2. To change the transfer-function, the *1.TF* button has to be pressed. After a transfer-function is chosen by the user, its colour can be determined by pressing the *2.Color* button. The second tab is the *High-level Parameter Editor* which can be seen in figure 4.8. Its purpose is to give an expert user the possibility to change, delete or create high-level parameters. The following options are available:

- The *diversion* value for the generation of the value sets as described in 4.4.1.
- The *number of value sets* determines how many different value sets for the high-level parameters should be generated.
- For *shading* the user can choose between two options. The first one is normal shading, the second one is cell or toon-shading in combination with parameters.
- The *low-level parameters*, as mentioned in table 3.1 in the design section, can be added or removed to a high-level parameter by simply clicking the "+ "or "- "button.

The final tab is the settings tab (figure 4.9). It determines which clustering algorithm should be used. DBscan has an extra option which is whether the automatic method or a fixed ε should be used when running the algorithm.
The last three options in the settings apply to the *LLE* algorithm. These are the number of K neighbours, the image size for the input image and the threshold for removing similar images for determining their Euclidean distances when calculating the K neighbours.

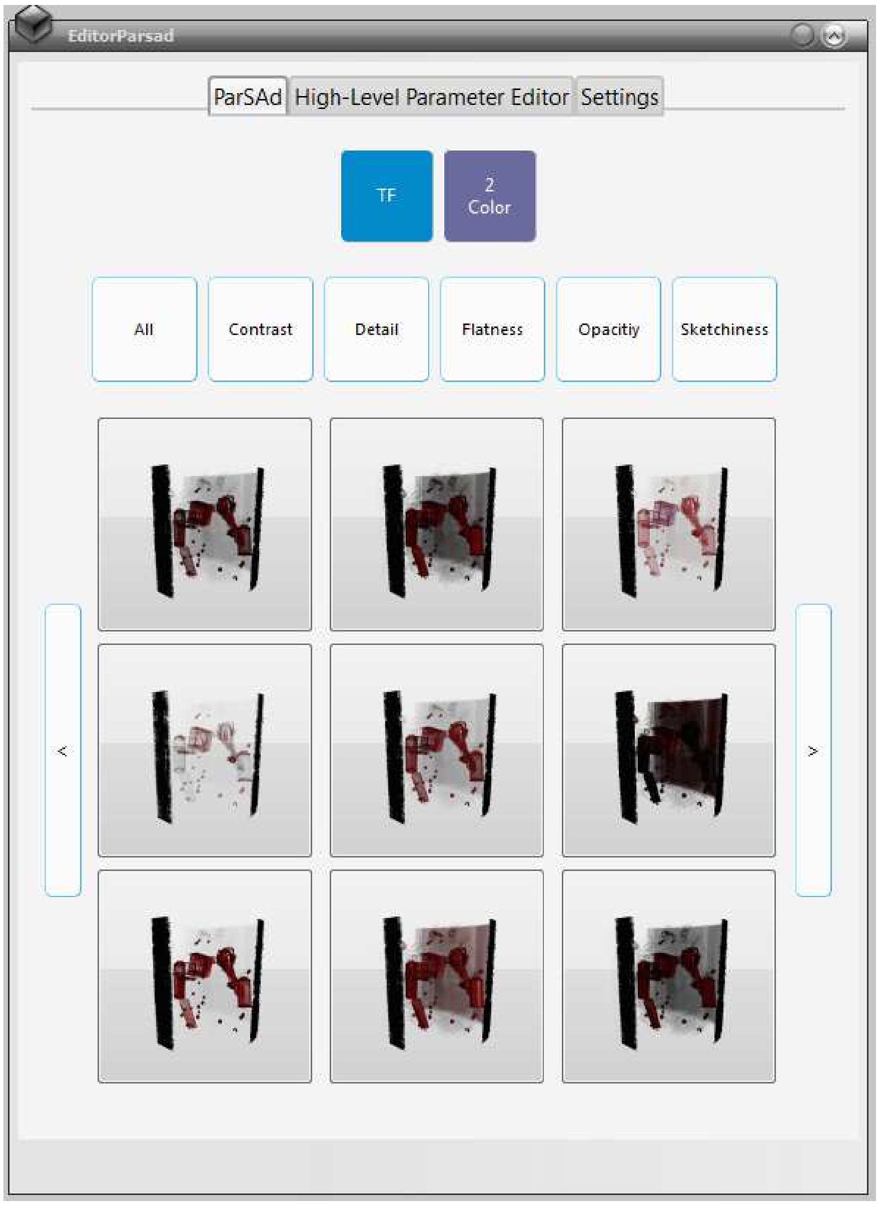
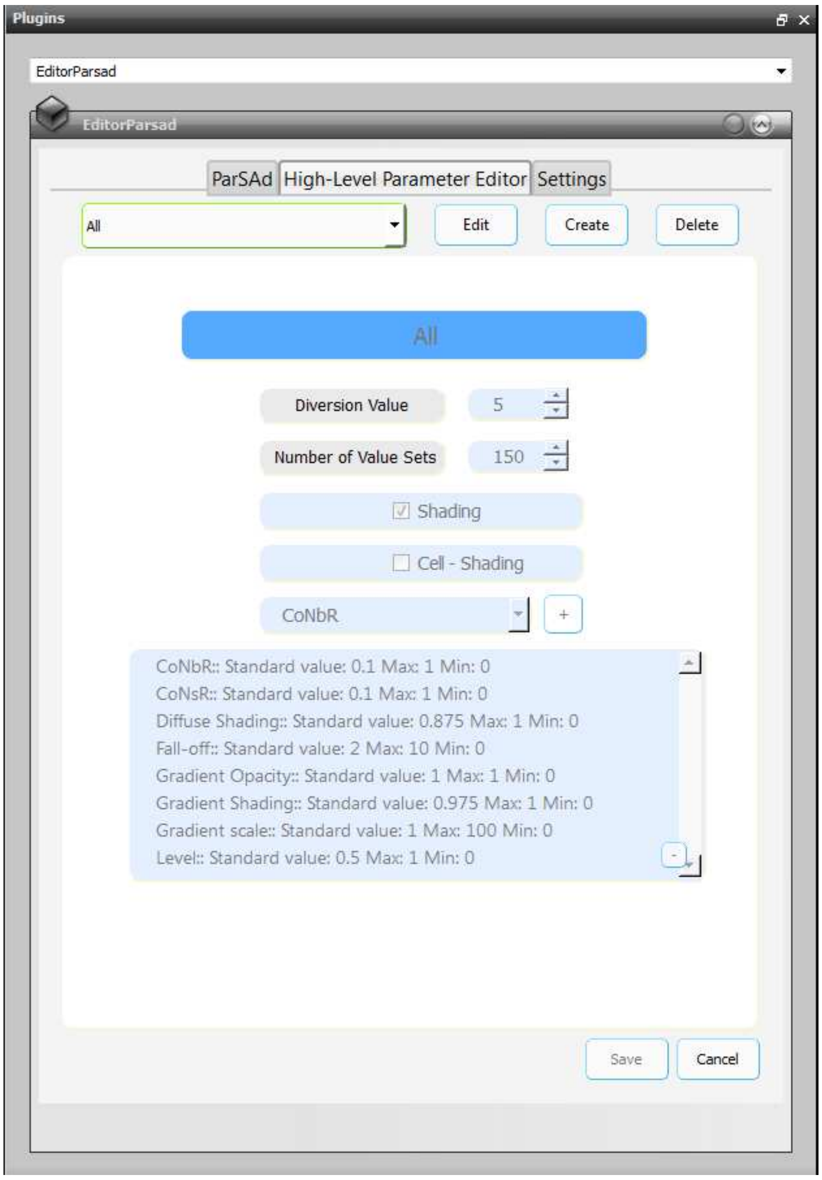
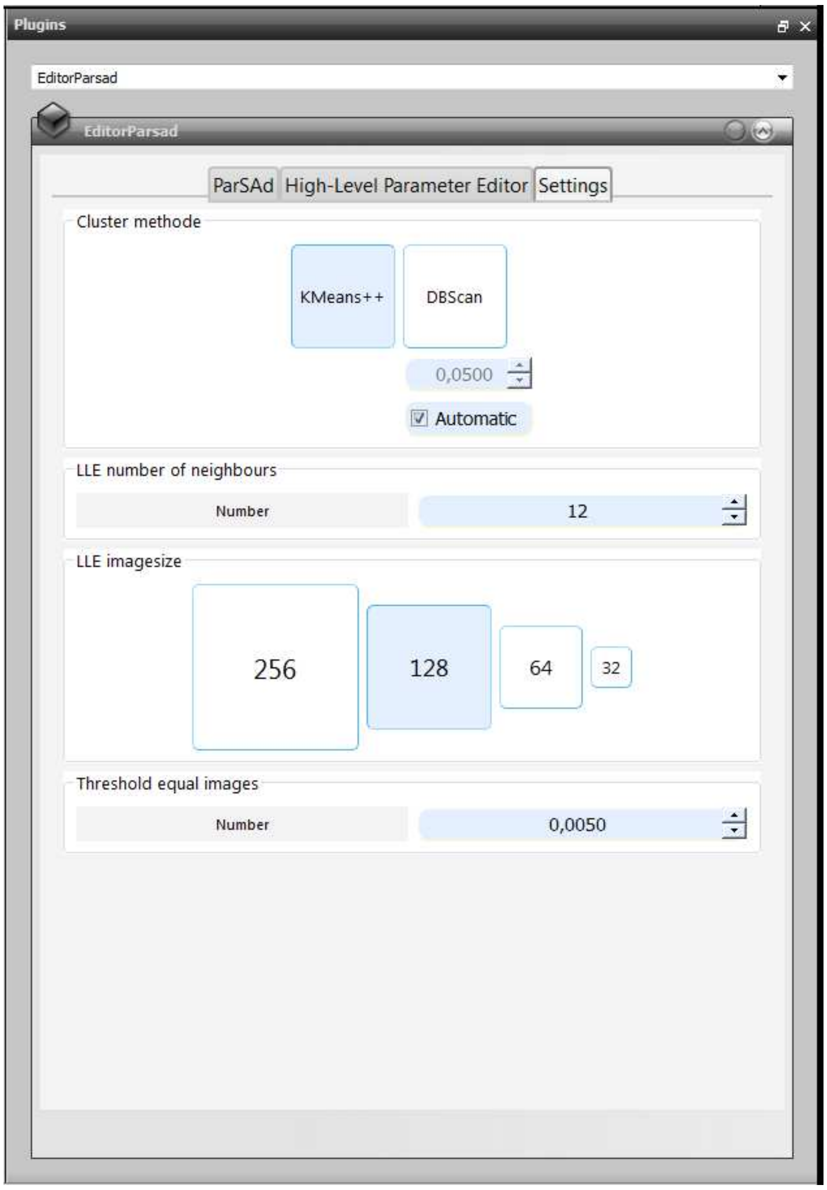


Figure 4.7: The main view which is visible when loading the *EditorParsad* plug-in.



CoNbR:: Standard value: 0.1 Max: 1 Min: 0
CoNsR:: Standard value: 0.1 Max: 1 Min: 0
Diffuse Shading:: Standard value: 0.875 Max: 1 Min: 0
Fall-off:: Standard value: 2 Max: 10 Min: 0
Gradient Opacity:: Standard value: 1 Max: 1 Min: 0
Gradient Shading:: Standard value: 0.975 Max: 1 Min: 0
Gradient scale:: Standard value: 1 Max: 100 Min: 0
Level:: Standard value: 0.5 Max: 1 Min: 0

Figure 4.8: Editor for creating, editing and deleting high-level parameters.



Plugins

EditorParsad

EditorParsad

ParSAd | High-Level Parameter Editor | Settings

Cluster methode

KMeans++ | DBScan

0,0500

[✓ Automatic]

LLE number of neighbours

Number: 12

LLE imagesize

256 | 128 | 64 | 32

Threshold equal images

Number: 0,0050

Figure 4.9: The settings for ParSAd to change the cluster method and the *LLE* settings.

4.8 Select an Image/ Images

When all images are mapped to *LLE*, clustered and the centres displayed to the user, then the next intended step is to explore the assigned images to the selected centres. Our approach differs between exploring and refining.

4.8.1 Exploring Generated Images

To start exploring similar images, the user has to click on one or more of the images in the grid. This enables the explore button. If the user has not found the final visualisation he/she can click on this button and start to explore the assigned images to the selected centre(s).Depending on whether the number of assigned images to the selected centre images is bigger than the number of which can be displayed in the grid, clustering is applied. We pass the assigned images either to the kMeans++ or the DBScan algorithm. If the number is smaller or equal to the grid size, no clustering is applied and all the remaining images are displayed at once to the user. Every time the explore button is pressed a `HistoryEntry` object is created. It saves the current state, which includes the displayed images and their assigned ones. If the user has already done an exploration step and presses the back button, the `History` class is called which saves all the `HistoryEntry` objects in a stack, and restores the last entry. After restoring, the user can change the selected centres and again conduct an exploration of the assigned images, or if not the first exploration step, he/she can continue navigating backwards.

4.8.2 Refining Low-Level Parameter Values

If one of the clustering classes returns no new images for display, the refinement is started. The user is informed by a message box that the parameters will be refined according to his/her exploration history. Every time a centre image is selected and further explored, the values of the belonging low-level parameters are saved. This is done for all selected centres during an exploration. The refinement method runs through all these values for every low-parameter assigned to a high-level one to find the maximum and minimum values for every low-level parameter within an exploration. These two values represent the new range in which the mapping of a value set is done.If the range is smaller than a given threshold, which is 0.01, than the low-level parameter becomes fixed and its value can no longer be changed. We take this threshold because we tested each low-level parameter and saw that when increasing or decreasing it by adding or subtracting 0.009, it no longer influenced the volume visualisation. Finally new value sets are generated and mapped to the new range.

4.8.3 Refine Control Points in the Transfer-Function

The refinement of the transfer-function works almost similarly to the one of the other high-level parameters. When the user explores different settings of the control points in the transfer-function, we save every opacity value - which corresponds to the position on the Y axis - the different control points had during the exploration.For each of the points, the minimum and maximum opacity value during the exploration is determined. The distance between the two

values is the range for varying the position of the single point. If this range is smaller than a given threshold, which is 0.08, then the point is fixed and no longer be varied. We also tested the transfer-function to see if increasing or decreasing the opacity value of a control point by 0.08 would influence the displayed volume; it did not in this case. Therefore it became the threshold value.
We also wanted to do a refinement on the X axis as described in section 3.8.2. First we checked whether two neighbouring control points were fixed. If this condition was true and the distance between two fixed points was long enough, three new control points were added between the two fixed ones. Their distribution was adapted to the distance and the hight difference between the two fixed points. The opacity values of the new control points depended on the slope between the two fixed points and was interpolated. The range in which the new points were varied was limited by the height of the before and following neighbour control point.

CHAPTER 5

Results

This chapter is about testing ParSAd. First the test methodology and then the testing itself is described.

5.1 Test Methodology

The method was tested on a machine with a 2.6 GHZ Intel Core i7 processor, 8GB RAM and a NVIDIA GeForce GT 650M with 1024MB RAM. In order to obtain knowledge on whether our approach was working and on whether we were able to achieve the same resulting images by using ParSAd as we would with manually set parameters, we defined test-cases for different volumes.
The image size and the number of neighbourhood images influence the accuracy of the *LLE* coordinates for each image, which may lead to widely scattered coordinates and similar images not laying together in the coordinate system. This again can influence the clustering so that non similar images are assigned to one cluster. To be able to obtain good results for clustering and make the test of our approach independent from the *LLE* settings, we first tested different K's and image sizes for *LLE* for each volume in combination with the two different clustering algorithms. Additionally we checked whether *LLE* is working as similarity measurement for kMeans++ and DBScan and whether our automatic method for selecting the parameters for the DBScan algorithm is working as well.Depending on these results we continued with executing the different test-cases in section 5.4.

5.2 Test Volumes

For testing we used four different volumes covering different sizes, resolutions and scanned objects. The following list describes them in more detail:

The *Stag Beetle* [35] is a volume with a resolution of 832 x 832 x 494 voxels with a size of approximately 625 megabytes. It is one of the biggest volumes the method has been tested with. Interesting parts of this volume are the inner areas, like the stomach, the gut and the complicated structure of tubes (tracheae and tracheoles) with which the beetle breaths through.

The *Stented Abdominal Aorta* [36] was the second biggest volume on which our method was applied. With a resolution of 512x512x164 voxels and a size of approximately 87 megabytes. Interesting parts were the pelvis, the spiral and the abdominal aorta with a stent.

The *Skewed Head* [37] is a CT of a human head with the resolution of 184 x 256 x 170 and a size of approximately 15 megabytes. The resolution of this volume is not very high. However it is possible to quickly render interesting parts, like the teeth, sections of the human skull and the cervical spine. This volume is suitable for fast testing.

The *Backpack* [38] is, as the name indicates, a CT of a backpack with the resolution of 512 x 512 x 373 and a size of approximately 55 megabytes. The interesting parts of this volume is the inside of the backpack, the different bottles and the contents of the chest.

5.3 Testing Volume-Specific *LLE* Settings, *LLE* in Combination with Clustering Algorithms and the Automatic Detection of DBScan Parameters

This section describes a test which is related to LLE in combination with the two clustering algorithms and the automatic selection of the ϵ parameter of DBScan. The main purpose is to find values for K - the number of nearest neighbours for LLE - which are, in combination with an image size, suitable for our further tests. This is of importance because this has a strong influence on the accuracy of *LLE* when approximating the linear data [39]. As it was not aim of this thesis to implement an automatic method for determining a good K-value for each volume before running the LLE-algorithm, we had to manually find suitable settings for our further tests. For each volume, different parameter settings for *LLE* were used (see table 5.1). To determine which of these settings works best for the clustering, we checked the assigned images to each cluster centre in terms of whether they looked similar or totally different to the image representing the cluster centre. When running this test we additionally wanted to show that *LLE* in combination with the two clustering algorithms can be used to cluster similar images. If totally different images would to be assigned to one cluster for all the different settings, we assumed that *LLE* is not working in combination with the clustering algorithm. Another main contribution of this thesis is an automatic way of detecting a suitable ϵ-value for DBScan. Every setting test was executed for kMeans++ and DBScan for which the automatic method was enabled. After running the test we checked whether our method was able to find a suitable ϵ or not. If one was found we checked the number of clusters and how similar the images in one cluster were. We compared this number with the cases where no good ϵ was found.
To check the similarity of the assigned images to one cluster we used the *Structural Similarity (SSIM) Index*. This measurement implements three different kinds of comparisons of two sig-

nals, where each signal represents one image. First the luminance followed by the contrast and the structure of the images were compared. Finally all three results were combined to measure the overall similarity, i.e., the SSIM index. Equal images have a SSIM index of 1.0 [40] while two totally different images would result in a value of 0.0.

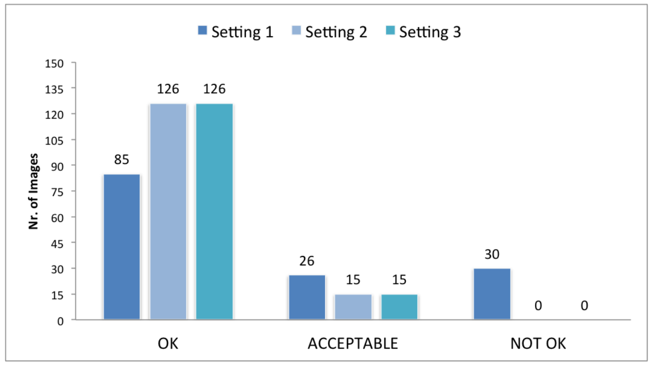
Test-setting	**Image Size(pixels)**	**Number of neighbours (*K*)**
1	256x256	20
2	256x256	12
3	256x256	5
4	128x128	20
5	128x128	12
6	128x128	5
7	64x64	20
8	64x64	12
9	64x64	5

Table 5.1: *LLE* settings

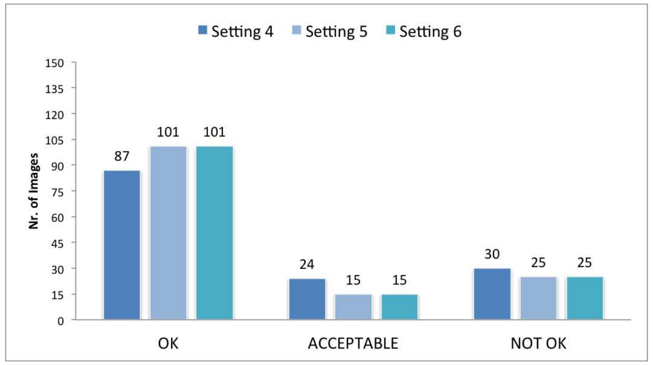
We ran our test with nine different *LLE*-Settings which can be seen in table 5.1. For image generation we used the same set, which contains of 150 different entries, where each entry had six different parameter values, for all our setting tests. By using the same set for every test, we ensured that for each run the same images were created, leading to comparable results. For every setting, the same 150 images were created and then clustered by using either kMeans++ or DBScan. Images which had been generated were save to a hard disk as *jpg* images. For each image we calculated the SSIM index which the actual image has between it and the assigned cluster centre image. Next the index for all the other cluster centres was calculated. If an image had a greater SSIM index than another cluster centre, we classified this image as *NOT OK*, otherwise as *OK*. Before classifying an image as *NOT OK* we checked whether the SSIM index to the other cluster centre was tolerable. To find an SSIM index for the tolerance we compared the SSIM index of images looking quite similar. If two images had a difference to each other of the SSIM index of 0.05, no visual difference was noticeable; therefore we set the tolerance to 0.05. This means the difference between the index value of the assigned cluster centre and the actual tested one did not have to be smaller than 0.05. If the image could be assigned to another cluster centre but the difference of the index values was tolerable, we classified this image as *ACCEPTABLE*. For the DBScan algorithm we also counted the images marked as *NOISE* by the algorithm itself. Depending on the number of images classified as *OK* we determined whether the *LLE* setting improves or worsens the clustering and which setting we had to choose for the other tests for the different volumes.
For evaluation we used the SSIM implementation in MATLAB from Wang et al. [40], which compares two images and plots their SSIM index. After categorizing them, the number of *OK*, *ACCEPTABLE* and *NOT OK* images were counted for each setting. Then bar charts for every volume were created, where for each image size the results of the different K's were compared.

A test setting from 5.1 was regarded as good for a volume in combination with the tested clustering algorithm if the number of *OK* and *ACCEPTABLE* images was high. An interpretation for every volume and its result was given in the following paragraphs. Finally a table was made with the recommended setting for every volume. For all the volumes and setting tests, over 11,000 images were automatically classified.

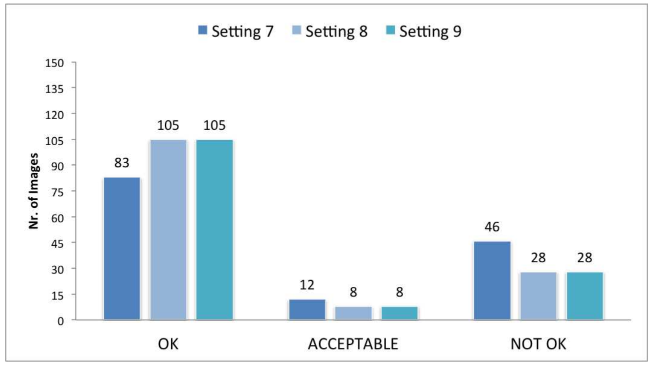
Stag Beetle
Figure 5.1 shows the results for the stage beetle after running the different stetting-tests. The bar charts 5.1a to 5.1c show the results for the kMeans++ algorithm. We obtained the best results with test settings two and three. A total number of 126 out of 150 images, where nine were removed because the were equal to other images, were classified as *OK*. Also the number of *ACCEPTABLE* images was equal for all of them. Taking these results into account we can assume that a K of five or 12 and an image size of 256 leads to *LLE* coordinates where similar images lie together in the coordinate system. This again induced the number of *OK* images for kMeans++. As the stag beetle can have, depending on the chosen transfer-function and parameters, many fine structures, we would recommend taking test setting number three because, as we already explained, *LLE* takes the rendered images with the size of 256x256 pixel as input for the calculation of the coordinates. We assumed that having a bigger image size for the rendered image would preserve more of these structures. We also decided to take setting number three instead of two because of the performance. Calculating the *LLE* coordinates by taking only five neighbours is much faster.
The bar charts 5.1d to 5.1f show the results for the DBScan algorithm. The numbers of *OK* and *ACCEPTABLE* images were not as high as for kMeans++. Test settings one, four, five, six and seven had less than 45 *OK* images. For these settings our method for automatically finding an adequate ϵ did not work. The chosen ϵ was too big. Figure 5.2 shows the distribution of the *LLE* coordinates for test setting number one, two and three. The scattering of the *LLE* coordinates increased with a smaller K. If the ϵ chosen is too big, for example, as occurred for test case number one, and the *LLE* coordinates lie very close together (see figure 5.2a), then almost all images of the value set were assigned to one big cluster, even those which are not similar to it. This again leads to many images classified as *NOT OK* or *NOISE*. The best results were achieved for settings number two, three, five and six. As for kMeanss++ we recommend setting number three for DBScan. It preserved most of the fine structures when rendering the image and a small K lead to better performance and results for the DBScan clustering algorithm because if the coordinates are more scattered, a chosen ϵ that is too big would not lead to too many wrongly classified images.



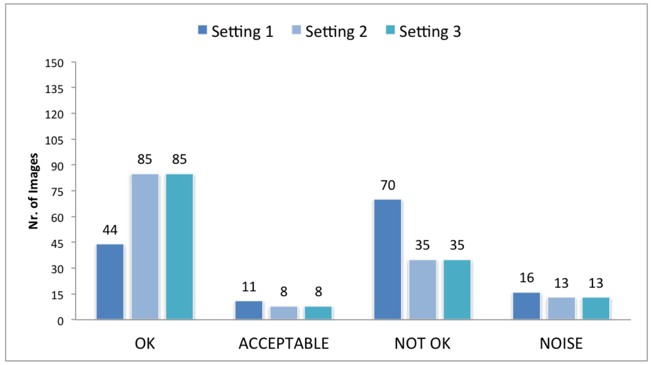
(a) Image size of 256x256 pixels, using kMeans++



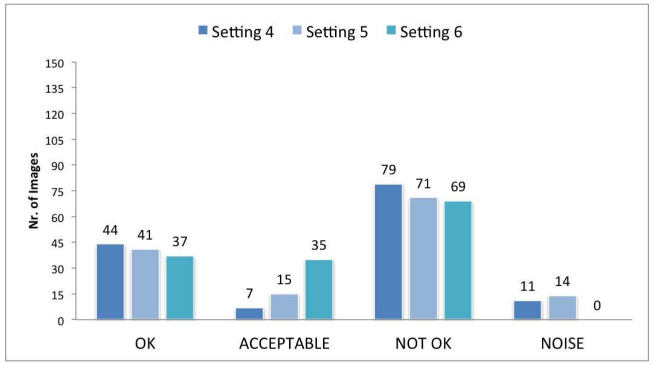
(b) Image size of 128x128 pixels, using kMeans++



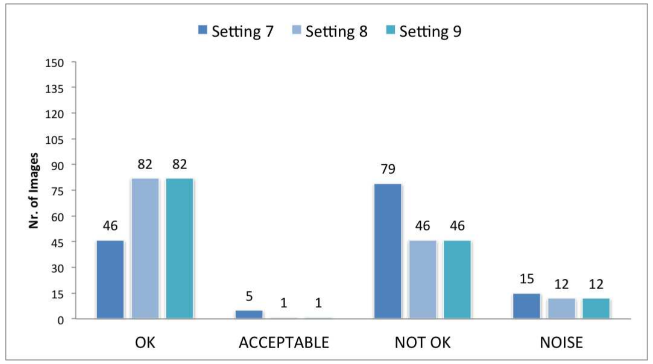
(c) Image size of 64x64 pixels, using kMeans++



(d) Image size of 256x256 pixels, using DBScan

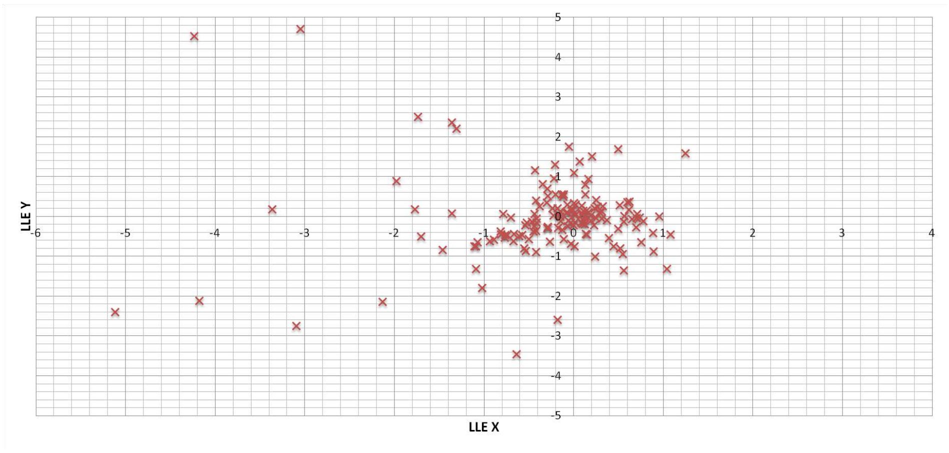


(e) Image size of 128x128 pixels, using DBScan

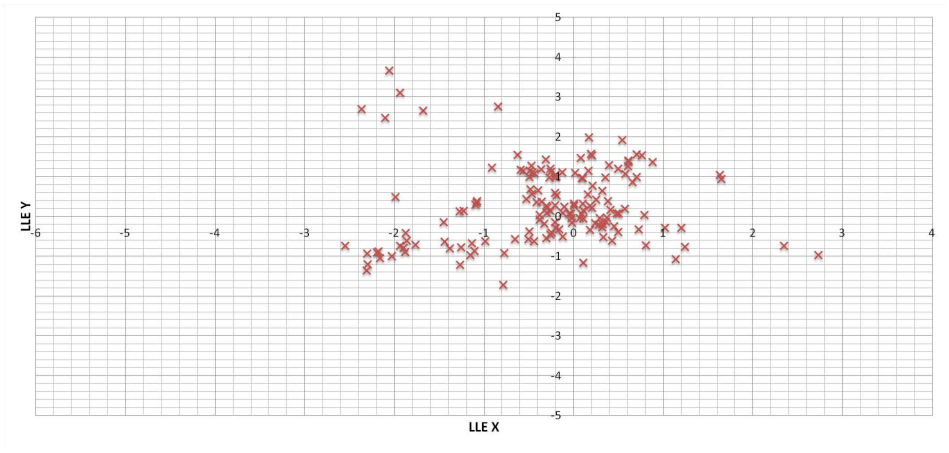


(f) Image size of 64x64 pixels, using DBScan

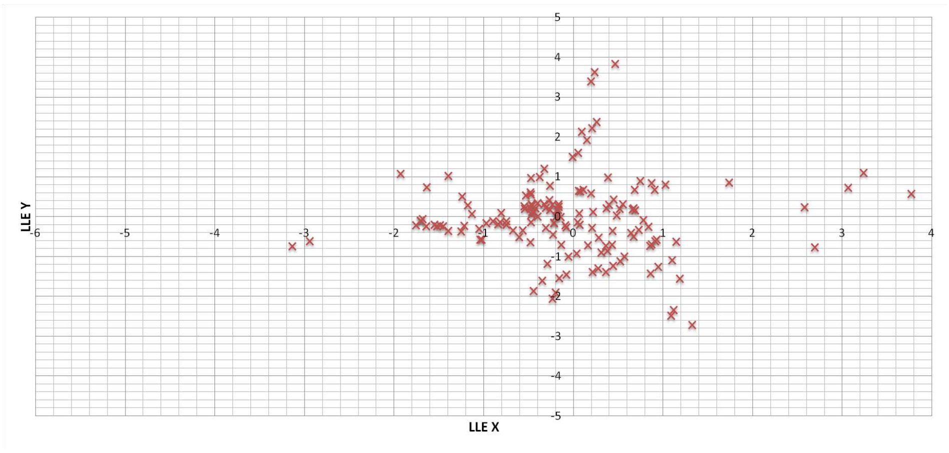
Figure 5.1: The results for the **Stag Beetle** volume using kMeans++ and DBScan clustering in combination with the test settings in table 5.1.



(a) 256x256 20 K



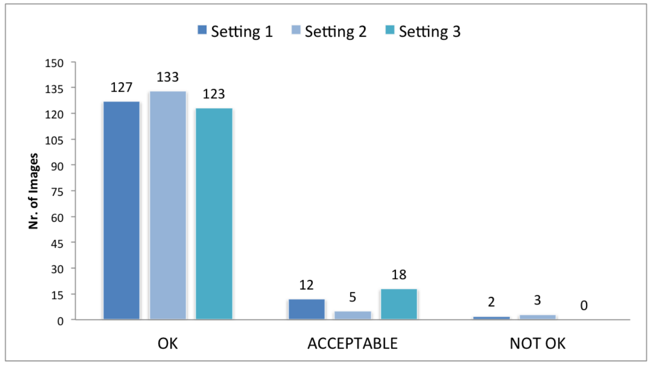
(b) 256x256 12 K



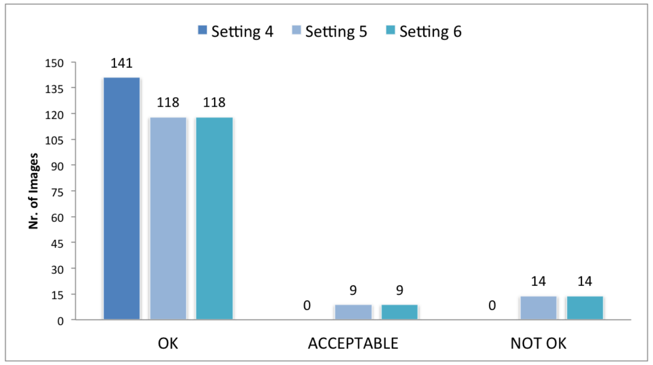
(c) 256x256 5 K

Figure 5.2: Distribution of the *LLE* coordinates of the **Stag Beetle** for the first three test settings. The space between the coordinates increased when decreasing the number of K neighbours.

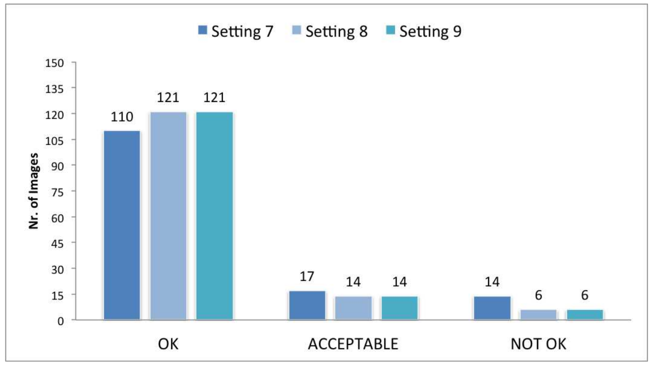
Stented Abdominal Aorta
The results of the test settings using the Stented Abdominal Aorta volume can be seen in figure 5.3. Figures 5.3a to 5.3c show the results for kMeans++ in combination with the test settings in table 5.1. Out of the 150 images, nine were removed as they were equal to other images. For all test settings the number of *OK* images for kMeans++ lay between 110 and 141 images. The best results were achieved for test setting number four in figure 5.3b. A total number of 141 images (100%) were classified as *OK*. Therefore an image size of 128x128 pixels and 20 k neighbours was the best setting for Stented Abdominal Aorta volume to obtain images classified as *OK* for the kMeans++ clustering. Slightly worse results were achieved for the test case with an image size of 64x64 pixels but the differences did not seem to be significant. As we were looking for the best results for the clustering, we would recommend setting number four for this volume in combination with kMeans++.
Figures 5.3d to 5.3f show the results for DBScan. For all the test settings an ϵ was found by the automatic method which led to more than 80% of the images being classified as *OK* when the DBScan algorithm was run. The best results were achieved for setting number four, as for kMeans++. 137 out of 141 were classified as *OK*. As there was no other better setting, we also recommend setting number four for DBScan.



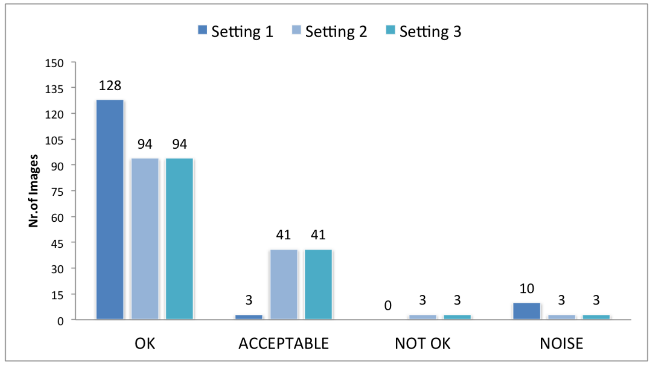
(a) Image size of 256x256 pixels, using kMeans++



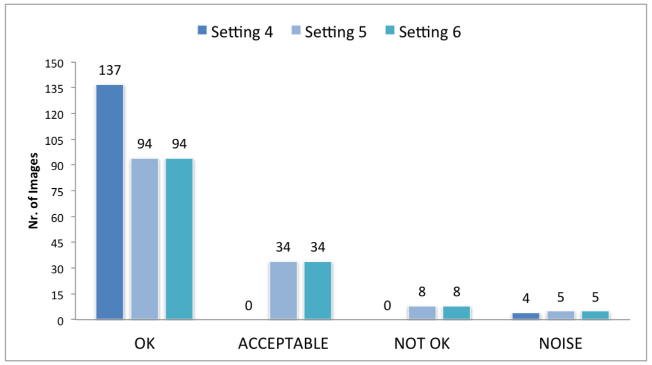
(b) Image size of 128x128 pixels, using kMeans++



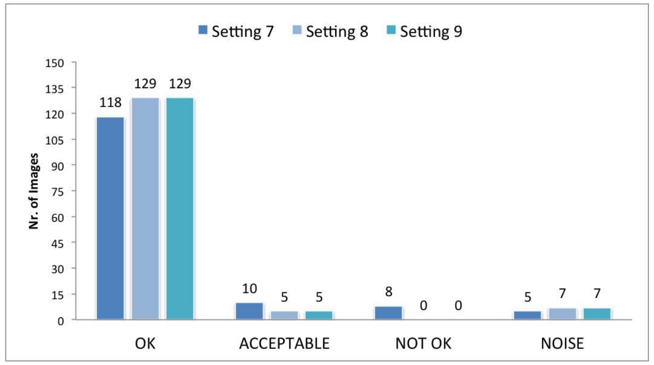
(c) Image size of 64x64 pixels, using kMeans++



(d) Image size of 256x256 pixels, using DBScan



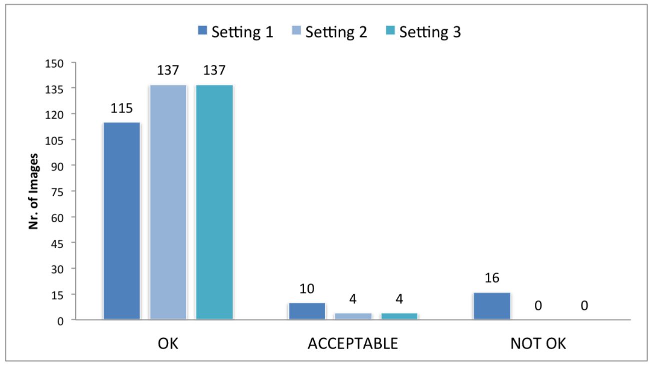
(e) Image size of 128x128 pixels, using DBScan



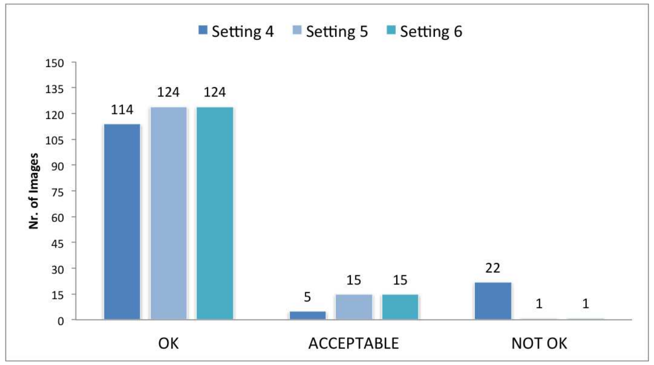
(f) Image size of 64x64 pixels, using DBScan

Figure 5.3: The results for the **Stented Abdominal Aorta** volume using kMeans++ and DBScan clustering in combination with the test settings in table 5.1.

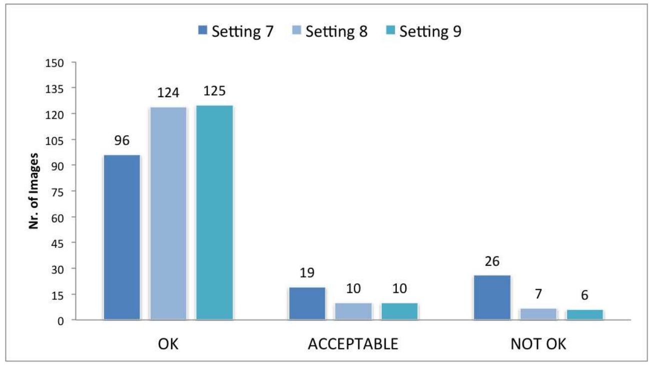
Backpack
The results for the Backpack can be seen in figure 5.4a. The average results for the number of *OK* images were good for the test settings in combination with kMeans++ (see figures 5.4a to 5.4c). We had two settings - number two and three which had 137 out 141 images - where nine were removed because they were equal to other images, thus classified as *OK*.Depending on whether the user is interested in a part of the volume which has fine structures, like the different ties and how they are connected with the fabric, we recommend setting number three for this task. It has a big image size, which should keep the structures and the number of neighbours small, leading to better performance when calculating the *LLE* coordinates. If the performance is important, setting six is recommendable for fast exploration. It took only 35 seconds to render 150 images, calculate their *LLE* coordinates and cluster them.
Figures 5.4d to 5.4f show the results for the DBScan. The number of images classified as *OK* for DBScan were only high for settings number two and three. For the rest of the settings, a non adequate ϵ was found which led to bad results in the classification. This was especially the case for settings number five, six and seven, where only ten to eight images were classified as *OK* and the rest of the images as *NOISE*. Here we had the problem that ϵ was too small, which led to many images not being reachable in ϵ distance and being marked as noise.For this volume we again recommend setting number three for the DBScan cluster algorithm. If performance is important, setting four is recommendable, however the number of images classified as *OK* was only 111 out of 141, which was less than 80% of the images.



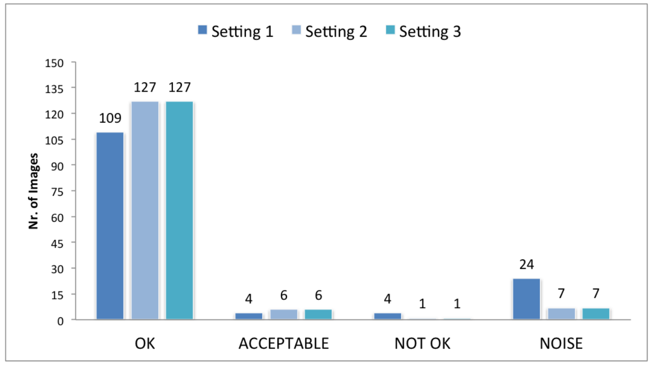
(a) Image size of 256x256 pixels, using kMeans++



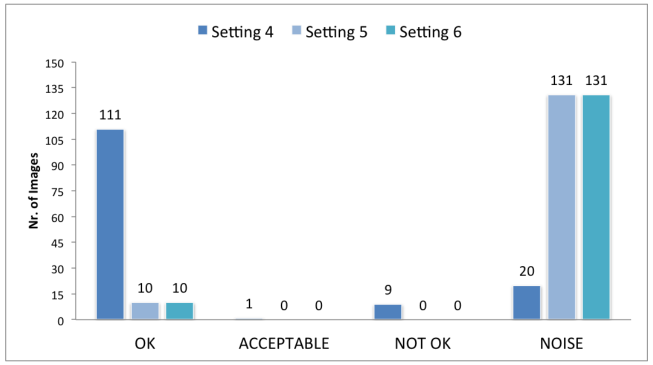
(b) Image size of 128x128 pixels, using kMeans++



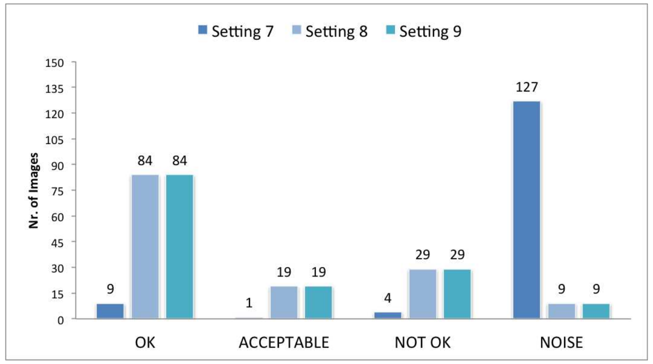
(c) Image size of 64x64 pixels, using kMeans++



(d) Image size of 256x256 pixels, using DBScan



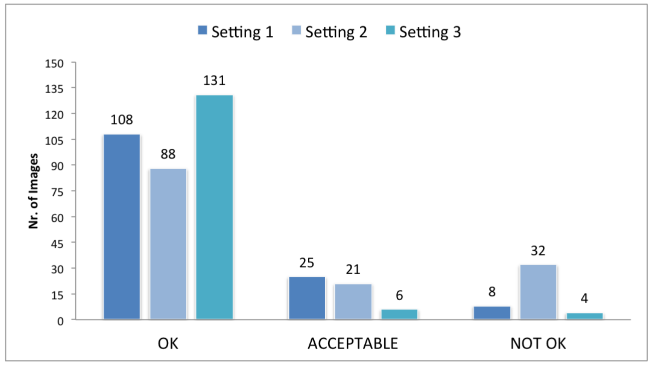
(e) Image size of 128x128 pixels, using DBScan



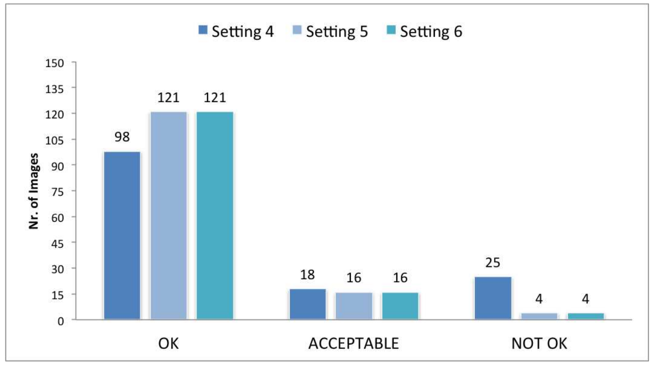
(f) Image size of 64x64 pixels, using DBScan

Figure 5.4: The results for the **Backpack** volume using kMeans++ and DBScan clustering in combination with the test settings in table 5.1.

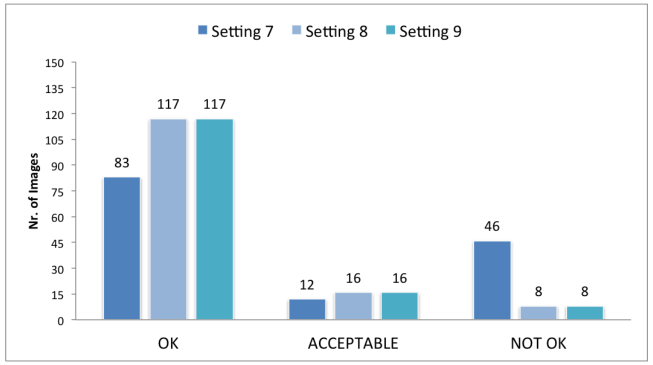
Skewed Head
The bar charts in figure 5.5 show the different results for the Skewed Head volume. Figures 5.5a to 5.5c show the results for kMeans++, where the best result was achieved with test setting number three. A total number of 131 out of 141, where again nine images were removed because they were equal to other images, were classified as *OK*. The six remaining images were *ACCEPTABLE* and only four were *NOT OK*. Therefore this settings is recommendable for this volume in combination with kMeans++. The image size of 256x256 was the biggest used for testing, however the number of neighbours is small which led to an acceptable time when calculating images. For example, test setting three took 55 seconds. This included comparison of the images, calculation of their *LLE* coordinates and clustering. For test setting number four, these same steps took 40 seconds and resulted in a smaller image size but a higher number of neighbours.Therefore, if performance is important, setting four is recommended.
Figures 5.5d to 5.5f show the results for DBScan where for setting one to three and seven to nine the automatic method for selecting ϵ for DBScan found a suitable ϵ which led to 90% of the images being classified as *OK*. Only for settings four to six were under 100 images classified as *OK*. The problem was again the too small chosen ϵ. For the Skewed Head, settings two and three are recommended for further testing in combination with the DBScan clustering algorithm. Both had 128 images classified as *OK*; three were *NOT OK* and only one a *NOISE* image. If taking performance into account, setting number three was optimal. The difference in execution time between setting two and three was only two seconds.



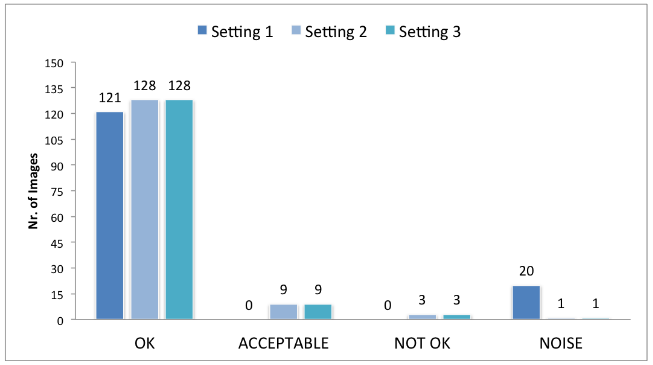
(a) Image size of 256x256 pixels, using kMeans++



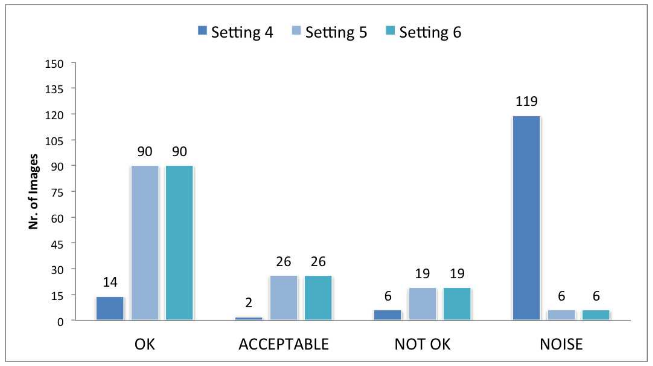
(b) Image size of 128x128 pixels, using kMeans++



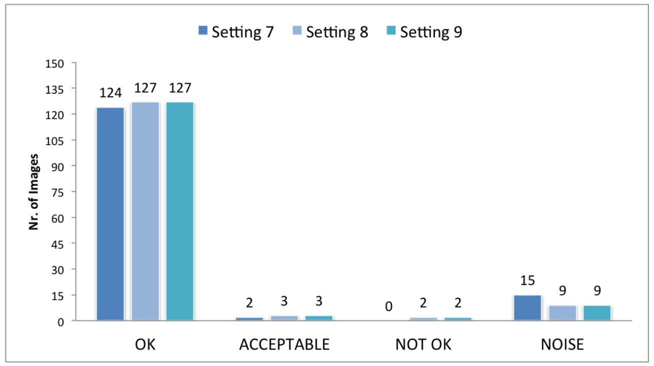
(c) Image size of 64x64 pixels, using kMeans++



(d) Image size of 256x256 pixels, using DBScan



(e) Image size of 128x128 pixels, using DBScan



(f) Image size of 64x64 pixels, using DBScan

Figure 5.5: The results for the **Skewed Head** volume using kMeans++ and DBScan clustering in combination with the test settings in table 5.1.

5.3.1 Settings for Further Testing and Additional Remarks

We checked more than 11,000 images for four different volumes using nine different settings and the two different clustering algorithms. For more than 85% of all the tests we ran we achieved more than 90% of the images being classified as *OK*. A summary of the recommended settings can be found in table 5.2.

Volume	Clustering	Image Size(pixels)	*K*
Stag Beetle	kMeans++	256x256	5
	kMeans++ (better performance)	128x128	20
	DBScan	256x256	5
Stented Abdominal Aorta	kMeans++	128x128	20
	DBScan	128x128	20
Skewed Head	kMeans++	256x256	5
	kMeans++ (better performance)	128x128	20
	DBScan	256x256	5
Backpack	kMeans++	256x256	5
	kMeans++ (better performance)	128x128	5
	DBScan	256x256	20
	DBScan (better performance)	128x128	12 or 5

Table 5.2: The *LLE* settings recommended for each volume and used for testing in section 5.4.

For all the tested volumes we reached similar results. For kMeans++ the best stetting was - except for the Stented Abdominal Aorta - number three with an image size of 256x256 pixels and 20 neighbours. This setting could also be used when using ParSAd in combination with other, not tested volume files. The results for the tested settings in combination with DBScan were not that similar. The best results were achieved for settings number one and six. If using DBScan in combination with another not tested volume file, it is recommended to take a setting with a bigger image size, like 256x256 or 128x128, and a small number of neighbours, like 12 or five. For this size the scattering of the *LLE* makes it easier for our automatic method for ϵ.

The results of the different setting tests with the different volumes indicated that our implementation for *LLE* is correct. Similar images lay close together in the 2D *LLE* coordinate system. The test showed that for most of the tested settings, our method for automatically detecting ϵ found an adequate value for ϵ, which led to images being assigned to the correct clusters instead of having one big cluster which almost all images belonged too. However there were also cases, like for setting number seven for the Backpack volume, where the method failed. We observed exactly the behaviour mentioned before. For this setting we obtained one big cluster which more than 120 images belonged to, but only eight were similar to the cluster centre image. Therefore the method for automatically detecting ϵ still needs improvement. If the user chooses this first big cluster for further exploration than good results can still be achieved when in a second run an adequate ϵ is found.

In comparing the performance between kMeans++ and DBScan, we observed that if ϵ was adequately chosen, both algorithms provided good results by having similar images assigned to the correct clusters. For kMeans++ we saw that most of the images classified as *NOT OK* lay near to or on the border of the assigned cluster which is a known problem for the algorithm. As opposed to kMeans++ the number of images classified as NOISE or NOT OK for DBScan were mostly dependent on ϵ.

5.4 Testing ParSAd - Reproducing Volume Visualisations with Manually Set Parameter Values

To test if ParSAd provides a way to achieve visualisations similar to such created by the manual setting of parameters, and to see if it speeds up this process, we came up with the following procedure:

We created four different visualisations by setting the necessary parameters manually and tried to recreate this visualisation with ParSAd as good as possible. For each volume we used all 14 different parameters. To compare the results of manual parameter setting and ParSAd, we wanted to count the steps necessary to create the visualisations. We defined a step as a change to the visualisation. For manual parameter setting, each changing of a parameter would directly lead to a change in the visualisation and therefore count as one step. As we were manually creating the reference visualisations on our own, it would not have been sensible to count each step made. Therefore we decided to take the absolute minimum of necessary steps to create these visualisations, which equals the number of parameters that have to be set. Ideally, each parameter has to be set only one time. In a user test under more realistic circumstances, more steps would be needed for sure. Such a user test is described in detail in section 5.5. When working with ParSAd, we counted the steps needed to achieve the final visualisation. A step is defined as a change of the visualisation which occurs when images are selected out of the nine or less presented cluster centres, and/or the right arrow button for further exploration or refinement is pressed.Finally we calculated the SSIM index between the image of the visualisation achieved with ParSAd and the manually created visualisation to decide if a visualisation of sufficient similarity had been achieved. As already mentioned, the SSIM index ranges between 0.0 and 1.0 where a value of 1.0 means total similarity of two images and 0.0 no similarity.
For every volume, except the Skull, we tried to reproduce the manually defined image by using first kMeans++ and second DBScan. For both algorithms we used the recommended *LLE* settings from table 5.2. Further we wanted to test whether we could reproduce a visualisation with a more difficult transfer-function. This was done for the Skull volume by using the transfer-function high-level parameter in combination with the *All* high-level parameter.
We stopped selecting images in ParSAd when hardly any visual differences between the manually created and the visualisation with ParSAd were noticeable. Additionally, as already mentioned, we measured the SSIM index - which we wanted to have between 0.995 and 1.0 - as 0.05. This was found to be a tolerable value in section 5.3. For every visualisation, we reproduced and saved the selected images, and displayed them in different figures after every test case (except

for the DBScan test case for the Stended Abdominal Aorta). We made one figure (figure 5.10) which shows all the displayed centres including the selected images. We did this to keep the thesis readable. For all the other test cases, only the selected images and high-level-parameter(s) are shown.

5.4.1 Reproducing One Visualisation of the Stented Abdominal Aorta

One of the main features of the Stented Abdominal Aorta volume is, as the name indicates, the stent. The manually defined visualisation concentrated on this feature. We tried to reproduce the same visualisation with both clustering algorithms. For kMeans++, setting number four was used, and for DBScan, setting three in table 5.1 was used for *LLE*. As it is the main idea of ParSAd to vary many different parameters at once and to not have the user care about any parameters, we wanted to show that it is possible for ParSAd to handle a large set of low-level parameters and help to find the desired setting values for them in less steps than setting them manually. Figure 5.6 shows the final manually created visualisation we wanted to reproduce with ParSAd.

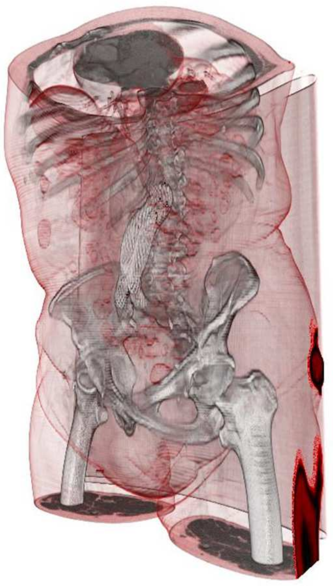
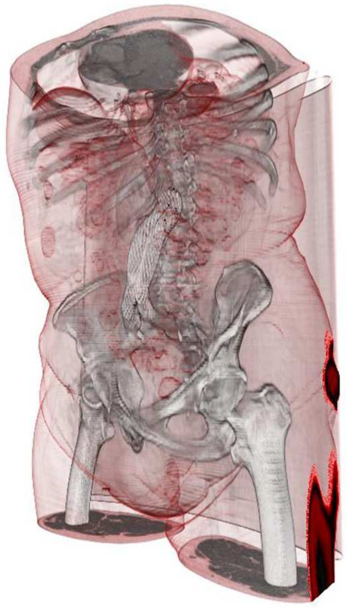


Figure 5.6: Reference visualisation created by setting the parameter values manually.

kMeans++
To reproduce the manually created visualisation for the Stented Abdominal Aorta, a total number of eight steps was necessary. The only high-level parameter used was *All*. Four hundred and fifty images were created for this test case. The first image in the first step had an SSIM index of 0.8975 compared to the last image in the last step having a SSIM index of 0.9935. Figure 5.7 shows the comparison of the manually, figure 5.7a created visualisation to the one created with ParSAd in combination with kMeans++, figure 5.7b. Figure 5.8 shows the different steps

of reproducing the visualisation. We always selected images which looked similar to the final visualisation or had parts in common with it. For example, with an SSIM index of 0.7982, image number two in step two looks very different from the manually made visualisation, however, the bones had the correct shading, therefore this image was selected. The SSIM index increased constantly during exploration and refining, which showed that we were getting closer to the manually created visualisation with every step made. The parameter values of the manually created visualisation where similar to those found by ParSAd (see figure 5.9).


(a) Visualisation of the Stented Abdominal Aorta by setting the parameter values manually

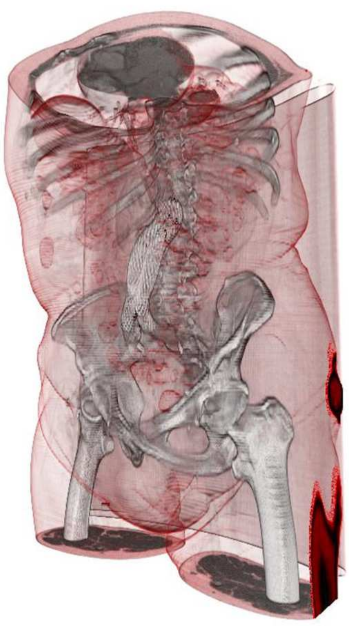

(b) Visualisation of the Stented Abdominal Aorta created by using ParSAd with kMeans++

Figure 5.7: Comparison of visualisations of the Stented Abdominal Aorta, done by setting the parameter values manually and by using ParSAd with kMeans++.

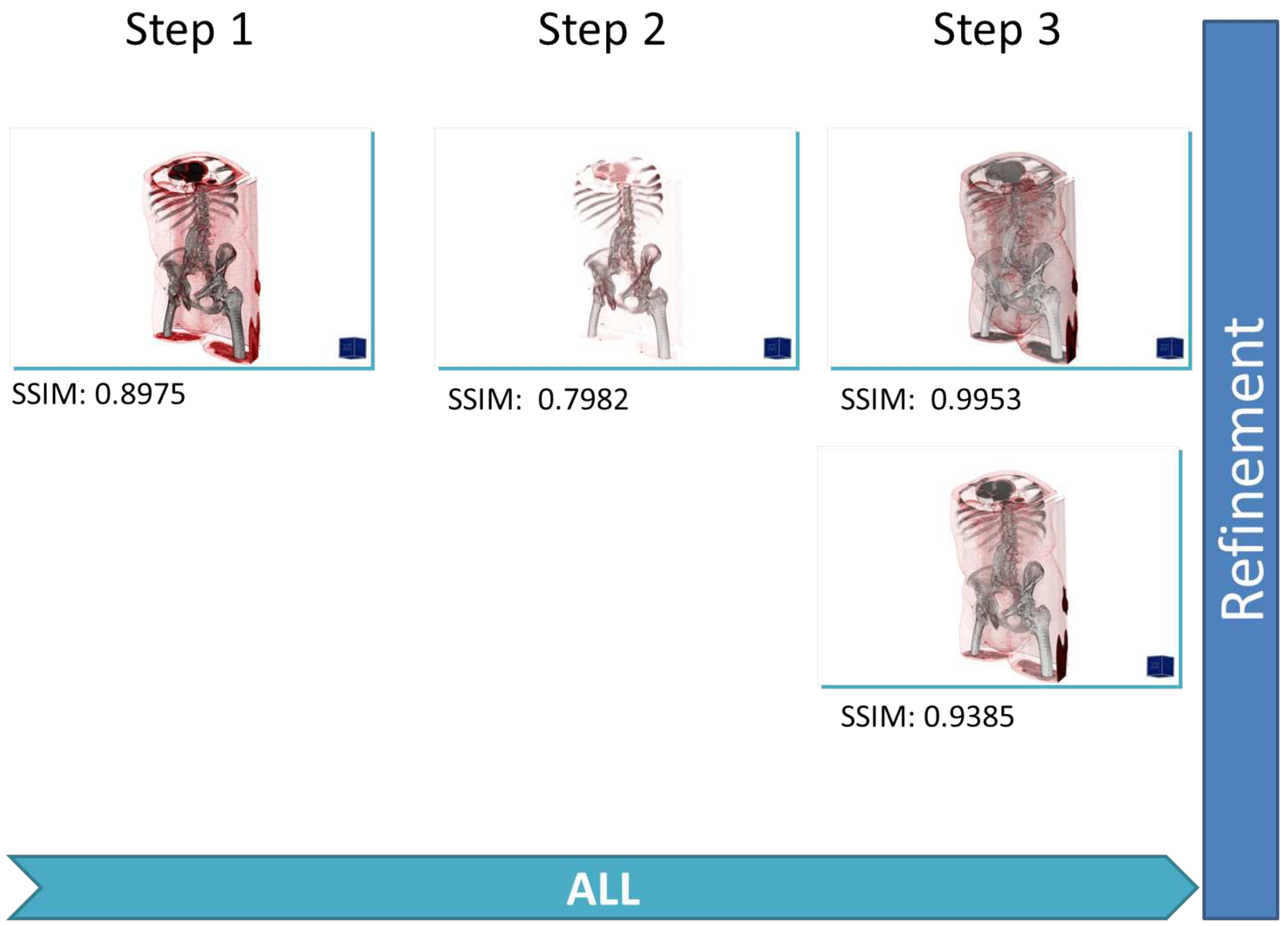


Figure 5.8: Steps one to seven for reproducing the manually created visualisation of the Stented Abdominal Aorta by using ParSAd in combination with kMeans++.

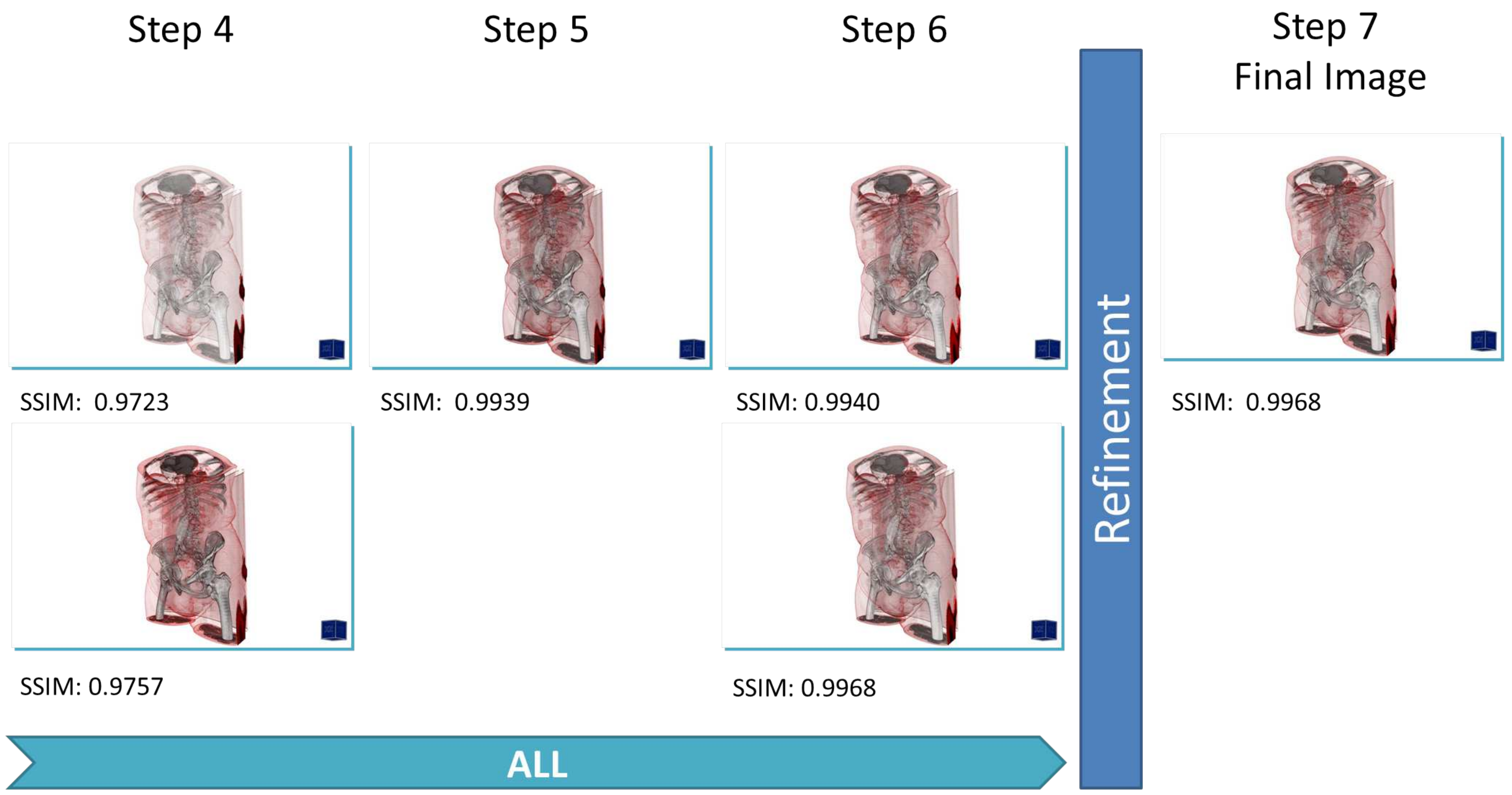
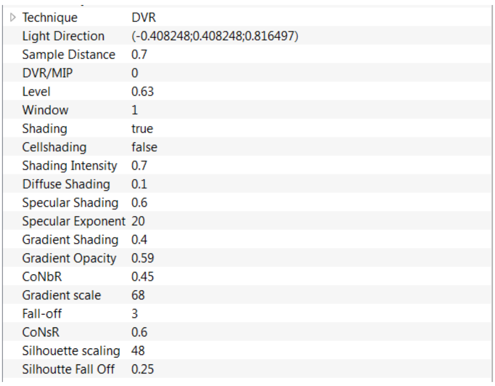
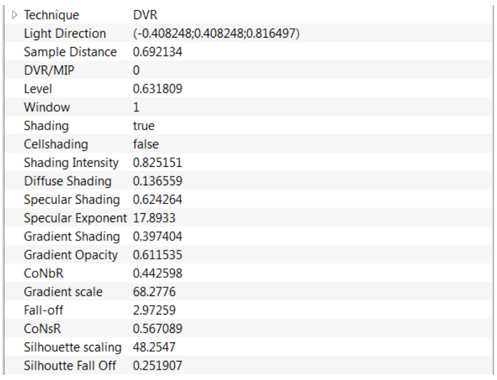


Figure 5.8 (continued)



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.7
DVR/MIP	0
Level	0.63
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.7
Diffuse Shading	0.1
Specular Shading	0.6
Specular Exponent	20
Gradient Shading	0.4
Gradient Opacity	0.59
CoNbR	0.45
Gradient scale	68
Fall-off	3
CoNsR	0.6
Silhouette scaling	48
Silhoutte Fall Off	0.25

(a) Parameter values of the manually created visualisation



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.692134
DVR/MIP	0
Level	0.631809
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.825151
Diffuse Shading	0.136559
Specular Shading	0.624264
Specular Exponent	17.8933
Gradient Shading	0.397404
Gradient Opacity	0.611535
CoNbR	0.442598
Gradient scale	68.2776
Fall-off	2.97259
CoNsR	0.567089
Silhouette scaling	48.2547
Silhoutte Fall Off	0.251907

(b) Parameter values of the visualisation created by ParSAd with kMeans++

Figure 5.9: Comparison of the parameter values between the manually created visualisation and ParSAd.

DBScan

The results when using the DBScan algorithm were slightly better. Only seven steps were taken to reproduce the visualisation. Again the SSIM index increased during the exploration and refinement of the images. However, the SSIM index of the final image, which was 0.9827, was not as good as for kMeans++ but still acceptable. For this test case 600 images were produced and again only the high-level parameter *All* was used. Figure 5.10 shows the process of producing the final result with ParSAd and the increase of the SSIM index after each step. Additionally we also made images of the displayed centres to prove whether the cluster centres presented to the user became more similar with each step. This can be seen in figure 5.10 above the selected images. After first clicking the *All* high-level parameter, we received seven totally different centres, which was good for the start because at the beginning the direction in which the user wants to change the visualisation is not known. After selecting one centre (see step one in figure 5.10) out of the seven pictures, the next displayed centres looked similar to the chosen image (step two). As there were only three images assigned to the selected centre in step one, we had to create new images for the next step (step number three). But before creating them we chose two images out of the three displayed centres. Both were similar to the final manually created visualisation we wanted to achieve. The centres displayed for step three looked similar to the before chosen images. For step number three only one image is best. When the images for step three had been created, a very bad ϵ value was received. One hundred and one images were assigned to the chosen image's cluster. Therefore we obtained six very different looking centres. But one centre looked again similar to the visualisation we wanted to reproduce. For the last three steps the displayed centres became more and more similar to each other and to the manually created visualisation (see figure 5.10 steps five to seven). The manual visualisation, figure 5.11a compared to the one made by using ParSAd in combination with DBScan, figure 5.11b, can be seen in figure 5.11. The parameter values of the manually created visualisation where similar for kMeans++ to those found by ParSAd (see figure 5.12).

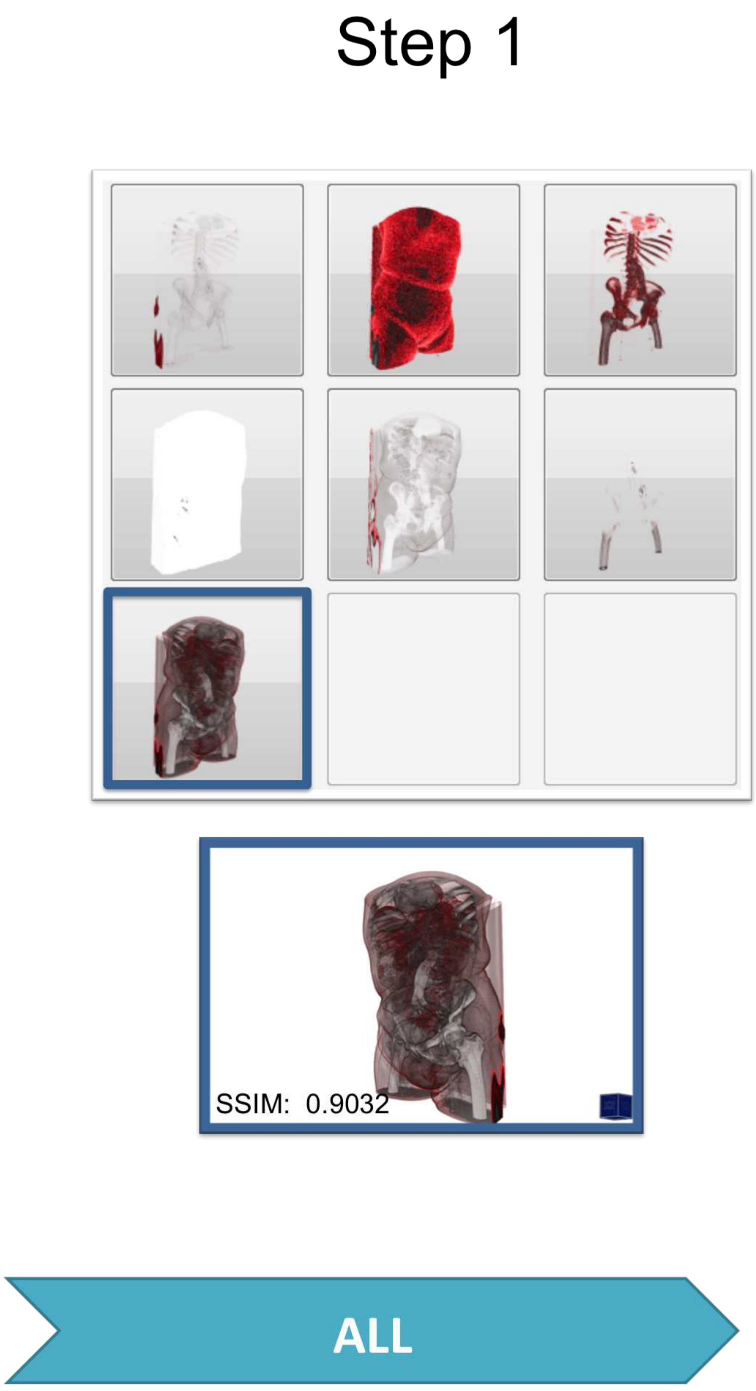
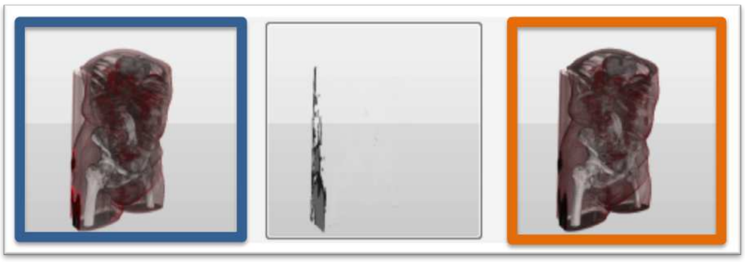
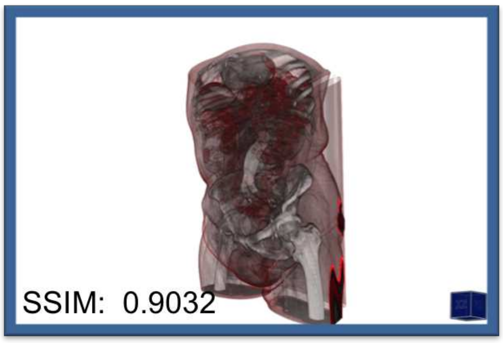
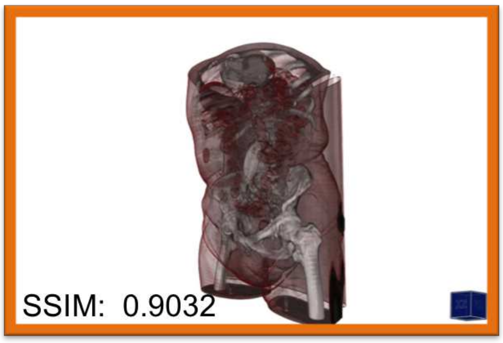


Figure 5.10: The seven steps for reproducing the manually created visualisation of the Stented Abdominal Aorta by using ParSAd in combination with DBScan.

Step 2







ALL

Figure 5.10 (continued)

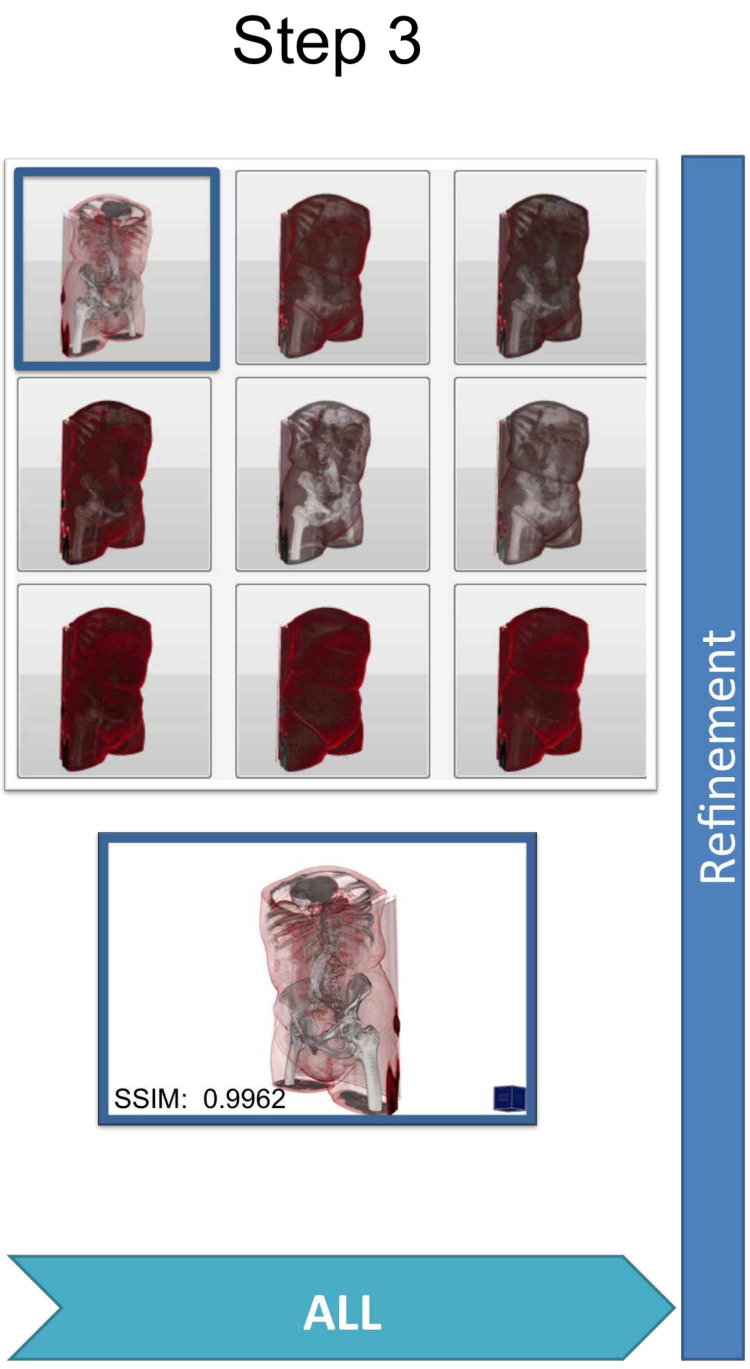


Figure 5.10 (continued)

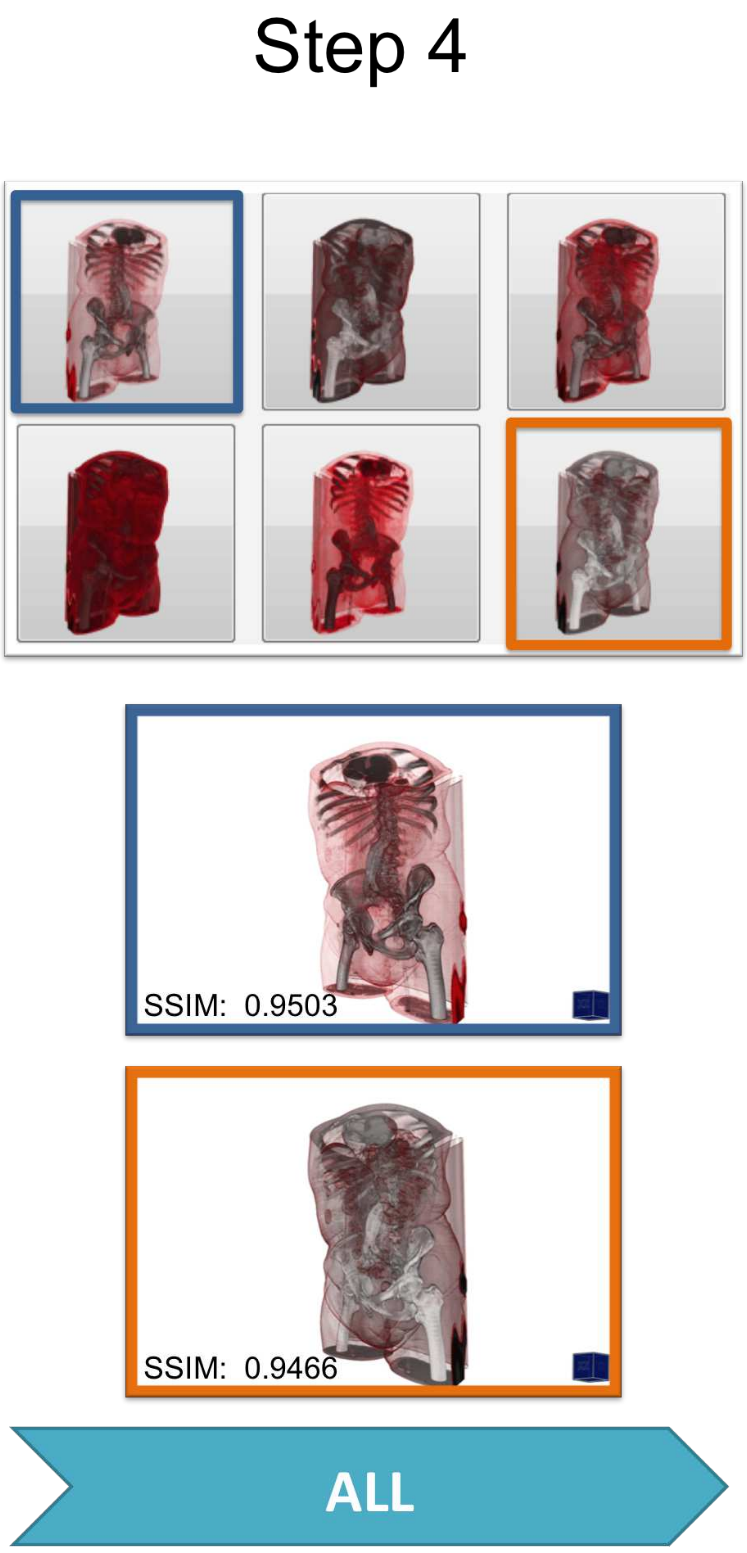
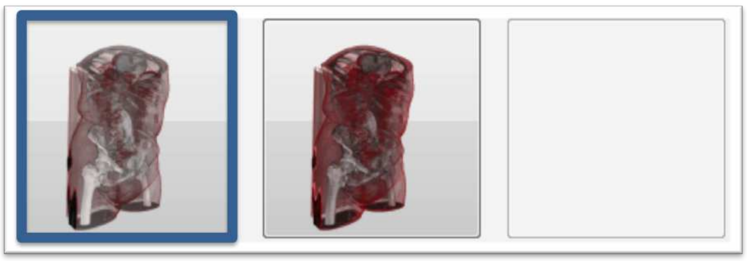


Figure 5.10 (continued)

Step 5



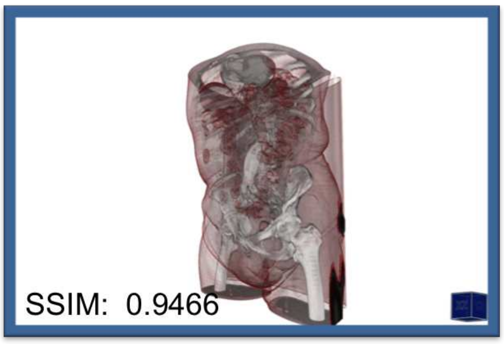
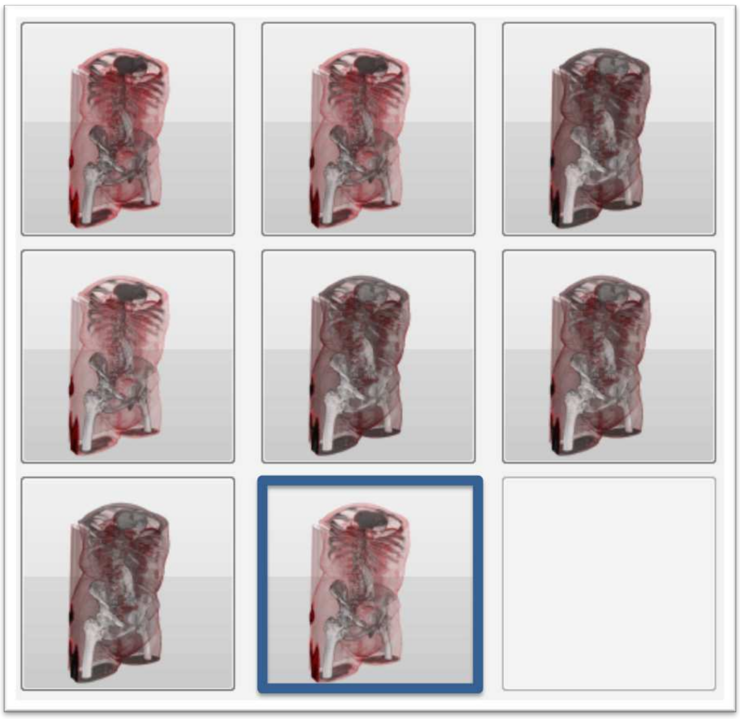


Figure 5.10 (continued)

Step 6



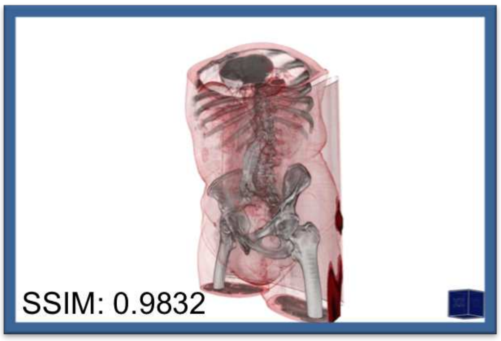


Figure 5.10 (continued)

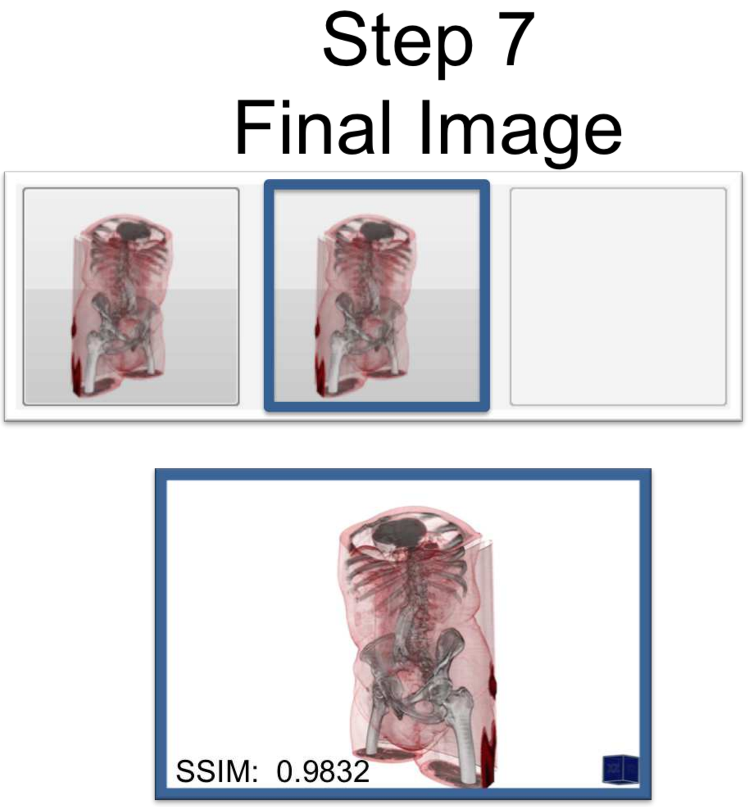
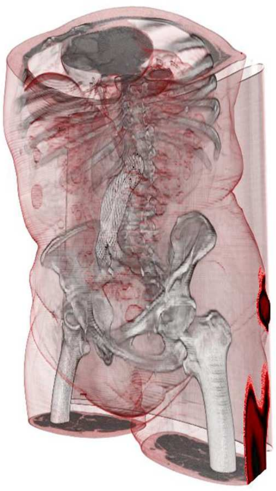


Figure 5.10 (continued)


(a) Visualisation of the Stented Abdominal Aorta by setting the parameter values manually

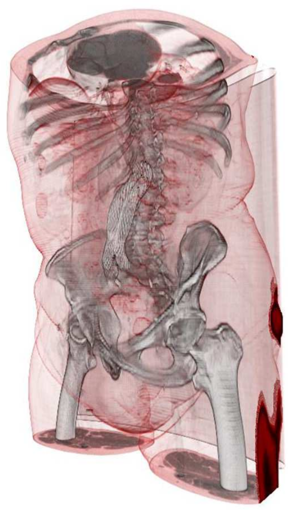
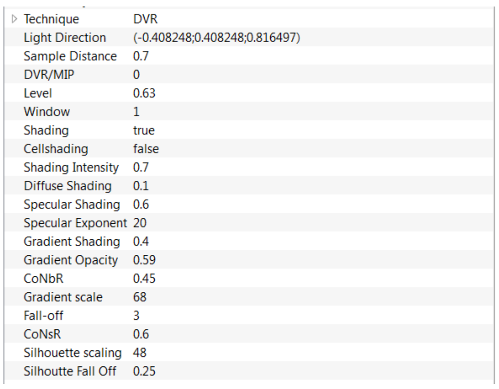
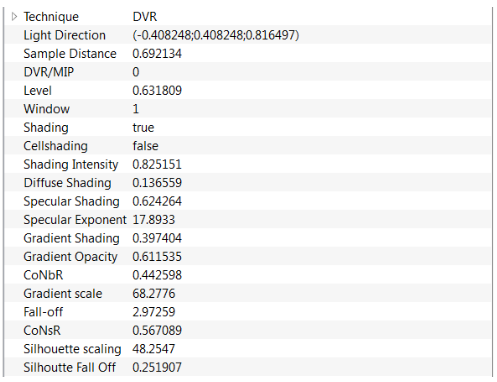

(b) Visualisation of the Stented Abdominal Aorta created by using ParSAd with DBScan

Figure 5.11: Comparison of visualisations of the Stented Abdominal Aorta, done by setting the parameter values manually and by using ParSAd with DBScan.



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.7
DVR/MIP	0
Level	0.63
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.7
Diffuse Shading	0.1
Specular Shading	0.6
Specular Exponent	20
Gradient Shading	0.4
Gradient Opacity	0.59
CoNbR	0.45
Gradient scale	68
Fall-off	3
CoNsR	0.6
Silhouette scaling	48
Silhoutte Fall Off	0.25

(a) Parameter values of the manually created visualisation



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.692134
DVR/MIP	0
Level	0.631809
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.825151
Diffuse Shading	0.136559
Specular Shading	0.624264
Specular Exponent	17.8933
Gradient Shading	0.397404
Gradient Opacity	0.611535
CoNbR	0.442598
Gradient scale	68.2776
Fall-off	2.97259
CoNsR	0.567089
Silhouette scaling	48.2547
Silhoutte Fall Off	0.251907

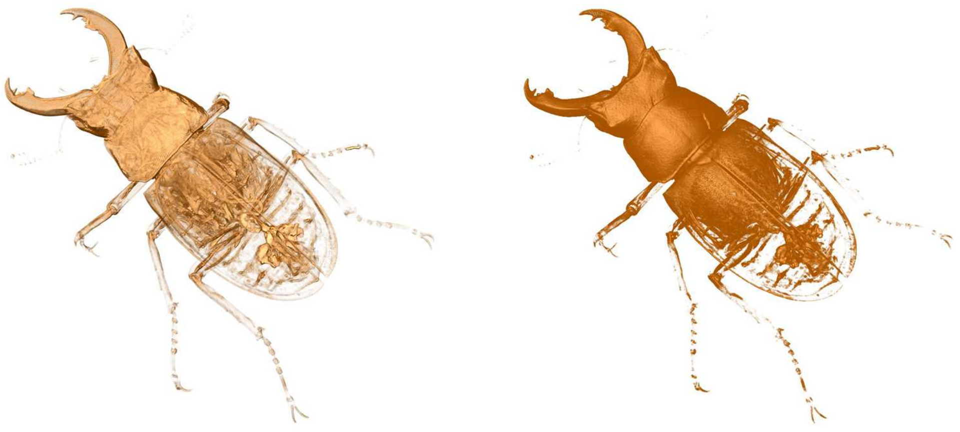
(b) Parameter values of the visualisation created by ParSAd with DBScan

Figure 5.12: Comparison of the parameter values between the manually created visualisation and ParSAd.

With these two test cases for the Stented Abdominal Aorta we showed that it is possible to reproduce the manually created visualisation with a lower number of steps. Further we showed that it is possible with ParSAd by using only the high-level parameter *All* to reproduce the visualisation either by using kMeans++ or DBScan. By capturing the displayed centres, the images displayed to the user became similar after every step. However we had the problem of choosing a wrong ϵ value for DBScan, which led to one big cluster and slightly different centres being displayed after selecting this big cluster.

5.4.2 Reproducing Two Visualisations of the Stag Beetle

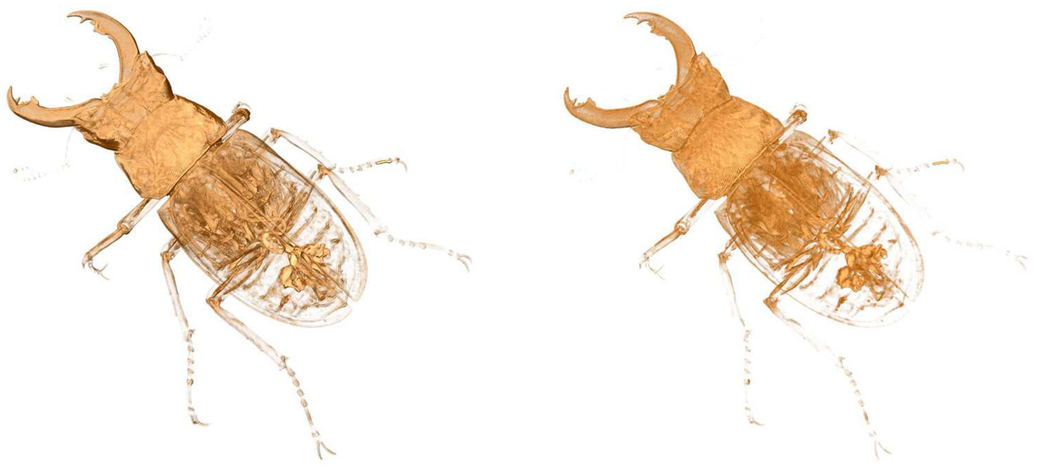
The focus of the manually crested image was on the gut and the inner tubes - which supply the organism with air - of the Stag Beetle. Figure 5.13 shows the two final manually created visualisations. First we tried to reproduce the image by using kMeans++ (figure 5.13a). For *LLE* we used setting number three in table 5.1. Second we did the same by using the DBScan algorithm using *LLE* setting number three and a slightly different manually produced visualisation of the Beetle (see figure 5.13b) to check whether a setting with a high specular exponent value could be reproduced by ParSAd.



(a) Reference visualisation for kMeans++ (b) Reference visualisation for DBScan

Figure 5.13: Manually created visualisations which were to be reproduced by using ParSAd.

kMeans++
With ParSAd it was possible to reproduce a quite similar visualisation to the manual one by exploring 450 images within seven steps, which is seven steps less than setting the parameters manually. The values of the parameters were similar as well, which can be seen figure 5.16. To reproduce the image by using ParSAd, we started with the high-level parameter *All*. It varies the values of all 14 parameters at once. By selecting *All*, 150 images were produced and explored in two steps. The image in step two was taken as input for the next high-level parameter which was *Sketchiness*. Again two steps were necessary to increase the similarity to the manually produced image. After four steps we had a SSIM index of 0.9642, which was a very good value if taking into account that one means two images are similar. With the *Detail* high-level parameter we tried to increase the details of the tubes and the gut. However, after selecting the first centre image, which was step number six, the SSIM index dropped to 0.9497. In step seven, the final step, an image that looked very similar to the manual one was selected. The final SSIM index was 0.97. The manual visualisation, figure 5.14a, compared to the one made by using ParSAd, figure 5.14b, in combination with kMeans++ can be seen in figure 5.14. Figure 5.15 shows the process of producing the final result with ParSAd and shows the increase of the similarity values for each step.



(a) Visualisation of the Stag Beetle by setting the parameter values manually

(b) Visualisation of the Stag Beetle created by using ParSAd with DBScan

Figure 5.14: Comparison of visualisations of the Stag Beetle, done by setting the parameter values manually and by using ParSAd with kMeans++.

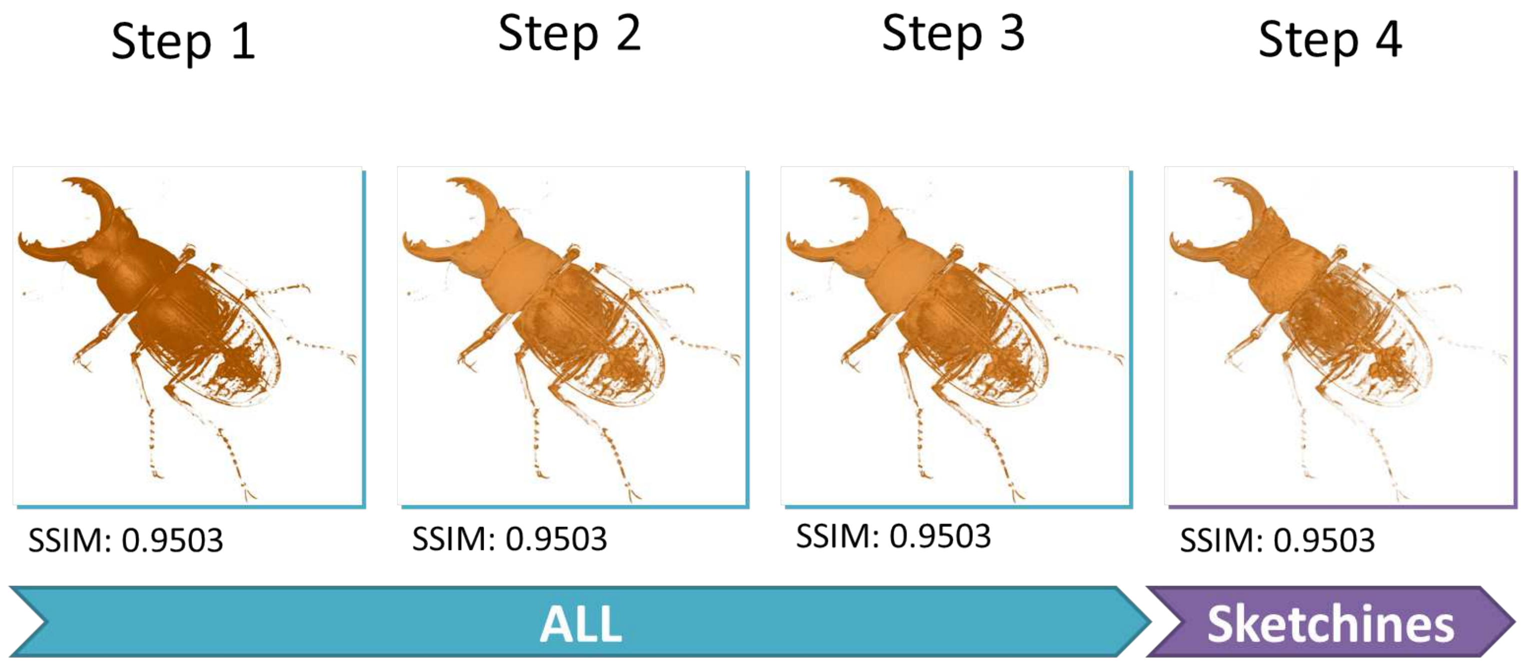


Figure 5.15: Steps one to seven for reproducing the manually created visualisation of the Stag Beetle by using ParSAd in combination with kMeans++.

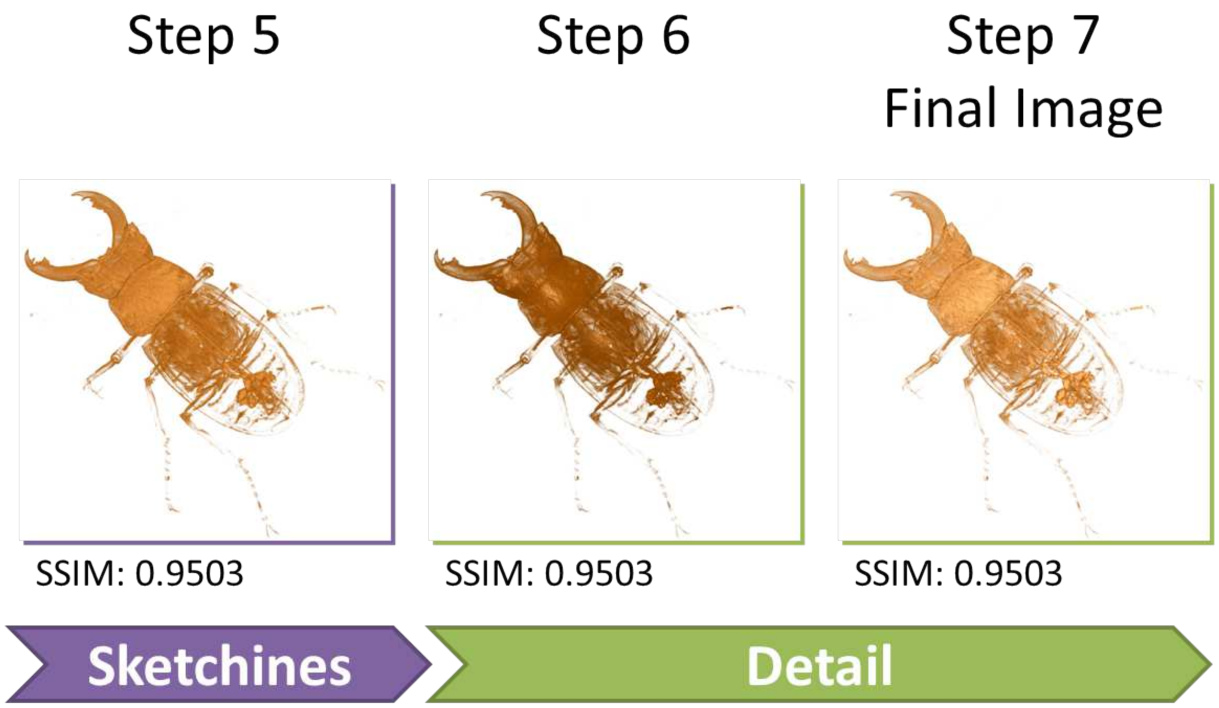
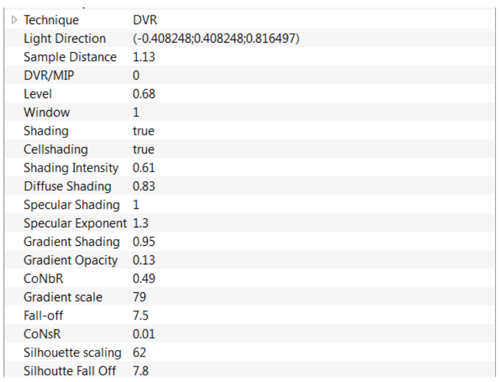
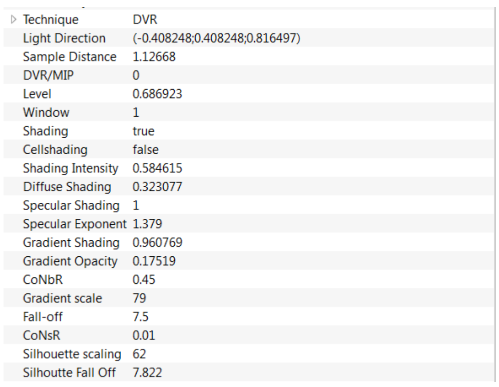


Figure 5.15 (continued)



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	1.13
DVR/MIP	0
Level	0.68
Window	1
Shading	true
Cellshading	true
Shading Intensity	0.61
Diffuse Shading	0.83
Specular Shading	1
Specular Exponent	1.3
Gradient Shading	0.95
Gradient Opacity	0.13
CoNbR	0.49
Gradient scale	79
Fall-off	7.5
CoNsR	0.01
Silhouette scaling	62
Silhoutte Fall Off	7.8

(a) Parameter values of the manually created visualisation



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	1.12668
DVR/MIP	0
Level	0.686923
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.584615
Diffuse Shading	0.323077
Specular Shading	1
Specular Exponent	1.379
Gradient Shading	0.960769
Gradient Opacity	0.17519
CoNbR	0.45
Gradient scale	79
Fall-off	7.5
CoNsR	0.01
Silhouette scaling	62
Silhoutte Fall Off	7.822

(b) Parameter values of the visualisation created by ParSAd with kMeans++

Figure 5.16: Comparison of the parameter values between the manually created visualisation and ParSAd.

DBScan

The final image having a SSIM index of 0.9999 was reached within 12 steps by using ParSAd compared to 14 when creating it manually. Still two steps less then defining 14 parameters manually. It was possible to reproduce a visualisation having a high specular exponent. However we saw that for this test case low-level parameter knowledge was still necessary to get the result more quickly otherwise this could lead to a long trial and error process for the user when trying different high-level parameters to reproduce the final image. Figure 5.17 shows the process of producing the final result with ParSAd and also the increase of the similarity values after each step. The similarity of the final image, figure 5.18a to the manually defined one, figure 5.18b, was almost 1. Compared to kMeans++ more steps were necessary to achieve this. The values of the parameter were very similar as well, figure 5.19

As for kMeans++ we started with the high-level parameter *All*. The first image in step one had a SSIM index of 0.8773. The image already showed all necessary parts. Therefore we only wanted to increase the detail and the sketchiness of the image to make it look similar to the manually produced one. Therefore the high-level parameter *Detail* was used to produce the images for the second step where one was selected for further exploration of the assigned images to the cluster. The selected one had the same SSIM index of 0.8773 as the image in step one. In step three, three different images were selected. The SSIM index of the three selected images was still under 0.9. Therefore we tried to increase the sketchiness of the visualisation which did not change within two steps. This was because of the very low values for the gradient opacity low-level parameter set by the *Detail* high-level parameter. To get a better result we tried using again the *Detail* parameter to see more of the gut and tubes. After generating the images we saw the influence of the *Detail* high-level parameter much better, as the gradient opacity low-level parameter had a higher value. After selecting and refining the detail two times, we got our final result within eight steps where the SSIM index increased slightly for every step.

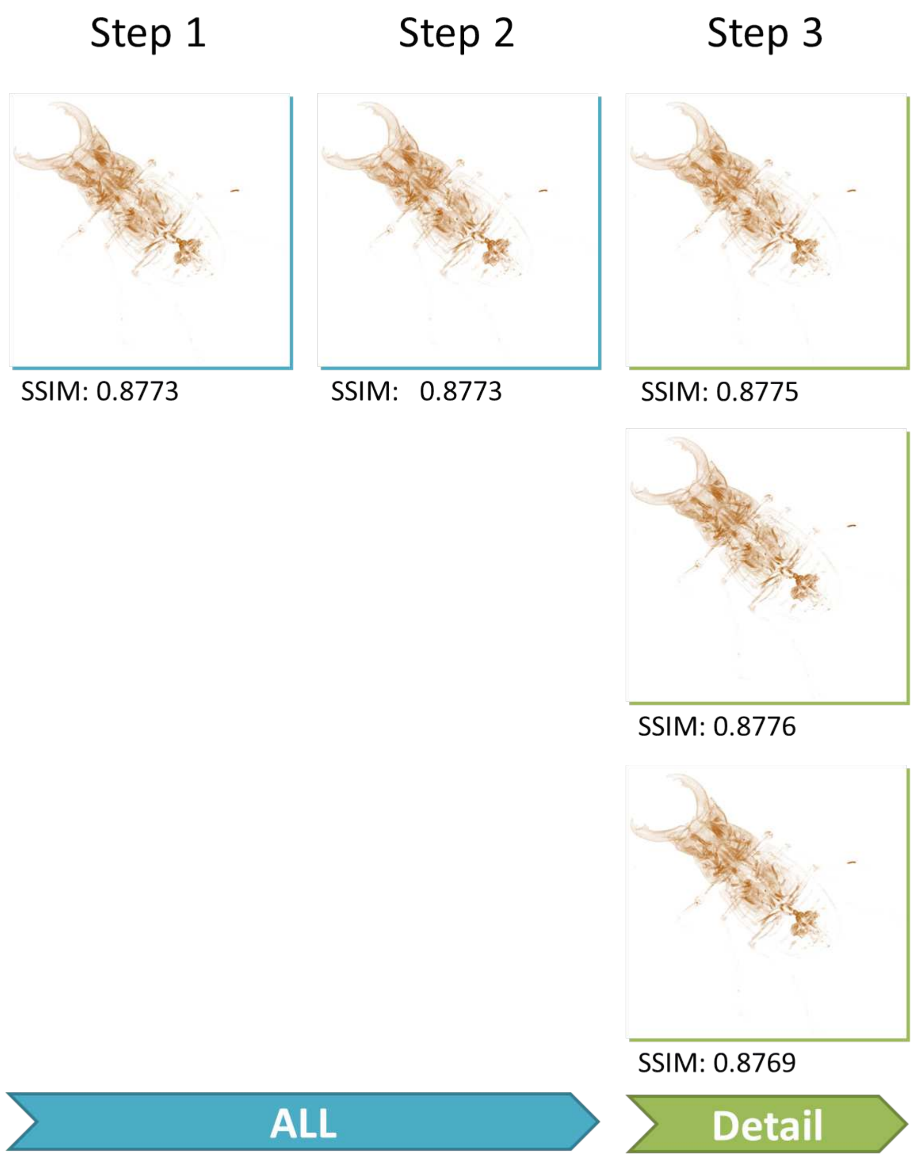


Figure 5.17: Steps one to ten for reproducing the manually created visualisation of the Stag Beetle by using ParSAd in combination with kMeans++.

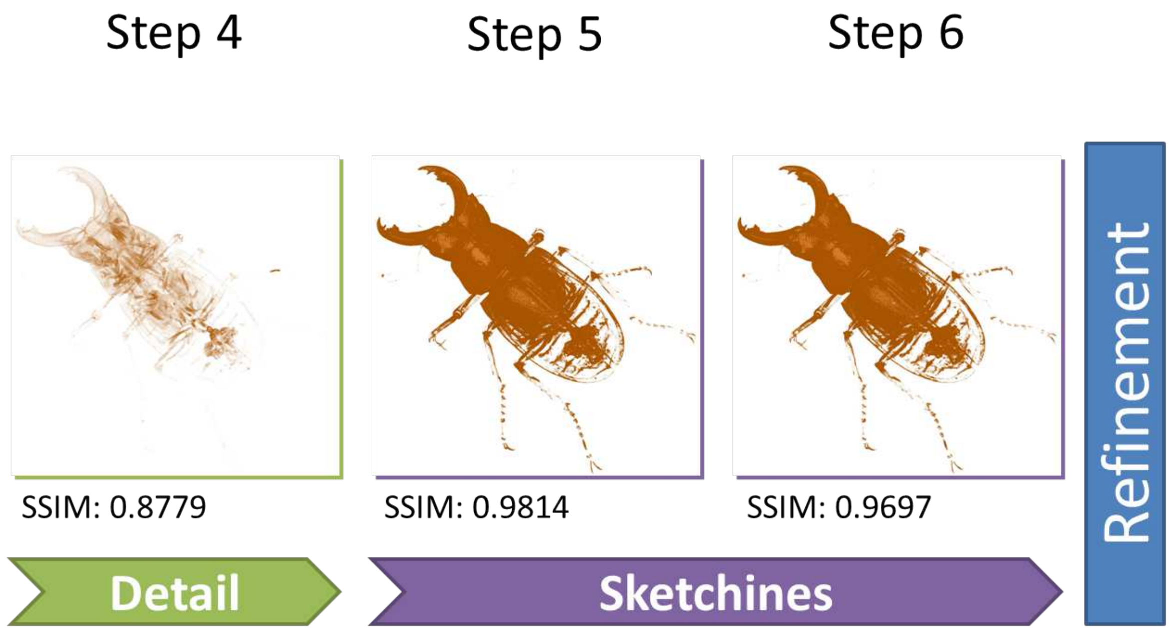


Figure 5.17 (continued)

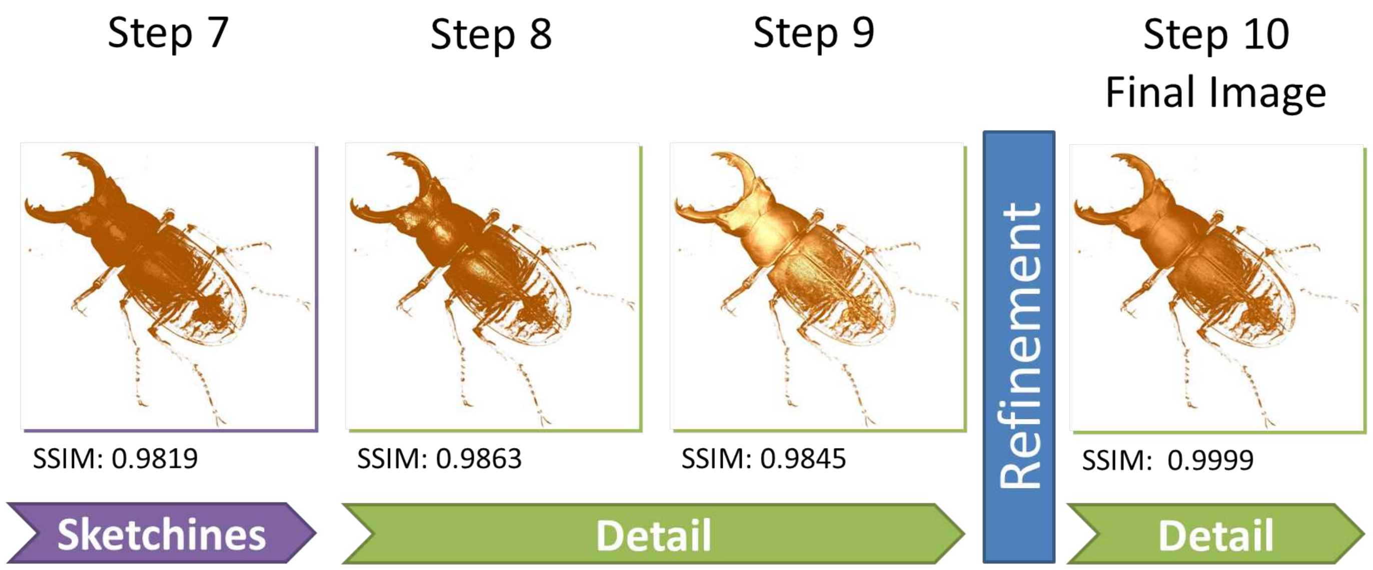
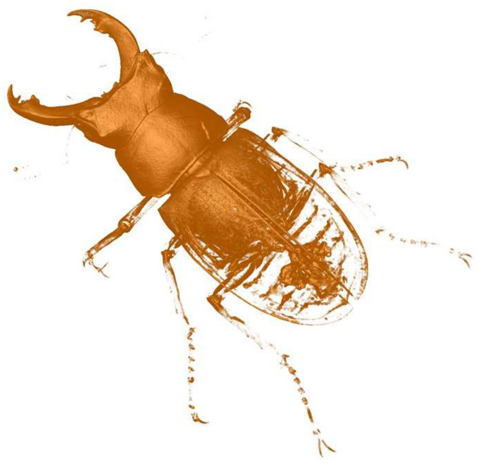
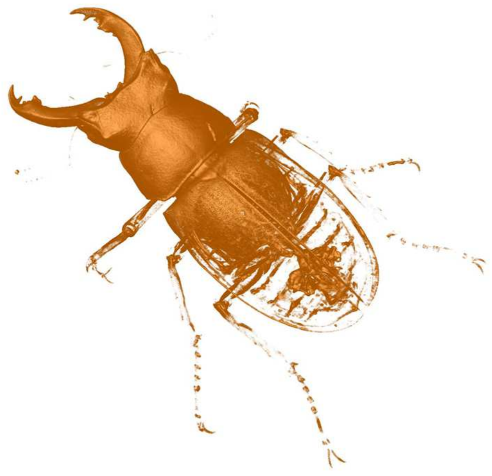


Figure 5.17 (continued)

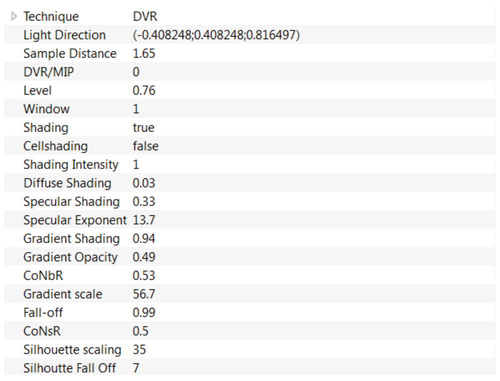


(a) Visualisation of the Stag Beetle by setting the parameter values manually



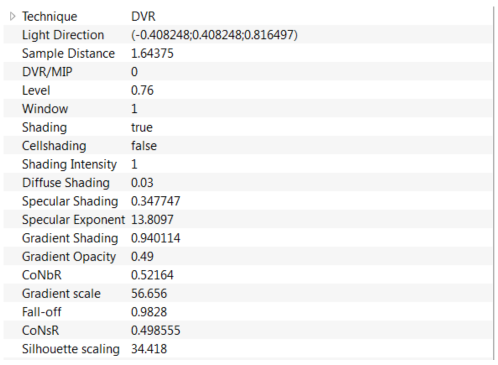
(b) Visualisation of the Stag Beetle created by using ParSAd with DBScan

Figure 5.18: Comparison of visualisations of the Stag Beetle, done by setting the parameter values manually and by using ParSAd with DBScan.



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	1.65
DVR/MIP	0
Level	0.76
Window	1
Shading	true
Cellshading	false
Shading Intensity	1
Diffuse Shading	0.03
Specular Shading	0.33
Specular Exponent	13.7
Gradient Shading	0.94
Gradient Opacity	0.49
CoNbR	0.53
Gradient scale	56.7
Fall-off	0.99
CoNsR	0.5
Silhouette scaling	35
Silhoutte Fall Off	7

(a) Parameter values of the manually created visualisation



▷ Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	1.64375
DVR/MIP	0
Level	0.76
Window	1
Shading	true
Cellshading	false
Shading Intensity	1
Diffuse Shading	0.03
Specular Shading	0.347747
Specular Exponent	13.8097
Gradient Shading	0.940114
Gradient Opacity	0.49
CoNbR	0.52164
Gradient scale	56.656
Fall-off	0.9828
CoNsR	0.498555
Silhouette scaling	34.418

(b) Parameter values of the visualisation created by ParSAd with DBScan

Figure 5.19: Comparison of the parameter values between the manually created visualisation and ParSAd.

5.4.3 Reproducing One Visualisation of the Backpack

For the backpack for we concentrated on the inner parts. We used both clustering algorithms in combination with three different high-level parameters to reproduce the manually created visualisation. For kMeans++ setting number three and number one for DBScan was used for *LLE*. Figure 5.20 shows the final manually created visualisation.

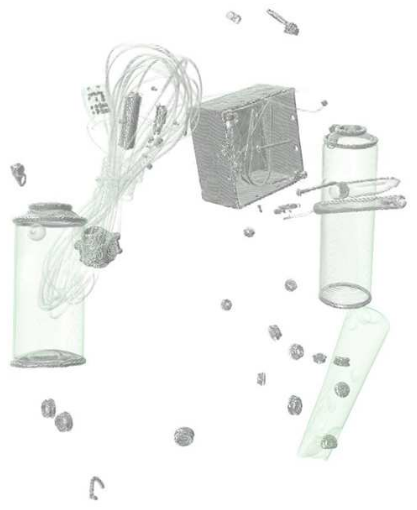


Figure 5.20: Reference of the Backpack visualisation created by setting the parameter values manually.

kMeans++
Reproducing the manually created visualisation with ParSAd took ten steps, see figure 5.21, three steps less as the manually created one. As opposed to the tests cases for the other volumes the SSIM index was jumping. At the beginning we had an index of 0.9736. After four steps it dropped to 0.9560. This was potentially because we concentrated on two different things: the opacity and the contrast of the bottles. Depending on these variables, different images where chosen. But after we selected the *Contrast* high-level parameter, the SSIM index increased continuously after every step. Looking at figure 5.21, at step four it is noticeable that the contrast did decreased after every step when we always chose the lightest and the most similar image to manually created visualisation. This showed again that it is possible to increase or decrease a high-level parameter with ParSAd. After four steps, where we tried to increase the contrast, we chose the *Detail* high-level parameter to make the visualisation look like more to be drawn. Two steps were required to get a similar visualisation, as we did it manually, with a SSIM index of 0.9996. The result can be seen in figure 5.22 which compares the manually created visualisation, figure 5.22a, to the one created by using ParSAD, figure 5.22b. The parameter values, see figure 5.23, were similar as well.

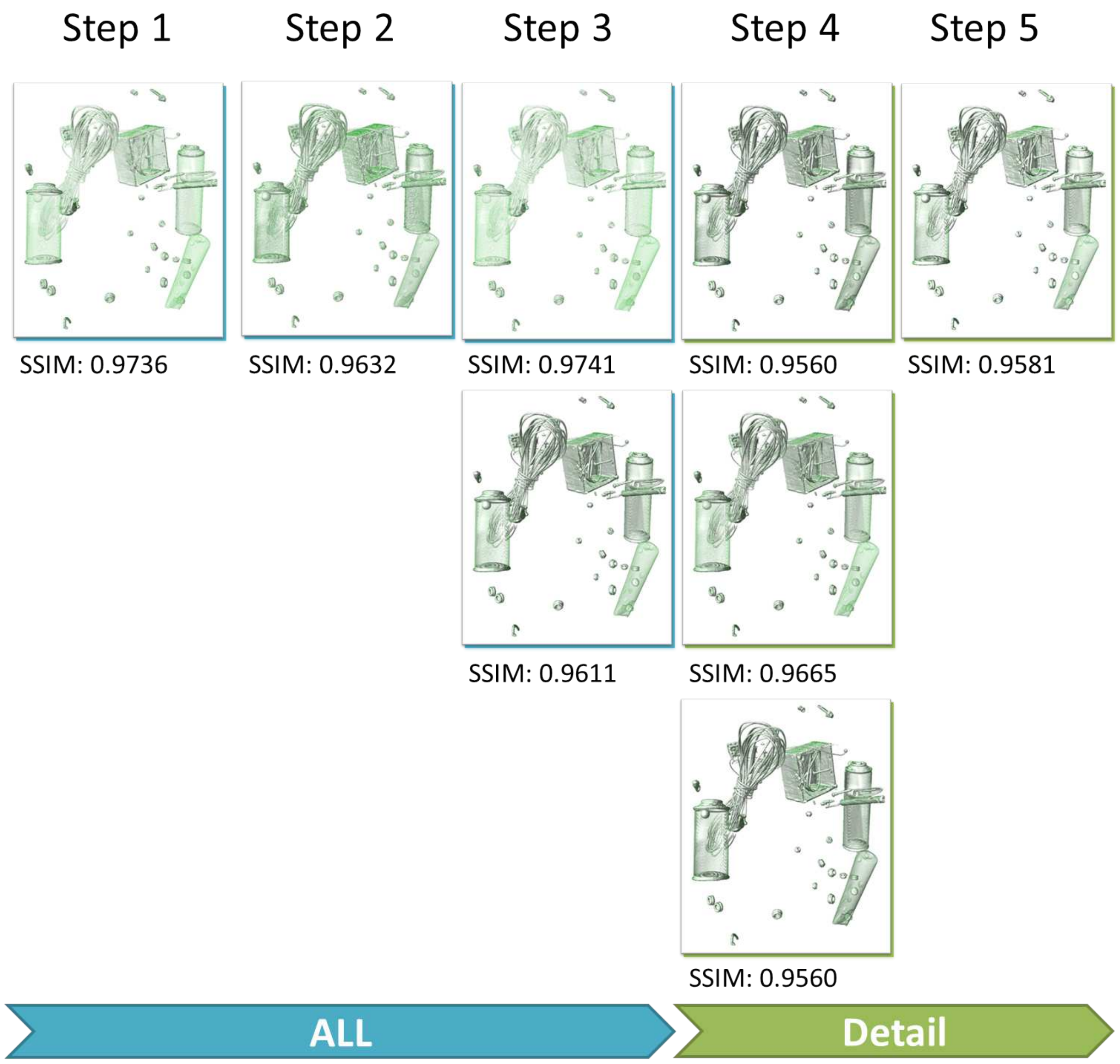


Figure 5.21: Steps one to ten for reproducing the manually created visualisation of the Backpack by using ParSAd in combination with kMeans++.

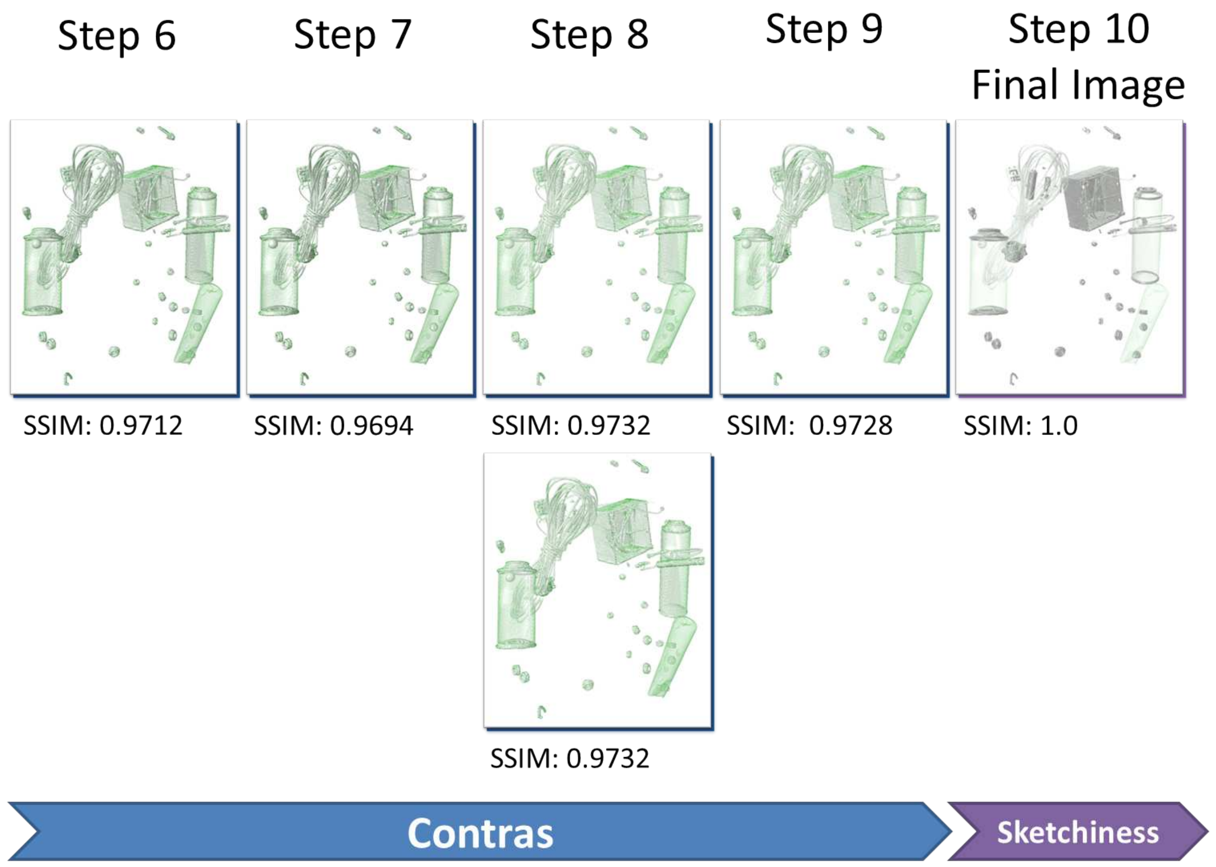
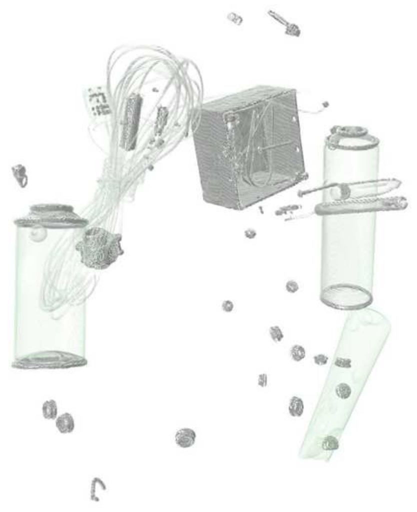
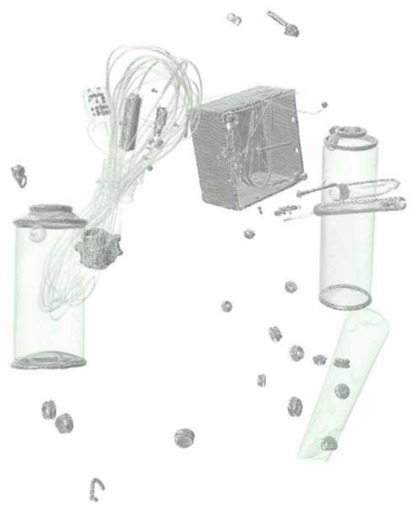


Figure 5.21 (continued)

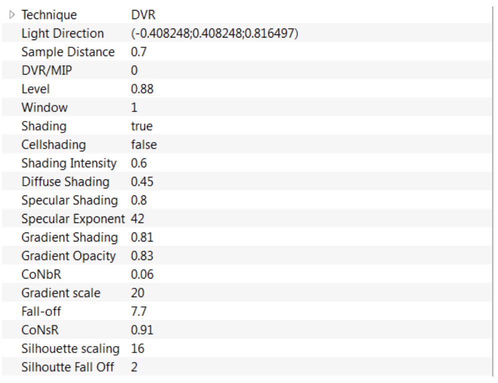


(a) Visualisation of the Backpack by setting the parameter values manually



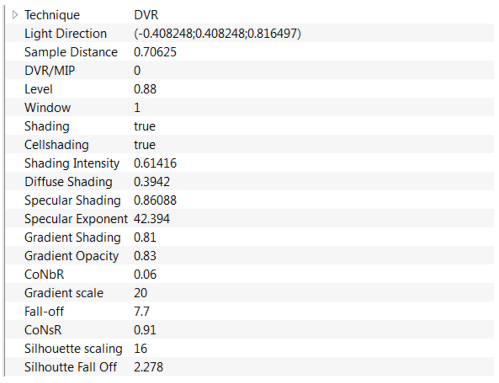
(b) Visualisation of the Backpack created by using ParSAd with kMeans++

Figure 5.22: Comparison of visualisations of the Backpack, done by setting the parameter values manually and by using ParSAd with kMeans++.



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.7
DVR/MIP	0
Level	0.88
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.6
Diffuse Shading	0.45
Specular Shading	0.8
Specular Exponent	42
Gradient Shading	0.81
Gradient Opacity	0.83
CoNbR	0.06
Gradient scale	20
Fall-off	7.7
CoNsR	0.91
Silhouette scaling	16
Silhoutte Fall Off	2

(a) Parameter values of the manually created visualisation

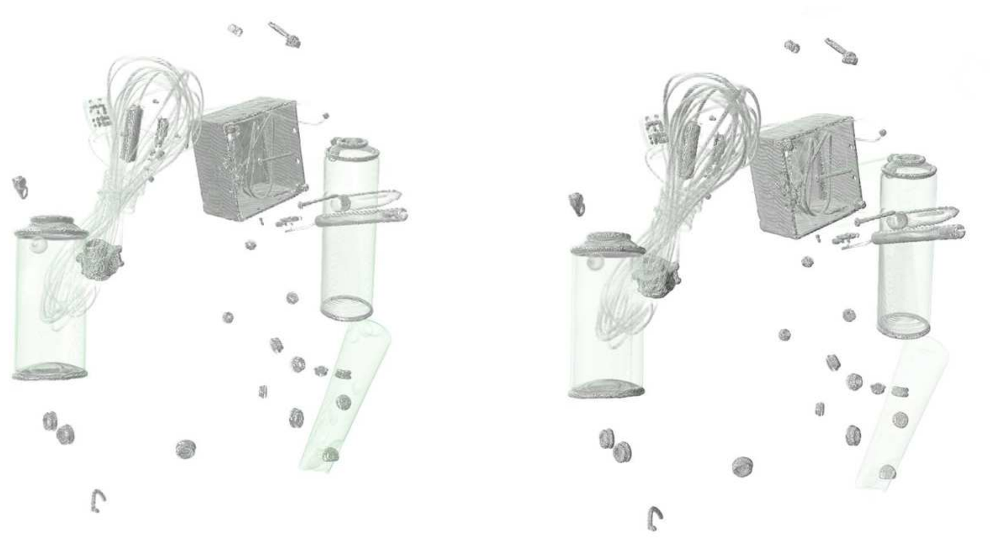


Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.70625
DVR/MIP	0
Level	0.88
Window	1
Shading	true
Cellshading	true
Shading Intensity	0.61416
Diffuse Shading	0.3942
Specular Shading	0.86088
Specular Exponent	42.394
Gradient Shading	0.81
Gradient Opacity	0.83
CoNbR	0.06
Gradient scale	20
Fall-off	7.7
CoNsR	0.91
Silhouette scaling	16
Silhoutte Fall Off	2.278

(b) Parameter values of the visualisation created by ParSAd with kMeans++

Figure 5.23: Comparison of the parameter values between the manually created visualisation and ParSAd.

DBScan
Four different high-level parameters - *All*, *Detail*, *Contrast* and *Sketchiness* - were necessary to obtain a final visualisation with a SSIM index of 0.9894 in comparison to the manually created one. The result can be seen in figure 5.24 which compares the manually created visualisation, figure 5.24a, to the one created by using ParSAD, figure 5.24b. A total number of 600 images was created and explored. It took 11 steps to reproduce the manually created visualisation when we used ParSAd in combination with the DBScan clustering algorithm (see figure 5.25). For every creation, including initial clustering, re-clustering and refinement, the automatic found an adequate ϵ so that different clusters could be achieved as opposed to one cluster that all images were assigned to.This led to reaching of the final image in only a few steps. Figure 5.26 shows the comparison of the parameter values. As for the other visualisations, the values were similar.


(a) Visualisation of the Backpack by setting the parameter values manually

(b) Visualisation of the Backpack created by using ParSAd with DBScan

Figure 5.24: Comparison of visualisations of the Backpack, done by setting the parameter values manually and by using ParSAd with DBScan.

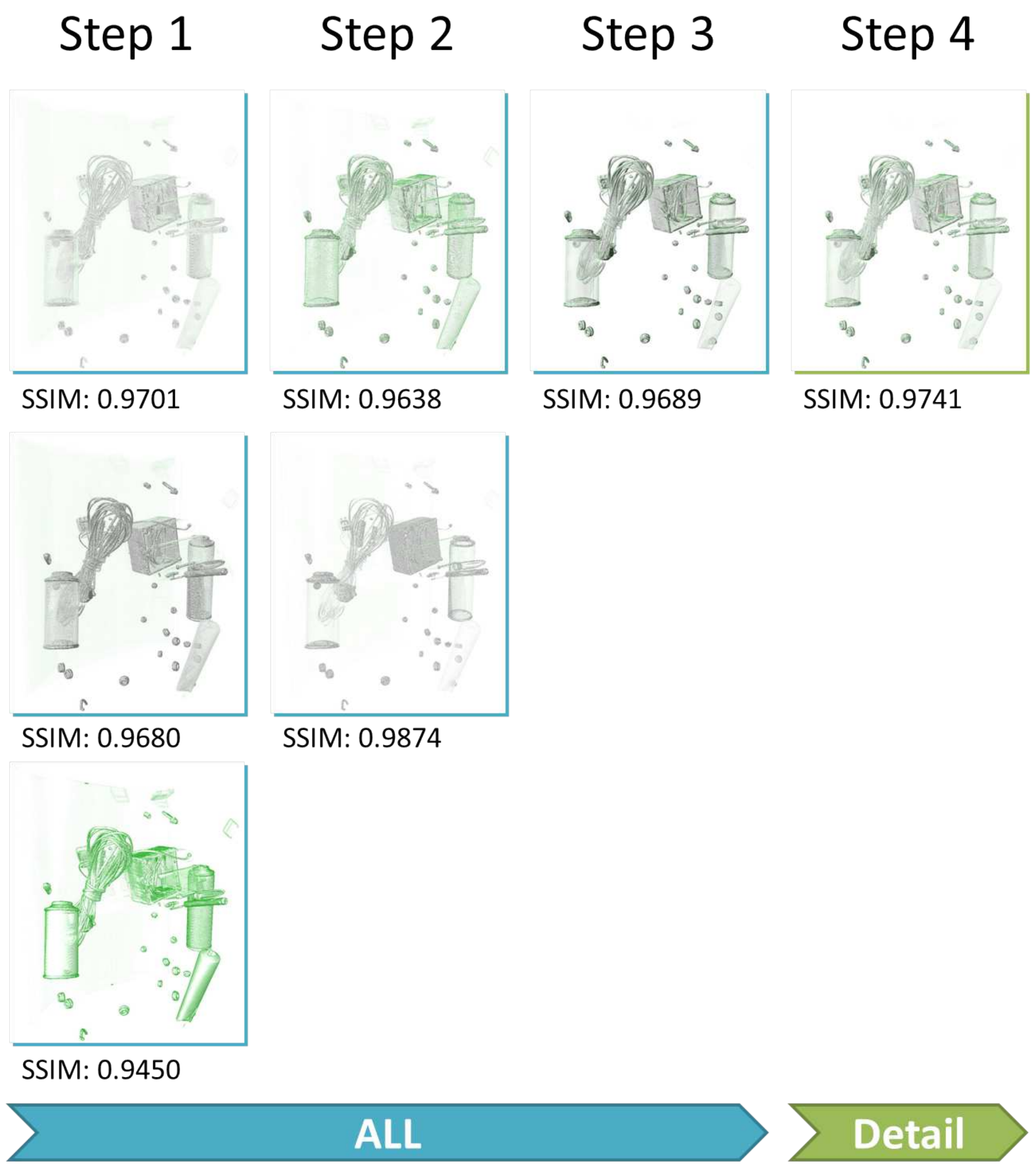


Figure 5.25: Steps one to ten for reproducing the manually created visualisation of the Backpack by using ParSAd in combination with kMeans++.

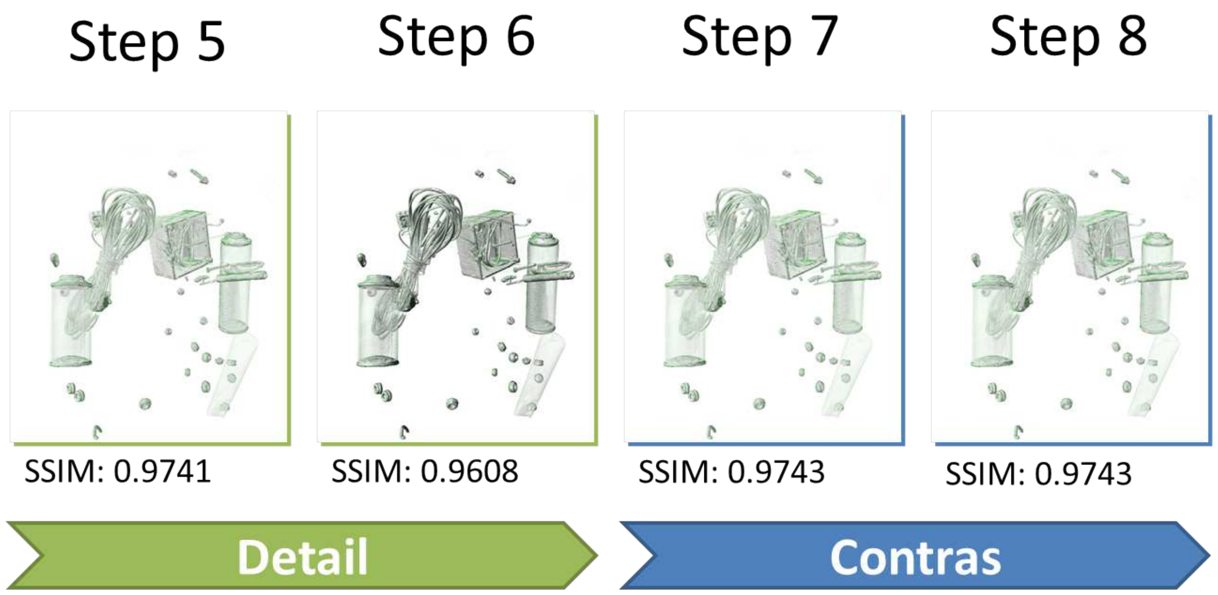


Figure 5.25 (continued)

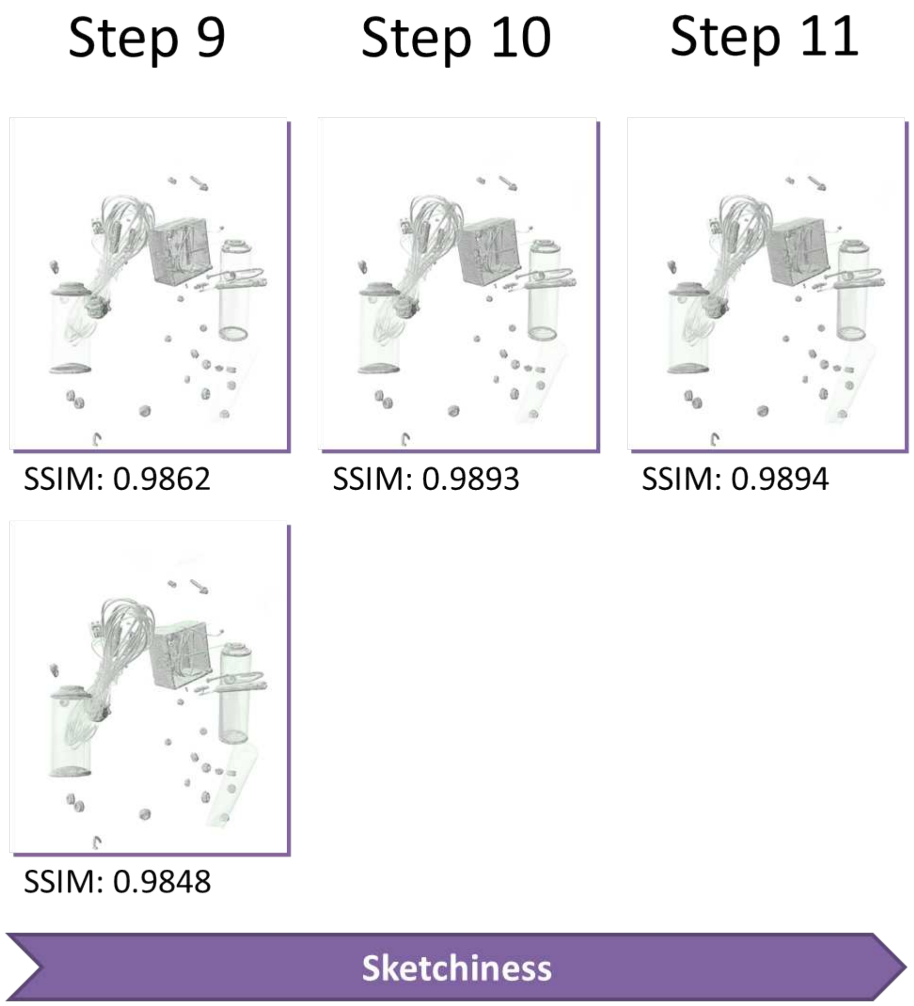
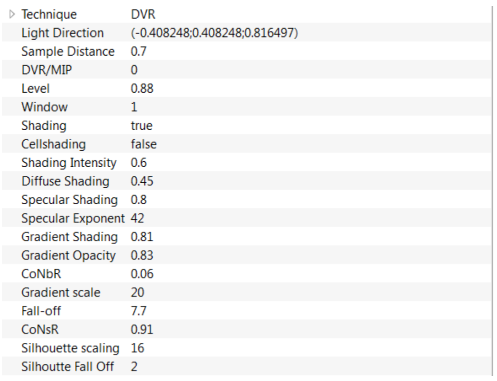
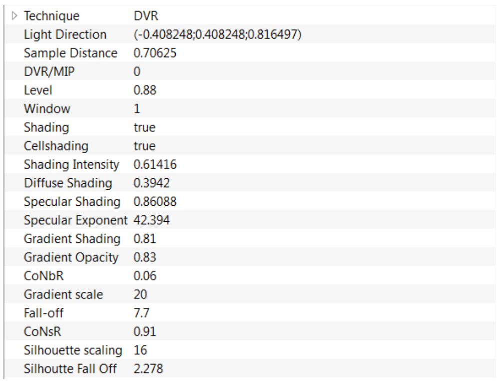


Figure 5.25 (continued)



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.7
DVR/MIP	0
Level	0.88
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.6
Diffuse Shading	0.45
Specular Shading	0.8
Specular Exponent	42
Gradient Shading	0.81
Gradient Opacity	0.83
CoNbR	0.06
Gradient scale	20
Fall-off	7.7
CoNsR	0.91
Silhouette scaling	16
Silhoutte Fall Off	2

(a) Parameter values of the manually created visualisation



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.70625
DVR/MIP	0
Level	0.88
Window	1
Shading	true
Cellshading	true
Shading Intensity	0.61416
Diffuse Shading	0.3942
Specular Shading	0.86088
Specular Exponent	42.394
Gradient Shading	0.81
Gradient Opacity	0.83
CoNbR	0.06
Gradient scale	20
Fall-off	7.7
CoNsR	0.91
Silhouette scaling	16
Silhoutte Fall Off	2.278

(b) Parameter values of the visualisation created by ParSAd with DBScan

Figure 5.26: Comparison of the parameter values between the manually created visualisation and ParSAd.

5.4.4 Reproducing One Visualisation of the Skewed Head

We tested one single and the combination of different high-level parameters. For all the before executed test cases, regardless of whether the parameter values were set manually or by using ParSAd, the same transfer-function was used for the volume. For the last test case we wanted to reproduce a visualisation of the Skewed Head (figure 5.27), which the transfer-function was set manually for (figure 5.28) by using the *Transfer-function* and *Colour* in combination with *All* high-level parameter. The main focus for manually created visualisation was on the teeth and their roots. For this test case we only used kMeans++ for testing, as the focus for this thesis was on parameter values and not improving the generation of the transfer-function and therefore we keep this short. For this test-case, setting number three in table 5.1 was used for *LLE*.

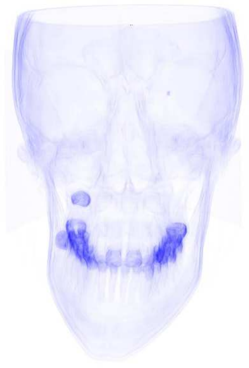


Figure 5.27: Reference visualisation created by setting the parameter values manually.

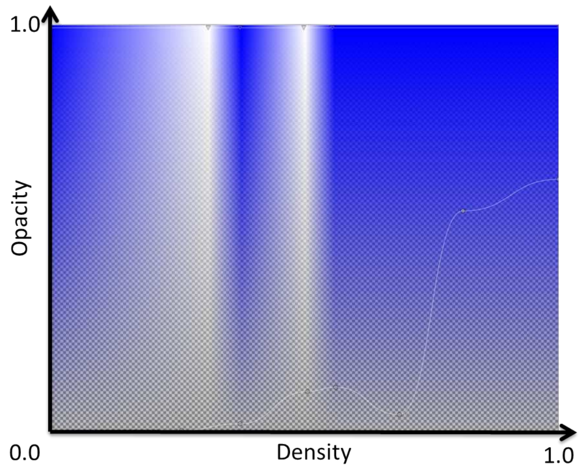


Figure 5.28: The transfer-function of the manually created visualisation.

To reach an identical visualisation to the manually created one, it took 17 steps. These included finding an acceptable transfer-function and colour and applying the *All* high-level parameter. The manual transfer-function had eight control points, and six points which were coloured. This would correspond to 14 manual steps compared to 12 we used ParSAd for the final transfer-function. If taking the 14 additional low-level parameters into account, 28 steps were necessary to create the final manual visualisation. The SSIM index increased after every step (see figure 5.29). The final transfer-function looked similar to the manual one (see figure 5.30). The only differences were that the transfer-function created with ParSAd had more control points than the manually created one and that the distribution of the colour is also slightly different. However both transfer-functions influenced the volume in the same way. After applying the *All* high-level parameter to the visualisation it took five steps, where the *All* high-level parameter was refined two times to create a visualisation which had a SSIM index of 0.9996 in comparison to the manually created one.

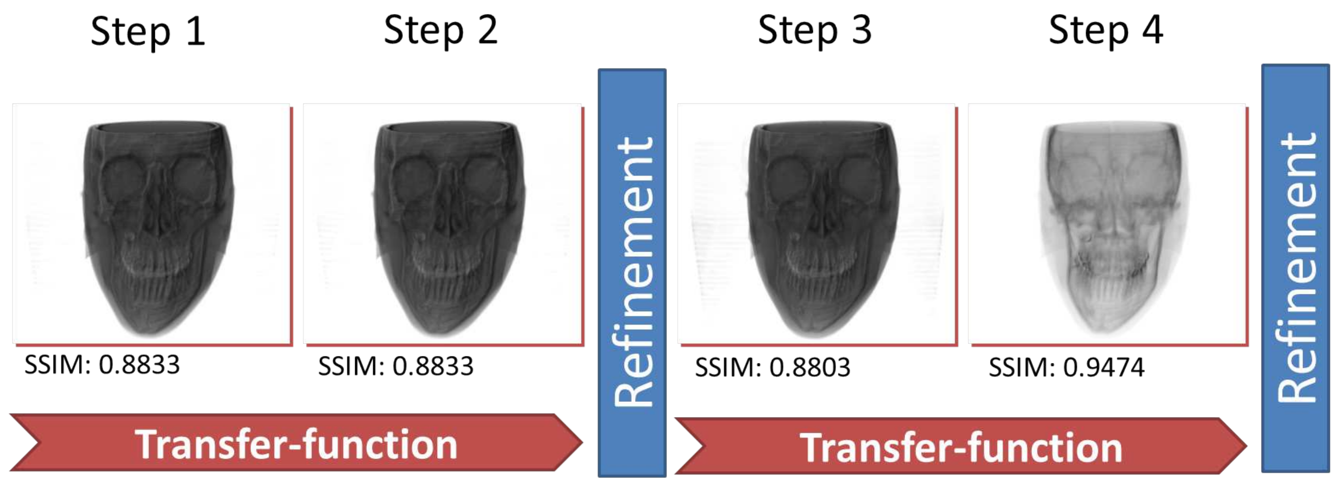


Figure 5.29: Steps one to seventeen for reproducing the manually created visualisation of the Skewed Head including the transfer-function and colour by using ParSAd in combination with kMeans++.

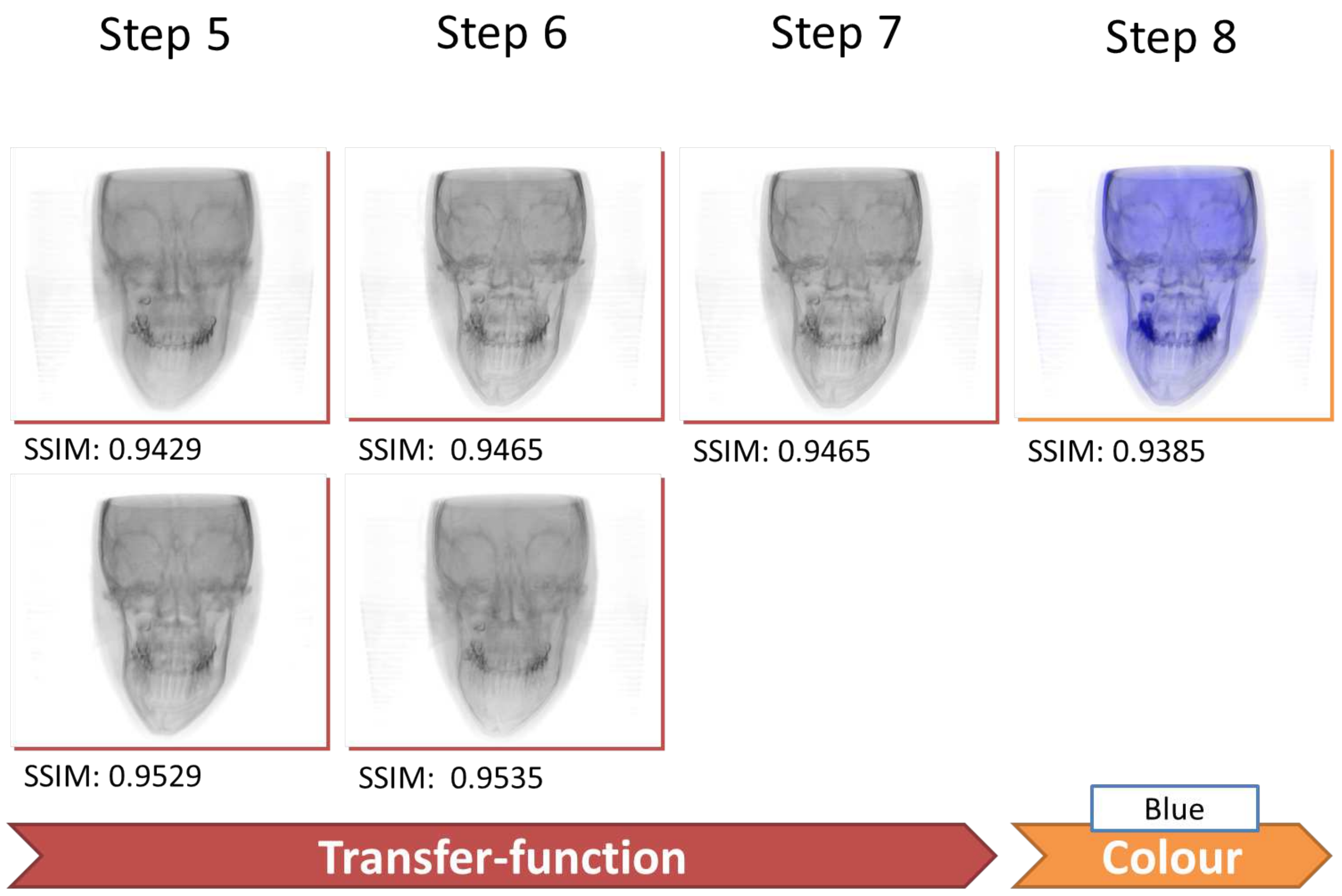


Figure 5.29 (continued)

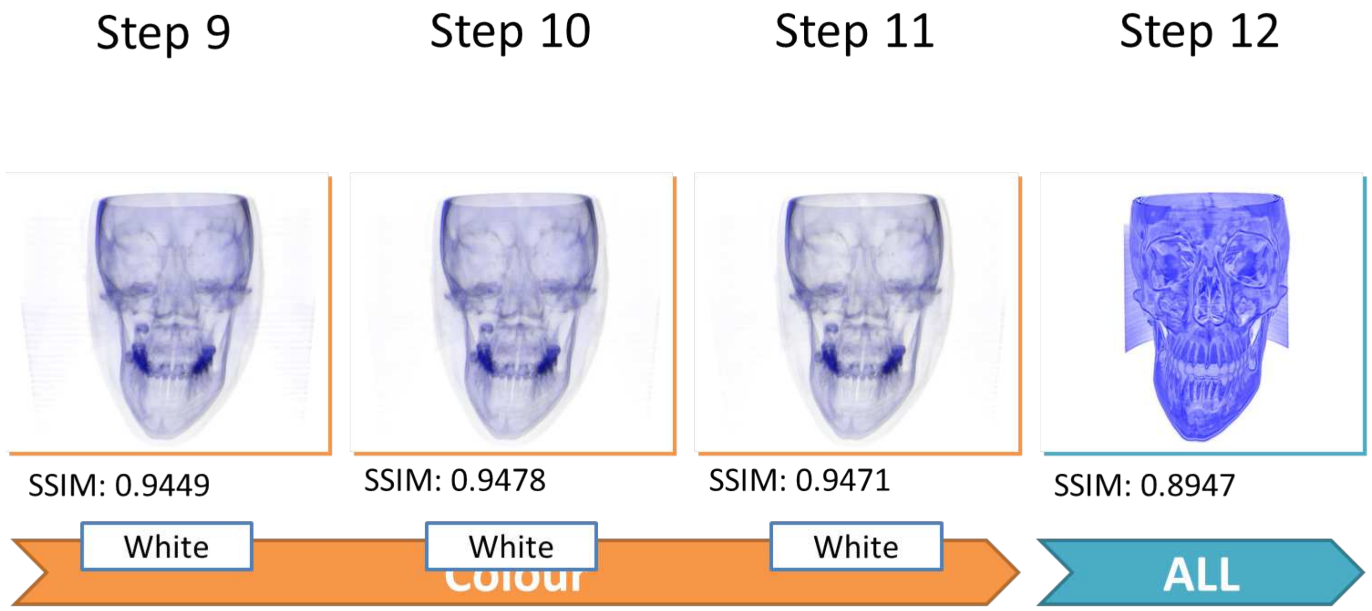


Figure 5.29 (continued)

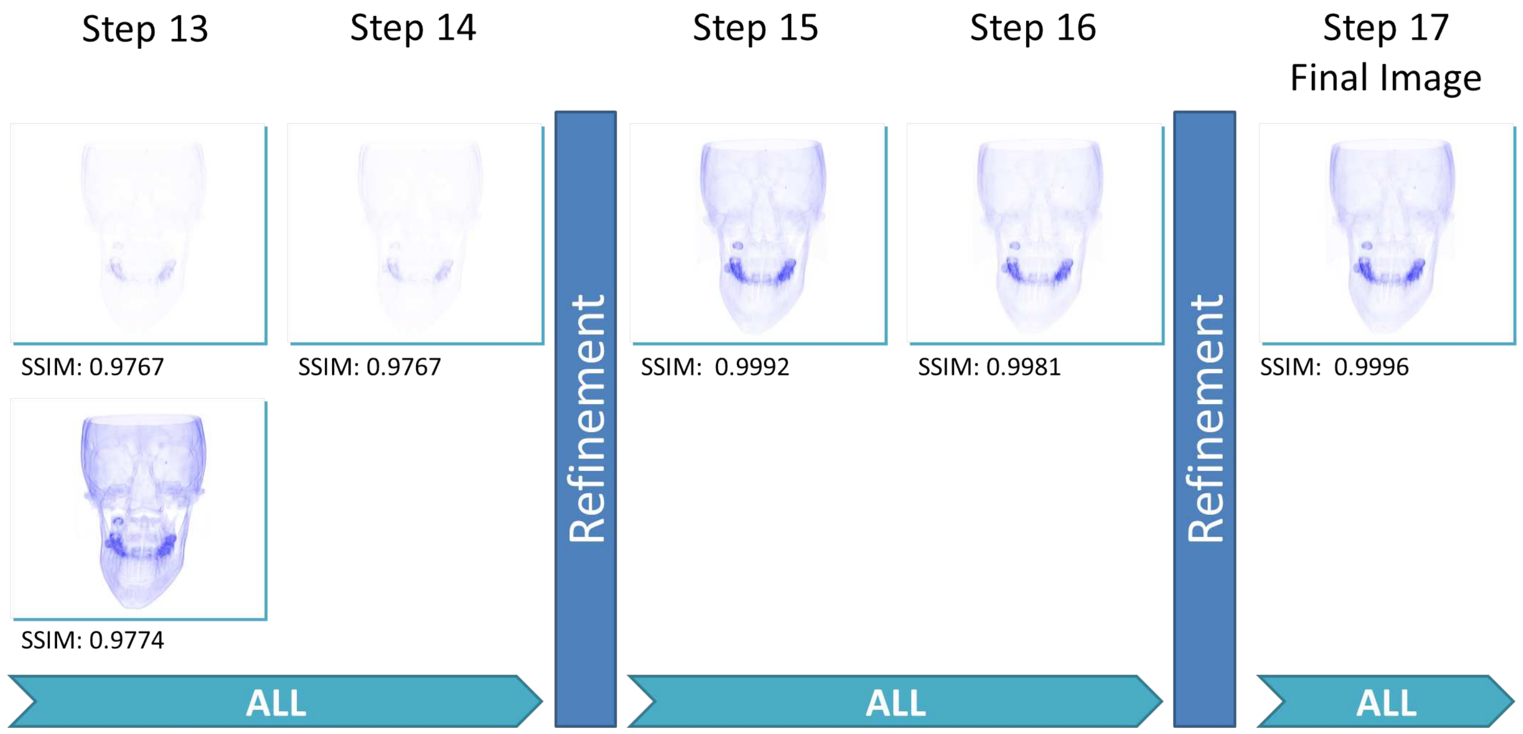
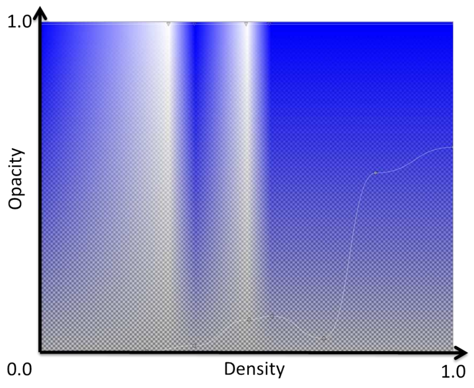
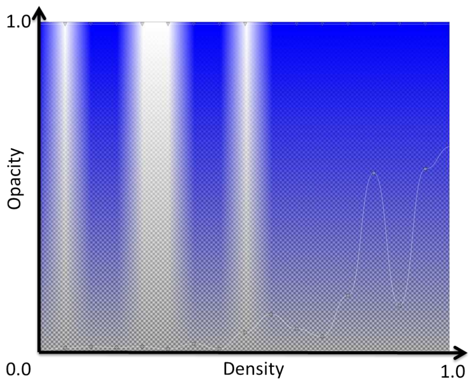


Figure 5.29 (continued)



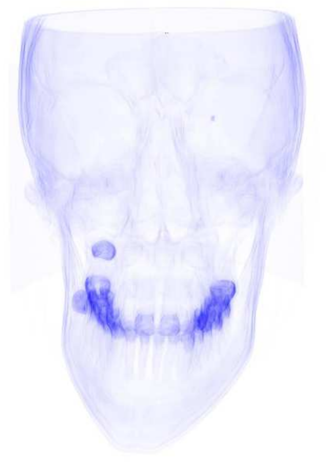
(a) Manually created transfer-function



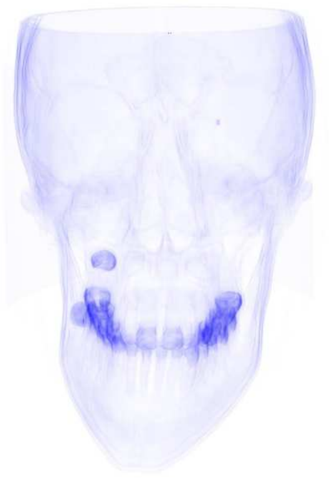
(b) Final transfer-function with ParSAd

Figure 5.30: Comparison of the manually and the with ParSAd created transfer-function.

With this test case we showed that it is possible to create almost the same transfer-function as when creating it manually. The number of steps needed to achieve this was two steps less than for the manually created visualisation. The result can be seen in figure 5.31 which compares the manually created visualisation, figure 5.31a, to the one created by using ParSAD, figure 5.31b. The parameter values differ slightly, as can be seen in figure 5.32.

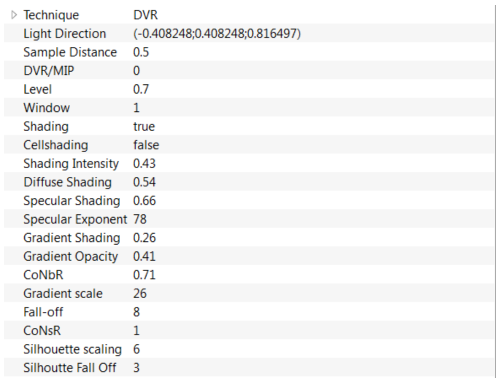


(a) Visualisation of the Skewed Head by setting the parameter values manually



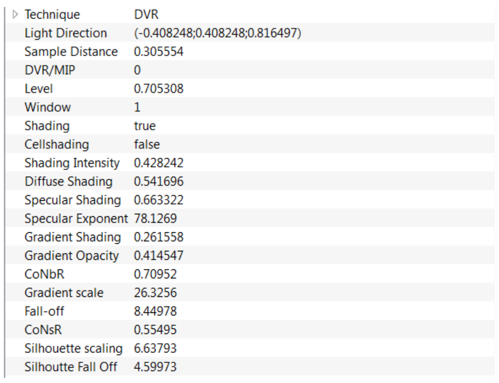
(b) Visualisation of the Skewed Head created by using ParSAd with kMeans++

Figure 5.31: Comparison of visualisations of the Skewed Head, done by setting the parameter values manually and by using ParSAd with kMeans++.



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.5
DVR/MIP	0
Level	0.7
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.43
Diffuse Shading	0.54
Specular Shading	0.66
Specular Exponent	78
Gradient Shading	0.26
Gradient Opacity	0.41
CoNbR	0.71
Gradient scale	26
Fall-off	8
CoNsR	1
Silhouette scaling	6
Silhoutte Fall Off	3

(a) Parameter values of the manually created visualisation



Technique	DVR
Light Direction	(-0.408248;0.408248;0.816497)
Sample Distance	0.305554
DVR/MIP	0
Level	0.705308
Window	1
Shading	true
Cellshading	false
Shading Intensity	0.428242
Diffuse Shading	0.541696
Specular Shading	0.663322
Specular Exponent	78.1269
Gradient Shading	0.261558
Gradient Opacity	0.414547
CoNbR	0.70952
Gradient scale	26.3256
Fall-off	8.44978
CoNsR	0.55495
Silhouette scaling	6.63793
Silhoutte Fall Off	4.59973

(b) Parameter values of the visualisation created by ParSAd with DBScan

Figure 5.32: Comparison of the parameter values between the manually created visualisation and ParSAd.

5.5 Testing ParSAd with Users

Finally we conducted a user study to obtain more relevant and independent results, and to evaluate how ParSAd performs under more realistic conditions. So as not to go beyond the scope of this thesis, we tested the program with only two users.

5.5.1 The Users

The first user was a physicist, who uses software to visualise mathematical functions and physical simulations, and who is used to setting parameters for this domain. He is a novice in the field of volume visualisation. We will further refer to this user as *User 1*
The second user, *User 2*, was an illustrator who works with software for photo manipulation, 3D modeling and artwork. He is used to terms like opacity, sketchiness, contrast, etc. and knows how to manipulate volumes and their transfer-function.

5.5.2 Aim and Evaluation of the Test

For the test we wanted the users to highlight the teeth of the Skewed Head volume first by setting the parameters manually, and second by using ParSAd. We decided to use kMeans++ - so as to not influence the test by a badly chosen ϵ - for DBScan and an image size of 128x128 with 20 neighbours for *LLE*, as this setting was already found to deliver good results in section 5.3.The users only received a verbal introduction to the test and were not shown what the target image should look like. Additionally we wanted the users to start with the *All* parameter of ParSAd and try to achieve their final visualisation by only using this parameter. Only if not satisfied with the final result were they to continue using all the other high-level parameters. We did this to see whether they would also achieve their aim with one single high-level parameter. To see whether creating the visualisation manually or by using ParSAd is more effective we compared

the number of steps and the time taken to highlight the teeth of the Skewed Head. For the manually created visualisation we counted the number of steps needed. One step is defined as moving one slider or pressing a button. Additionally we measured the time needed from the first click to the final result.
To evaluate the performance of ParSAd we counted the number of exploration steps, which meant selecting an image/images and pressing the ">" (*right arrow*) button or one of the high-level parameters. Also for ParSAd we measured the time. This included the user's idle time while ParSAd generated and clustered images.

5.5.3 Preparation of the User Test

Before we started our test, both users were given an introduction to VolumeShop. We showed them how to set the transfer-function and the parameters manually for a volume visualisation. For this introduction the Backpack volume was used so as not to influence them afterwards when performing the original test. Next we showed the users how to use ParSAd and the idea behind it. First we started our test with *User 1* by showing him VolumeShop, how to load a template and how to use and set the different low-level parameters. Introduction into the program and practising the use of different parameters so that *User 1* felt comfortable took about 40 minutes. *User 2* took only 15 minutes to get used to setting parameters manually in VolumeShop. ParSAd was quickly explained in two minutes to both users.

5.5.4 Results and Conclusion

The first test was executed by *User 1*, the second by *User 2*. For both users, table 5.3 shows the overall number of steps and time, including idle time for ParSAd, both needed to achieve the aim of highlighting the teeth manually and using ParSAd. Figure 5.33 shows the initial volume visualisation both users started with.

User	Steps Manually	Parameters	Time	Steps ParSAd	High-level Parameters	Time
User 1	25 steps	7 parameters	12 minutes	17 steps	3 high-level parameters	14 minutes
User 2	19 steps	6 parameters	5 minutes	16 steps	3 high-level parameters	7 minutes

Table 5.3: The number of steps and time needed by using ParSAd compared to the number of steps needed to set the low-level parameters for the visualisation manually for both user.

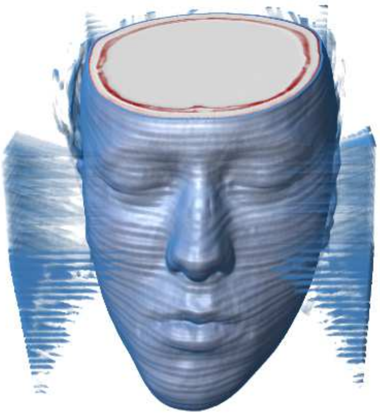
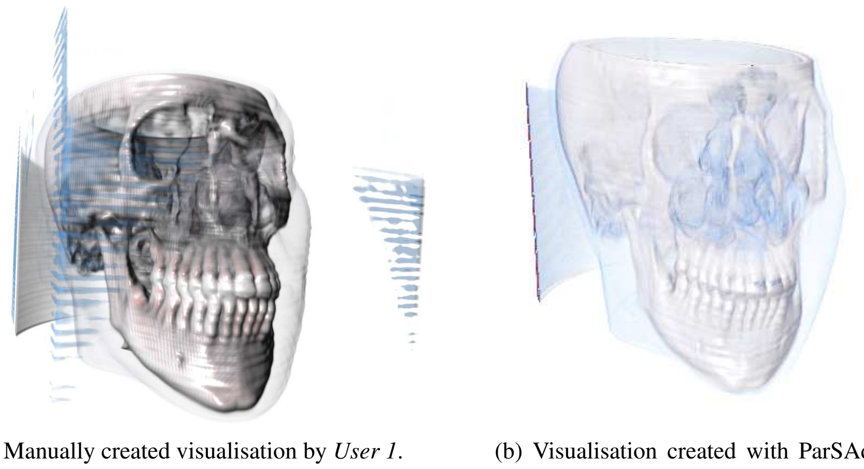


Figure 5.33: Starting visualisation for setting the parameter values manually and with ParSAd.

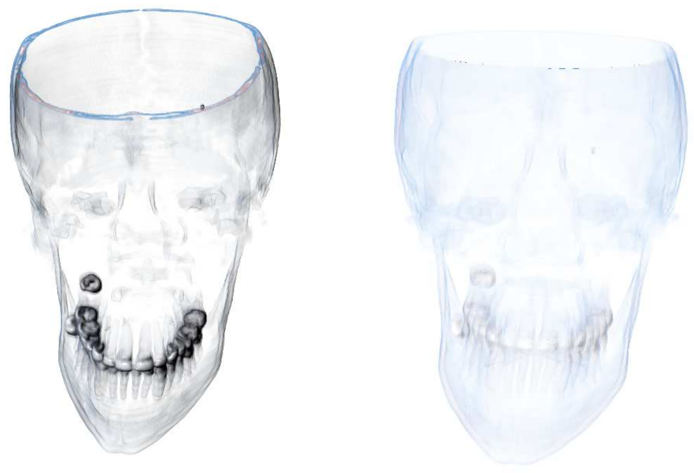
For both users the number of steps was lower when they used our approach. Both started with using only one high-level parameter. We saw that they were able to achieve an almost satisfying result. It took *User 1* ten steps in eight minutes and *User 2* nine steps in five minutes to achieve this. Figure 5.34a shows the results of the manually created image, and figure 5.34b shows the image created with ParSAd using the *All* high-level parameter for *User 1*. Figure 5.35a and 5.35b show the results for *User 2*.



(a) Manually created visualisation by *User 1*.

(b) Visualisation created with ParSAd by *User 1*.

Figure 5.34: Comparison of the results for *User 1*, between the manually and ParSAd created visualisation for which only the high-level parameter *All* was used.



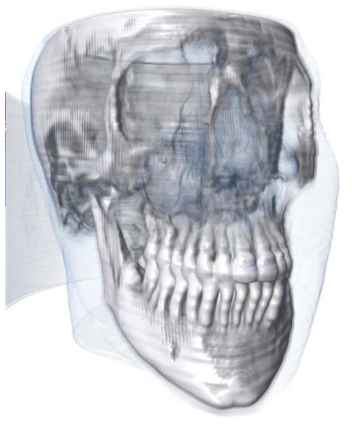
(a) Manually created visualisation by *User 2*.

(b) Visualisation created with ParSAd by *User 2*.

Figure 5.35: Comparison of the results for *User 2*, between the manually and ParSAd created visualisation for which only the high-level parameter *All* was used.

However, both users wanted to increase the level of detail and contrast. Therefore both used two additional high-level parameters for further refinement. *User 1* took seven additional steps which included three steps for the high-level parameter *Contrast* and four for *Detail*. *User 2* chose these two high-level parameters as well, where four steps for *Contrast* and three for *Detail* were needed. After these refinements, both achieved an almost similar result with ParSAd as with setting parameters manually. See figure 5.36 for the final results. Figure 5.36a shows the results for *User 1* and figure 5.36b displays the results for *User 2*.
Compared to setting parameters manually, ParSAd took more time overall for the same task in our tests. However both users' tests included one initial and five refinement calculations of the images, where each re-calculation took about 30 seconds, which totals to about three minutes. Taking this into account, the effective working time of the users when using ParSAd was less or equal to those when they set the parameters manually. Table 5.4 shows a detailed overview of the number of steps, and the time needed for calculating and selecting the images. It includes the times for the initial step for each high-level parameter, the exploration and refinements.

After the test we asked both users how satisfied they were with their result and how they felt about their experience when setting the parameters manually and using ParSAd. *User 1* said that he was not quite satisfied with the result when setting the parameter values manually. After five to six steps, it became frustrating because there were so many different parameters to use, where some of them influenced each other, and different parameters had the same effects to the visualisation. Sometimes he found a good setting for one of the parameter values but a slight change of another parameter completely changed the whole visualisation. Therefore he said that


(a) Final visualisation created with ParSAd by *User 1*.

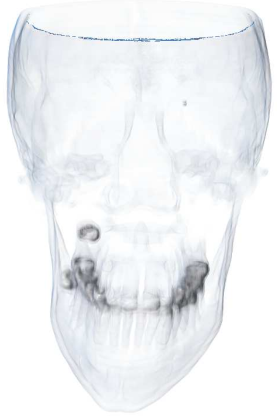

(b) Final visualisation created with ParSAd by *User 2*.

Figure 5.36: The final results of both users created by ParSAd using the high-level parameters *All*, *Contrast* and *Detail*.

User	High-level Parameter	Nr. of Steps	Time
User 1	Initial All	1 step	30 seconds
	All	4 steps	4 minutes
	Refinement All	1 step	30 seconds
	All	4 steps	3 minutes
	Initial Detail	1 step	35 seconds
	Detail	3 steps	3 minutes
	Initial Contrast	1 step	30 seconds
	Contrast	2 steps	2 minutes
User 2	Initial All	1 step	30 seconds
	All	4 steps	2 minutes
	Refinement All	1 step	30 seconds
	All	5 steps	2 minutes
	Initial Contrast	1 step	30 seconds
	Contrast	2 steps	30 seconds
	Initial Detail	1 step	35 seconds
	Detail	1 step	30 seconds

Table 5.4: Detailed list of steps needed by using ParSAd including the time for each step.

his process of setting parameter values manually resulted in a pure trial and error one. For example he tried to remove the surrounding box of the head, but he was not able to find an adequate setting of the parameter values for that. When using ParSAd, he said everything was very easy and almost no explanation was necessary. “I just clicked the images I liked best and reflected what I wanted to achieve. I did not have to take care about any parameters or combination of them “.
For *User 2* the process of setting parameters manually was easier as he was already familiar with this kind of process from his experience with other software. However he said that depending on the number of parameters, the process sometimes led also to trial and error. He was not as satisfied with the final result when using ParSAd as the first user but he said that it is much easier to do a visualisation with our approach. He would prefer using ParSAd for an initial visualisation and then do a refinement by setting a few parameters manually. Further he wished the calculation of the images would be faster.

The user tests showed that our approach equally enables non-expert and expert users to quickly achieve results in visualisation similar to those achieved when setting parameters manually. This can be done with a fewer number of steps while a longer time is taken. This is due to the time-consuming image calculations which are part of our method. An increase of the calculation performance might help our method in being accepted by users and would therefore be a logical next step in further developing our approach.

5.6 Overall Results of Testing ParSAd

The aim of this thesis was to find a new and easy way of setting many different parameter values for a visualisation algorithm to get the desired result and to explore how an image-centric method can be utilized for efficient specification of parameter values for visualization algorithms. By mixing several known techniques from the area of image processing, as mentioned in the design chapter (3), we presented a prototype, called parameter selection advisor - short ParSAd.

By running different tests to find an adequate setting for *LLE* (see section 5.3) for the different volumes in combination with the two clustering algorithms, we showed that our implementation of *LLE* works and similar images lay together in the 2D *LLE* coordinate system. Using the correct settings for *LLE* made it possible to perform an accurate clustering where similar and almost similar images were correctly assigned to one cluster. Furthermore doing the calculation of the Euclidean distance between two images and the calculation of the different matrices for *LLE* in parallel made ParSAd fast. More than 150 images, having the size of 256x256, were able to be compared within 30 seconds. Additionally we were able to test the automatic method to select ε for DBScan. If this feature is activated it can improve the result of the DBScan clustering algorithm (see figures 5.3d to 5.3f or 5.5d in section 5.3 where the number of *OK* was very high).However, sometimes the method was not able to find a good ε value. For example for setting numbers five, six and seven, see figures 5.4d to 5.4f for the Backpack volume in section 5.3. The ε was too small which led to many images being classified as noise. The method still needs improvement to avoid such behaviour.

As we have seen from the test cases, ParSAd is applicable for easily assisting the user in exploring many different parameter values for a volume visualisation in a few steps. It is possible by using ParSAd to create the same visualisations as when setting the low-level parameters manually. For all conducted tests, table 5.5 shows a summary of the number of test steps needed by ParSAd compared to the number of steps for the manually created visualisation.

Volume	Clustering	Nr. of Steps - Manually	Nr. of Steps - ParSAd
Stag Beetle	kMeans++	14 steps	7 steps
	DBScan	14 steps	7 steps
Stented Abdominal Aorta	kMeans++	14 steps	8 steps
	DBScan	14 steps	7 steps
Backpack	kMeans++	14 steps	10 steps
	DBScan	14 steps	11 steps
Skewed Head	kMeans++	28 steps	18 steps
Skewed Head User one	kMeans++	25 steps	17 steps
Skewed Head User two	kMeans++	19 steps	16 steps

Table 5.5: The number of steps needed by using ParSAd compared to the number of steps needed to set the low-level parameters for the visualisation manually.

For every test we made, the number of steps were less than the number of steps that would be needed by an expert user, who needs only one step to set one low-level parameter.There was also no big difference between the two clustering algorithms when we reproduced the different visualisations. The number of steps between them was almost equal.
We also wanted to find a way to make parameters more understandable to a non-expert user. The concept of high-level parameters made that possible. The high-level parameter *All* can be used to set all low-level parameters at once and to create a visualisation without using any other high-level parameters, as we showed for the visualisation of the Stented Abdominal Aorta in section 5.4.1.This offers a non-expert user easy access to volume rendering or any other visualisation software using parameters, without much knowledge of low-level parameters and how to set and combine them.

CHAPTER 6

Conclusion and Future Work

The implementation of a prototype of our idea enabled us to show that our approach works with a direct volume rendering algorithm. We conducted different tests, including two user tests with an almost non-expert and an expert user, while creating and reproducing different visualisations. The outcome of these tests shows that the use of our approach takes fewer steps than setting parameters manually and that our approach is also easier for users to work with. One of our initial intents was that almost no low-level parameter knowledge should be necessary for the use of ParSAd. If the user does not know how low-level parameter combinations influence the visualisation, he/she can use different pre-defined high-level parameters which combine such combinations to one single parameter. However we found that sometimes knowledge of the underlying low-level parameter is necessary when using the different high-level parameters in combination. An example would be in reproducing a visualisation of the Stag Beetle with DB-Scan in section 5.4.2. In this situation, if the user does not know for this test case the value of the gradient opacity low-level parameter causes the weak influence of the *Detail* high-level parameter, and that he/she has to change the sketchiness of the visualisation to increase the gradient opacity, then this could lead to too many steps when creating the visualisation. In summary we can say that ParSAd can be used to create a volume visualisation very quickly within a few steps. There is no need for the user to set low-level parameters manually.
With this thesis we showed that *LLE* works in combination with the different clustering algorithms. Equal images where assigned to the same cluster. Implementing the calculation of the coordinates and kMeans++ in parallel made it possible that no over night pre-processing was necessary, as the calculation of the coordinates of 150 images took about 30 seconds. However the user test showed that this still needs improvement if ParSAd should be used in practise.
Finally an automatic method for setting parameters of DBScan became a secondary contribution of our thesis because without any knowledge of the distribution of the *LLE* coordinates in space it is nearly impossible for the user to find an adequate ε himself/herself. The tests showed that implementation still needs improvement, as in some cases it was not possible to find an ε which led to one single cluster containing all images. But the tests also showed that in such cases, when an ε was found, the results of DBScan did improve.

Future Work
We want this approach to also be applicable for other visualisation algorithms which need parameters as input. At the current state of implementation, the concept of ParSAd could be used for different visualisation algorithms. One direction for future work could be adaption of our method for dynamic visualisations which include animation or simulation. To do so, we would have to make changes in the image generation and the clustering to handle sequences. In this case, instead of comparing single images, sequences of an arbitrary number of images rendered with a certain parameter setting could be compared and clustered. A weighting of the single *LLE* coordinates of every single image in the sequence could be used as measurement for the distance between two sequences. Again, the cluster centres are presented to the user in a grid interface. Instead of presenting a static single image, a sequence which could be played when hovering over the centres is displayed to the user.

Single techniques in our approach can still be improved, including the automatic detection of ε for the DBScan algorithm. In our actual implementation the DBScan algorithm found an adequate ε for most of the tested settings.One possibility to improve the method is the use of a better heuristic to determine the first valley in the k distance graph. If it is still not possible with a better heuristic to decrease the number of noise images, another potential improvement is assigning them to the closest cluster, even if this cluster is not reachable in ε distance.This could lead to images being displayed which look different from the previously chosen ones when the assigned images to the chosen cluster centre are again clustered. Second the kMeans++ clustering algorithm could be improved by applying the images - which lay on or close to the border between two clusters - to the correct cluster. There was no solution in the implementation which correctly detects them.One possibility to solve this problem is the detecting and copying of the border images and applying them to both clusters where the could potentially belong.

The improvement of the refinement process of the parameters is also another direction for future work. Instead of only displaying the cluster centres to the user, it is possible to take the actual parameters of the chosen centre and generate a short set of three to five images by only varying the low-level parameters of this centre +/- 10%. This set is shown in addition to the cluster centres to the user and from here he/she can choose images. By choosing one of the additional images, a re-clustering could be done by finding images from the complete set of images which looking similar to the chosen one in the sequence.
Another direction could be to improve the implementation by making the high-level parameter editor more comprehensible for the non-expert user by showing the influence of every different low-level parameter in a pre-rendering when it is selected [41]. If a second one is chosen by the user the influence of both in combination can then be shown to the user. This could help a non-expert user to define his/her own high-level parameters. Finally the optimisation of the parallel distance calculation between two images and the general image handling to make the pre-processing step even faster could be another direction for future work.

Bibliography

[1] A Johannes Pretorius, M-AP Bray, Anne E Carpenter, and Roy A Ruddle. Visualization of parameter space for image analysis. In *Visualization and Computer Graphics, IEEE Transactions on*, volume 17 (12), pages 2402–2411. IEEE Press, 2011.

[2] Kwan-Liu Ma. Visualizing visualizations - user interfaces for managing and exploring scientific visualization data. In *Computer Graphics and Applications, IEEE Transactions on*, volume 20 (5), pages 16–19. IEEE Press, 2000.

[3] S. Bhagavatula, P. Rheingans, and M. desJardins. Discovering high-level parameters for visualization design. In *Proceedings of the Seventh Joint Eurographics / IEEE VGTC Conference on Visualization*, pages 255–262. Eurographics Association, 2005.

[4] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[5] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96 (34), pages 226–231. AAAI Press, 1996.

[6] Sam T Roweis and Lawrence K Saul. Nonlinear dimensionality reduction by locally linear embedding. In *Science*, volume 290 (5500), pages 2323–2326. American Association for the Advancement of Science, 2000.

[7] Lawrence K Saul and Sam T Roweis. An introduction to locally linear embedding. *Unpublished. Available at: https://www.cs.nyu.edu/˜roweis/lle/publications.html. Accessed on 23.11.2012*, 2000.

[8] Stefan Bruckner, Ivan Viola, and Eduard Gröller. Volumeshop: Interactive direct volume illustration. In *ACM SIGGRAPH 2005 Sketches*, page 60. ACM, 2005.

[9] Stefan Bruckner and Eduard Gröller. Volumeshop: An interactive system for direct volume illustration. In *Proceedings of IEEE Visualization 2005*, pages 671–678. IEEE Press, 2005.

[10] Jarke J van Wijk and Cornelius WAM van Overveld. Preset based interaction with high dimensional parameter spaces. In *Data Visualization*, pages 391–406. Springer, 2003.

[11] Stefan Bruckner and Torsten Möller. Result-driven exploration of simulation parameter spaces for visual effects design. In *Visualization and Computer Graphics, IEEE Transactions on*, volume 16 (6), pages 1468–1476. IEEE Press, 2010.

[12] Christof Rezk Salama, Maik Keller, and Peter Kohlmann. High-level user interfaces for transfer function design with semantics. In *IEEE Transactions on Visualization and Computer Graphics*, volume 12 (5), pages 1021–1028. IEEE Press, 2006.

[13] Stefan Bruckner, Christoph Heinzl, and Eduard Gröller. Session 6: Closing session. *http://www.cg.tuwien.ac.at/research/publications/2012/VisWeek-Tutorial-2012-Uncertainty/Session-S6.html*, Accessed on 28.11.2014, 2012.

[14] Christoph Heinzl, Stefan Bruckner, Eduard Gröller, Alex Pang, Hans-Christian Hege, Kristin Potter, Rüdiger Westermann, Tobias Pfaffelmoser, and Torsten Möller. Ieee visweek 2012 tutorial on uncertainty and parameter space analysis in visualization. *http://www.cg.tuwien.ac.at/research/publications/2012/VisWeek-Tutorial-2012-Uncertainty/*, Accessed on 28.11.2014, 2012.

[15] Alex T. Pang. Session 1: Introductory session. http://www.cg.tuwien.ac.at/research/publications/2012/VisWeek-Tutorial-2012-Uncertainty/Session-S1.html, Accessed on 25.11.2014, 2012.

[16] Hans-Christian Hege. Session 2: Uncertainty modeling. *http://www.cg.tuwien.ac.at/research/publications/2012/VisWeek-Tutorial-2012-Uncertainty/Session-S2.html*, Accessed on 27.11.2014, 2012.

[17] Joe Marks, Brad Andalman, Paul A Beardsley, William Freeman, Sarah Gibson, Jessica Hodgins, Thomas Kang, Brian Mirtich, Hanspeter Pfister, and Wheeler Ruml. Design galleries: A general approach to setting parameters for computer graphics and animation. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 389–400. ACM, 1997.

[18] Margaret Burnett, Andrei Sheretov, and Gregg Rothermel. Scaling up a what you see is what you test methodology to spreadsheet grids. In *Proceedings of the IEEE Symposium on Visual Languages*, VL '99, pages 30–37. IEEE Press, 1999.

[19] TJ Jankun-Kelly and Kwan-Liu Ma. A spreadsheet interface for visualization exploration. In *Proceedings of the Conference on Visualization '00*, pages 69–76. IEEE Press, 2000.

[20] Kwan-Liu Ma. Image graphs - a novel approach to visual data exploration. In *Proceedings of the Conference on Visualization '99: Celebrating Ten Years*, pages 81–88. IEEE Press, 1999.

[21] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *Proceedings of the conference on Visualization'01*, pages 255–262. IEEE Press, 2001.

[22] Fernando Vega Higuera, Natascha Sauber, Bernd Tomandl, Christopher Nimsky, Guenther Greiner, and Peter Hastreiter. Automatic adjustment of bidimensional transfer functions for direct volume visualization of intracranial aneurysms. In *Medical Imaging 2004*, pages 275–284. International Society for Optics and Photonics, 2004.

[23] Andreas König and Eduard Gröller. Mastering transfer function specification by using volumepro technology. In *Spring Conference on Computer Graphics*, volume 17 (4), pages 279–286, 2001.

[24] Leopold Kühschelm. Advanced image-based transfer function design. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2005.

[25] Andreas Opitz. Classification and visualization of volume data using clustering. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2009.

[26] Pavel Berkhin. Survey of clustering data mining techniques. *http://www.cc.gatech.edu/ isbell/reading/papers/berkhin02survey.pdf*, Accessed on 15.05.2014, 2002.

[27] Mihael Ankerst, Stefan Berchtold, and Daniel A. Keim. Similarity clustering of dimensions for an enhanced visualization of multidimensional data. In *Proceedings of the 1998 IEEE Symposium on Information Visualization*, INFOVIS '98, pages 52–60. IEEE Press, 1998.

[28] H. Sebastian Seung and Daniel D. Lee. The manifold ways of perception. In *Science*, volume 290 (5500), pages 2268–2269, 2000.

[29] Penny Rheingans and David Ebert. Volume illustration: Non-photorealistic rendering of volume models. In *Proceedings of the Conference on Visualization '00*, pages 195–202. IEEE Press, 2000.

[30] John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. In *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, volume 28 (1), pages 100–108. JSTOR, 1979.

[31] NVIDIA Cooperation. Cuda toolkit. https://developer.nvidia.com/cuda-toolkit. Accessed on 12.02.2013.

[32] EM Photonics. Cula tools. http://www.culatools.com, Accessed on 18.10.2013.

[33] Mark Harris. Optimizing parallel reduction in cuda. In *NVIDIA Developer Technology*, volume 2 (4). Nvidia Corporation Santa Clara, CA, USA, 2007.

[34] Gaël Guennebaud and Benoît Jacob. Eigen v3. *http://eigen.tuxfamily.org*, Accessed on 12.01.2013.

[35] Johannes Kastner and Eduard Gröller. stagbeetle832x832x494.dat. http://www.cg.tuwien.ac.at/research/publications/2005/dataset-stagbeetle/dataset-stagbeetle-832x832x494.zip, Accessed on 12.02.2013, 2005.

[36] Viatronix Inc. Michael Meißner. stent16.raw. http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/data/stent16.raw.zip, Accessed on 25.02.2013.

[37] skewed_head.dat. http://www.cg.tuwien.ac.at/courses/Visualisierung/data/skewed_head.zip, Accessed on 12.02.2013.

[38] Viatronix Inc. Kevin Kreeger. backpack16.raw. http://www.gris.uni-tuebingen.de/edu/areas/scivis/volren/datasets/data/backpack16.raw.zip, Accessed on 12.02.2013.

[39] Andrés Álvarez-Meza, Juliana Valencia-Aguirre, Genaro Daza-Santacoloma, and Germán Castellanos-Domínguez. Global and local choice of the number of nearest neighbors in locally linear embedding. In *Pattern Recognition Letters*, volume 32 (16), pages 2171–2177. Elsevier, 2011.

[40] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. In *IEEE Transactions on Image Processing*, volume 13 (4), pages 600–612. IEEE Press, 2004.

[41] Marius Gavrilescu, Muhammad Muddassir Malik, and Eduard Gröller. Custom interface elements for improved paramter control in volume rendering. In *14th Int. Conf. on System Theory and Control 2010*, pages 219–224, 2010.