FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Animated Transitions Across Multiple Dimensions for Volumetric Data

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Informatik

eingereicht von

Christian Basch
Matrikelnummer 0026388

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Mitwirkung: Dipl.-Ing. Dr.techn. Peter Rautek

Wien, October 17, 2011

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Animated Transitions Across Multiple Dimensions for Volumetric Data

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Science

by

Christian Basch
Registration Number 0026388

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.-Prof. Dipl.-Ing. Dr.techn. Eduard Gröller
Assistance: Dipl.-Ing. Dr.techn. Peter Rautek

Vienna, October 17, 2011 _____ _____
(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Christian Basch
Barichgasse 21/19, 1030 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 Verfasser)

Acknowledgements

First and foremost I want to thank my supervisors Peter Rautek and Eduard Gröller for their patience and valuable input. Furthermore, I would like to thank my parents for their financial and moral support, and my girlfriend for her constant encouragement and persistence. Without them I would not have been able to finish my studies. Thank you!

Abstract

There are several techniques, that can be used to visualize volumetric data. A data set can be illustrated using slicing (depicting arbitrary slices through the volume), direct volume rendering (DVR), or in a more abstract way, histograms and scatter plots. Usually these different methods of visualization are being applied separately. To recognize coherencies between the representations, methods based on Linking and Brushing can be utilized. These methods highlight voxels in one view, as soon as they are selected in another one. Coming from scientific visualization, these methods are very useful, when selecting voxels from 2D data representations, like scatter plots. Of course they are less useful, when trying to select voxels directly from the volume. Therefore this thesis explored methods, that are not based on selection and highlighting. Rather, the correlation between different representations is shown by moving voxels between different volume representations. As a basis, methods like staggered animation, acceleration, and deceleration were adopted, which had been previously used in the graphical analysis of statistical data.

Kurzfassung

Zur Visualisierung volumetrischer Daten stehen zahlreiche Möglichkeiten zur Verfügung. Ein Datensatz kann mittels Slicing (Darstellung von einzelnen Schnitten durch das Volumen), Direktem Volumenrendering (DVR), oder auf abstraktere Weise mittels Histogrammen oder Scatterplots dargestellt werden. Üblicherweise werden diese verschiedenen Visualisierungsmethoden getrennt voneinander angewendet. Um Zusammenhänge zwischen den einzelnen Ansichten zu erkennen, stehen Linking und Brushing Methoden zur Verfügung. Dabei werden Elemente in einer Ansicht hervorgehoben, wenn sie in einer anderen Ansicht ausgewählt wurden. Aufgrund ihres Ursprungs in wissenschaftlicher Visualisierung, ist diese Methode sehr gut auf 2D Datenrepräsentationen, wie Scatterplots, anwendbar. Allerdings ist sie weniger nützlich, wenn die Auswahl der Voxel direkt am Volumen durchgeführt wird. Aus diesem Grund wurden im Rahmen dieser Diplomarbeit Möglichkeiten untersucht, die Zusammenhänge zwischen den unterschiedlichen Darstellungsformen aufzeigen, ohne eine Auswahl an Voxeln treffen zu müssen. Dies wurde realisert, indem Voxel von einer Repräsentation in die andere mittels animierter Übergänge wechseln. Dafür wurden Methoden wie versetzte Animationen und Be- und Entschleuningung adaptiert, die bereits bei der grafischen Auswertung statistischer Daten angewendet wurden.

Contents

CHAPTER 1

Introduction

According to Kosara [15], visualization needs to meet certain criteria, to actually be called visualization. It must be (a) *based on (non-visual) data*, (b) *produce an image* and (c) *the result must be readable and recognizable*. The purpose of visualization is to present data that is abstract or at least not immediately visible (e.g., the inside of a human body). Usually this data is produced by scientific simulations or measurements. This work focuses on volumetric medical data, but is also applicable to any grid based volumetric data. In practice volumetric data consists of volume elements (voxels), that are arranged on a 3D equidistant grid. Each voxel has two attributes: position and intensity. Additionally, for each voxel, a surface normal might be approximated by calculating the gradient from neighboring voxels. The gradient is a 3D vector, that points in the direction of the greatest rate of increase. Using these attributes, volumetric data can be visualized using several representations. The most common form of visualization are slices, that are perpendicular to one of the major axes. These are used in medical diagnosis. Furthermore, 3D renderings are used to literally gain insight into volumetric data. Besides slicing and 3D renderings, there are more abstract representations, like statistics and histograms. Histograms strip the voxels off their grid coordinates and use one or both of the remaining attributes to accumulate voxels in classes. Often used histograms include 1D intensity histogram, 1D gradient magnitude histogram, and a 2D intensity/gradient magnitude histogram. These representations have advantages in different use cases, and when combined correctly can provide a very powerful tool for volume exploration. It is difficult for users to establish a connection between these representations, when they are just presented side by side. In addition to exploring each representation on its own, a user must be able to discover correlations among them. This helps users to quickly gain insight into the composition of the volumetric data set, and to identify areas of interest. For this purpose *Linking and Brushing* techniques can be used, where voxels that are selected in one view also get highlighted in the other ones.

Linking and Brushing is a technique heavily used in 2D data representations. Applied to volumetric data, it is very useful when selecting voxels in the 2D histogram. If the user wants to use Linking and Brushing to pick voxels from the volume, less intuitive techniques like 3D brushes need to be used. Our goal was to allow the user to intuitively and interactively explore the volume. Inspired by the work of Heer and Robertson [14], who explored the usefulness of animation in statistical data graphics, we wanted to extend the idea of Linking and Brushing. Instead of highlighting voxels, we move them from one representation to the other one. Therefore the paths described by the voxels need to be calculated. Furthermore the timing of the animation is essential. Letting all voxels start at the same time, makes the animation unrecognizable. Staggering the starting and ending times is an effective way to prevent extensive overlap by reducing the number of simultaneously moving objects. A hybrid volume renderer, consisting of a raycaster and a volume splatter, was necessary to produce appealing renderings while keeping the computational cost at a minimum.
Chapter 2 describes the state-of-the-art of volume rendering methods and the underlying optical model. Chapter 3 takes a look at related work. Chapter 4 illustrates the hybrid volume renderer. The particulars on voxel trajectory calculation and animation timing can be found in Chapter 5. Chapter 6 describes the details of the implementation, and its results are then discussed in Chapter 7. Finally, the contents of this diploma thesis are summarized in Chapter 8.

CHAPTER 2

State-Of-The-Art in Volume Rendering

This chapter gives a brief overview of state-of-the-art volume rendering techniques and their common optical model.

2.1 Optical Model for Volume Rendering

Optical models for volume rendering can be best explained, by looking at the volume as a cloud of particles [26]. These particles can either absorb or scatter light coming from a source. Models that take into account absorption, emission, scattering and shadowing tend to be very complicated, which is why practical models are simplified in several ways. A common approximation for the volume rendering integral is given by [27]:

$$I_\lambda(x,r) = \int_0^L C_\lambda(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds \tag{2.1}$$

Here, I_λ is the amount of light of wavelength λ coming from the direction of ray r that is received at location x on the image plane. L is the length of the ray r and μ is the density of volume particles that receive light from the light sources and reflect it towards the observer according to their material properties. C_λ is the light of wavelength λ reflected and/or emitted at location s in the direction of r. The equation takes emission and absorption effects into account, but discards more advanced effects such as scattering and shadowing.
In general, Equation 2.1 cannot be computed analytically. Hence, practical volume rendering algorithms discretize Equation 2.1 into sequential intervals i of size Δs. The result is the common compositing equation:

$$I_\lambda(x,r) = \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i) \cdot \prod_{j=0}^{i-1}(1-\alpha(s_j)) \tag{2.2}$$

For each interval i along a ray r the volume density is classified, via transfer functions C_λ and α, which respectively assign color and opacity to each sample value s_i in the volume.

2.2 Volume Rendering Techniques

A three dimensional grid $V : \mathbb{Z}^3 \mapsto \mathbb{R}$ is considered to represent a spatial density function $f : \mathbb{R}^3 \mapsto \mathbb{R}$ sampled at regular grid points, where the density samples are defined as [7]:

$$V_{i,j,k} = f([x_i, y_j, z_k]) \tag{2.3}$$

These density samples are also referred to as *voxels*, which is an abbreviation for volume elements. Volumetric data can be processed for viewing in two fundamentally different ways. Using *indirect* volume rendering the volumetric data is first converted into an intermediate representation (e.g. via Marching Cubes [23]), and is subsequently rendered with polygon rendering hardware. Contrarily, methods that assign optical properties directly to the volume elements (voxels) without generating any intermediate representation are referred to as *direct* volume rendering techniques. Since this work makes use of direct volume rendering techniques exclusively, the remainder of this chapter will focus on them. The direct techniques are a robust and very flexible way of displaying volumes, allowing us to illustrate the interior of volumes as well. Although there is no need for an intermediate representation, they are still computationally expensive due to the tremendous number of voxels which have to be processed. The direct volume rendering techniques can be further subdivided into two categories. The image-order methods produce the image by casting rays through each pixel of the viewing plane and the color of the pixels is determined by the contributing voxels. In contrast, the object-order methods process the volume voxel-by-voxel and project them on the image plane. Each approach has its advantages and drawbacks, which will be discussed in the remainder of this chapter.

2.2.1 Image-Order Volume Rendering

Using the image-order approach, the computation emanates from the output image, and not from the object as it is the case with object-order approaches. For each pixel of the output image the contributing data samples are determined. The most commonly used image-order algorithm is raycasting, introduced by Levoy et. al. [19]. This approach

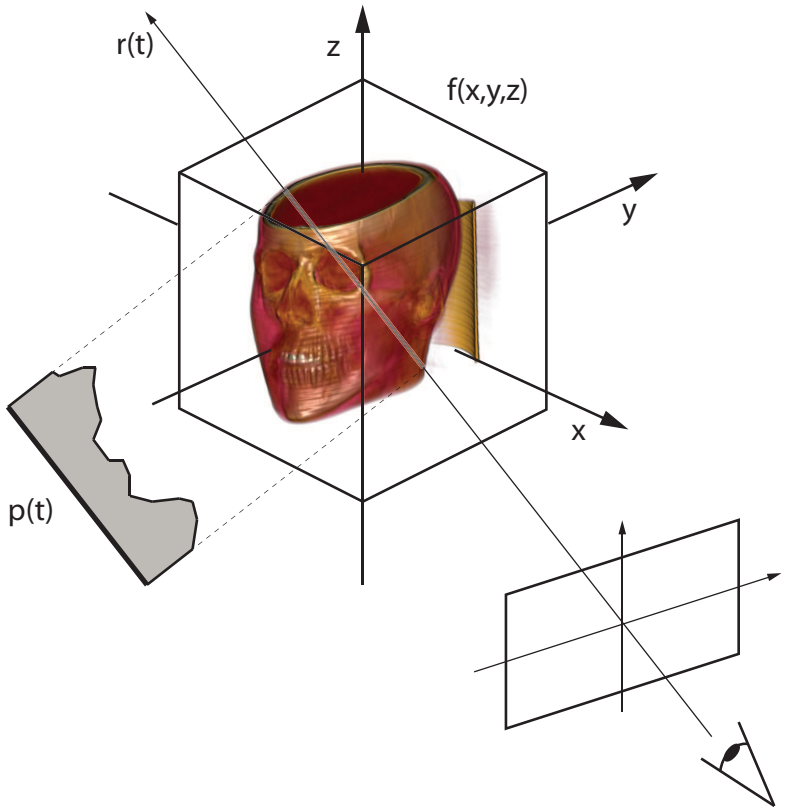


Figure 2.1: Illustration of a ray and its corresponding reconstructed density function $p(t)$ [7].

obtains the pixel values by casting rays through the viewing plane and the volume. These rays can be defined by their parametric equation [7]

$$\mathbf{r}(t) = \mathbf{x} + \omega \cdot t \tag{2.4}$$

where **x** is the origin, ω the unit direction and t is the ray parameter. Each ray intersecting the volume has a specific density profile p, which depends on the density values along the ray segment lying inside the volume. This density profile can be expressed by combining Equation 2.3 and Equation 2.4, yielding:

$$p(t) = f(\mathbf{r}(t)) = f(\mathbf{x} + \omega \cdot t) \tag{2.5}$$

Figure 2.1 illustrates a ray and its corresponding reconstructed density profile. The three-dimensional volume function is reconstructed from the samples taken at discrete intervals along the ray and by evaluating the optical model for them. If the colors and opacities are composited in front-to-back order, a ray can be terminated, as soon as full opacity is reached. This is one of the main advantages of raycasting, because it

avoids processing of occluded regions. Medical datasets usually contain a large number of voxels that do not contribute to the resulting image (i.e., voxels that are classified as transparent by the transfer function). Efficiently skipping these non-contributing regions is one challenge in raycasting and has a major impact on performance. Due to the large number of voxels to be processed and the high computational complexity, several performance improvements have been proposed over the years:

Empty Space Skipping

Researchers [20, 40] have reported that in typical volumes 70-95% of the voxels are classified as transparent by the transfer function. Hence skipping these non-contributing regions has a major performance impact. Levoy [20] introduced a hierarchical space skipping method using a binary pyramid, that encodes empty and non-empty space. Here raycasting starts at the top level of the pyramid. Whenever a non-empty cell is encountered, the algorithm descends one level, entering whichever cell encloses the ray's current location. Otherwise the ray is forwarded to the intersection point with the next cell on the same level. This idea can been taken further by generating a min-max octree based on the volume's data values. Whenever the classification changes, this octree can be used to efficiently rebuild the binary pyramid.

Adaptive sampling

Volumetric datasets contain regions of identical or similar values. Avoiding sampling inside those regions, is one way to speed up raycasting and has been proposed by Wal-sum et al. [42]. Basically, a ray starts sampling the volume at large intervals and compares each sample to the previous one. If the difference of the values of two adjacent samples is beyond a certain threshold, additional samples are taken. This idea can also be extended to regions with low opacity and as a result minor contribution to the final image.

Inter-Frame Coherency

When interactively viewing a volume, the difference between two consecutive frames is usually very small. This fact is being exploited by the C-Buffer technique, proposed by Yagel and She [49], which for each pixel stores the first non-empty voxel hit by the corresponding ray. When the viewing parameters are changed, this information is used to estimate the initial position of a ray in the subsequent frame, by transforming the C-Buffer accordingly.

Adaptive refinement

Proposed by Levoy [21], this technique significantly reduces computational effort by casting rays only from a subset of pixels of the output image. The values of pixels, for which no ray was cast from, are interpolated from neighboring pixels. Adaptive refinement is an effective technique to ensure interactivity. While dragging the mouse, only part of the pixels is rendered and the remainder is interpolated. As soon as the interaction stops, the interpolated pixels get rendered as well. This technique exploits the high coherency between pixels of the output image. For example, it is highly probable, that between two pixels of similar color, another pixel of a similar color can be found.

Efficient Memory Access

The way large datasets are stored and accessed in memory has a large performance impact. The simplest way to store the volume is a three dimensional array. Due to view-dependent memory access patterns, this leads to variations in rendering times, when changing the viewing parameters. A way to circumvent these limitations is the usage of a storage scheme named bricking [33]. Here the volume data is stored in equally-sized blocks, which reduces the view-dependent performance variations without increasing the memory footprint. A similar storage scheme has been used in the raycasting technique develpoed by Law and Yagel [18]. Here, each ray is linked to the cell it initially enters and also to a list of rays waiting for this cell to become active. Only one block is active at a time. All rays waiting for the active cell are advanced until they exit the cell. Since a block is only active once, this approach effectively avoids cache thrashing.

2.2.2 Object-Order Volume Rendering

Opposed to image-order techniques, object-order methods determine, how each data sample affects the pixels on the image plane. In its simplest form, an object-order algorithm loops through the data samples, projecting each sample onto the image plane.

Splatting, introduced by Westover [47], is a technique that traverses the volume and projects footprints (known as splats) onto the image plane. Voxels with zero opacity can be skipped, as they do not contribute to the final image. This is one of the greatest advantages of splatting, as it can dramatically reduce the number of voxels that have to be processed. Using orthographic projection, all the kernels have the same projection or footprint. Thus, the footprint can be pre-computed once and used for the projection of all the voxels. Perspective projection requires the footprints to be distorted according to the distance of the voxels to the observer. In the original approach of the algorithm, all the voxels are splatted directly onto the final image. This is why the algorithm is known as *composite-every-sample*. This method may cause color bleeding and sparkling artifacts, because the visibility ordering of splats is imperfect.

To increase image quality Westover [47] proposed *object-space sheet-buffer splatting*. This method uses three stacks of volume slices, one for each major axis. Here voxel kernels are summed up within the slices of the stack most parallel to the image plane. Those slices are then composited to the final image. This approach indeed corrects color bleeding but it also introduces noticeable popping artifacts when the camera moves around the volume. This happens because the samples along rays may not be aligned anymore, after a small change in viewing angle leads to a change of the slicing direction.

Mueller and Crawfis [29] introduced a method which eliminates these drawbacks and also enhances the approximation of the light transport inside voxels: *image-space sheet-buffer splatting*. In their approach voxel kernels are processed within slices parallel to the image plane. Therefore, voxels can contribute to more than one sheet. All voxel kernels that overlap a slab (region between two slices) are clipped to the slab and summed into a sheet buffer. Once a sheet buffer has received all contributions, it is composited with the current image, and the slicing slab is advanced forward.

In the image-space sheet-buffer splatting [31], early splat elimination is possible in front-to-back composition by subdividing the image into small tiles and avoiding to splat voxels that cover tiles that have already reached the maximum opacity. However, the projection transformation still has to be performed for these voxels, which makes this optimization less effective than early ray termination in raycasting.

Vega-Higuera et al. [43] proposed the use of *point sprites* to render neurovascular data. This reduces the geometry needed for each voxel from four points (needed for the quads to represent splats) to one. This idea is also used in the GPU-based implementation of the image-space sheet-buffer splatting proposed by Neophytou and Mueller [32]. Their approach comprises two steps: First the density values of all voxels of a slice are projected into an auxiliary buffer using textured point sprites. Then all the pixels of the buffer are classified and shaded using a fragment shader that computes the gradient vectors at the pixels on the basis of their density central difference (see Equation 4.4). Finally, the buffer is composed into the final image.

2.2.3 Hybrid-Order Volume Rendering

Image-order and object-order algorithms have very distinct advantages and disadvantages. Therefore, some effort has been put into combining the advantages of both approaches.

Shear-warp factorization, introduced by Lacroute [17], is such an algorithm. It is considered to be the fastest software-based volume rendering algorithm and is based on a factorization of the viewing transformation into a shear and a warp transformation. The volume is sheared such that all viewing rays are parallel to the principal viewing axis in sheared-object-space. This way the volume and the image can be traversed simultaneously. Compositing is performed into an intermediate image. Since this inter-

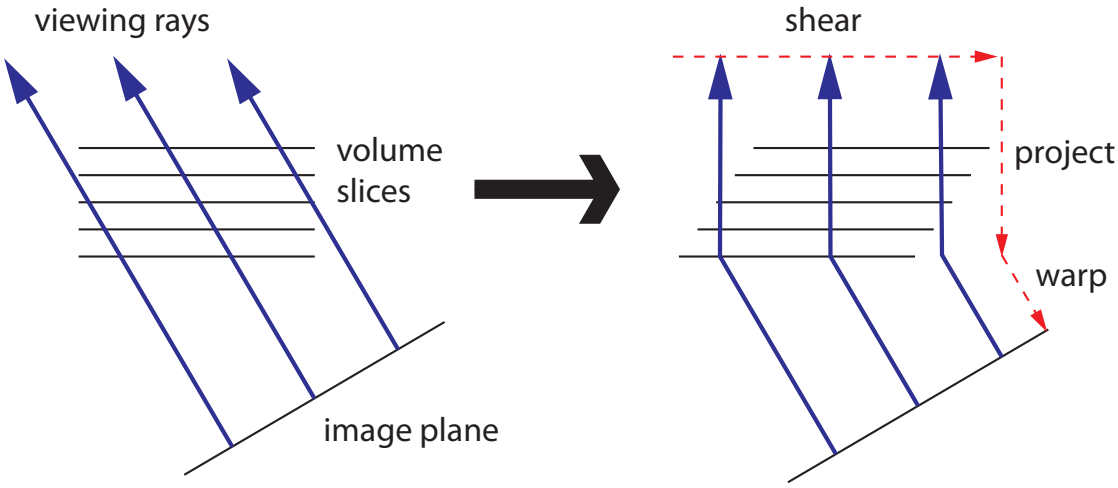


Figure 2.2: Illustration of the basic principle of the shear-warp factorization. The volume slices are sheared, such that the viewing rays become parallel to the major viewing axis. After the sheared volume has been projected onto an intermediate image, this image is then warped, yielding the final image.

mediate image is warped, a two-dimensional warp transformation is applied, producing the final image. This basic mechanism of shear-warp factorization is illustrated in Figure 2.2.

The aligned traversal between image and volume is the basis for many optimizations. A runlength-encoding of the intermediate image allows for an efficient early-ray termination approach. Additionally, runlength-encoding of the volume in each of the three major viewing directions allows skipping of transparent voxels. Furthermore, utilizing a min-max octree allows for empty space skipping. In contrast to runlength-encoding, this approach allows fast classification and does not require three copies of the volume. The problem of shear-warp factorization, however, is the low image quality caused by the use of bilinear interpolation for reconstruction and a varying sample rate that depends on the viewing direction and projection. These issues result in an inferior image quality compared to other methods, such as raycasting.

2.2.4 Texture-based Techniques

Texture slicing on programmable graphics processing units [22] is one of the predominant volume rendering techniques. One method incorporating graphics hardware is based on 2D texture mapping [37]. This method stores stacks of slices as 2D textures in graphics memory for each major viewing axis. The stack most parallel to the viewing direction is chosen and mapped on an object-aligned proxy geometry, which is then rendered in back-to-front order using alpha blending (see Figure 2.3). This approach

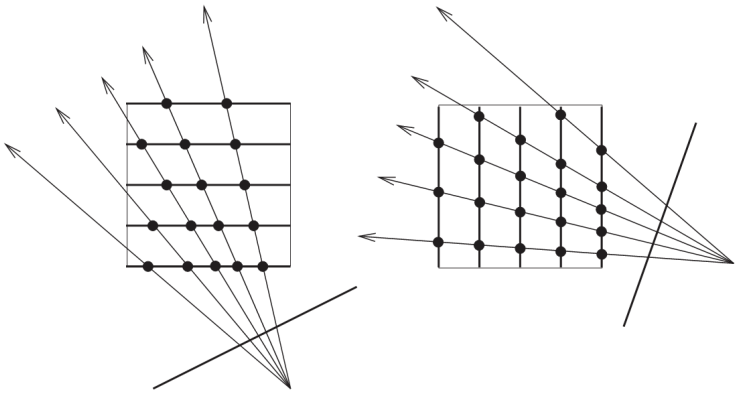


Figure 2.3: Object-aligned 2D texture slicing [34]. For each of the three major axes, a stack of 2D textures is stored. The stack most parallel to the image plane is chosen, and rendered in back-to-front order as textured quads using alpha blending.

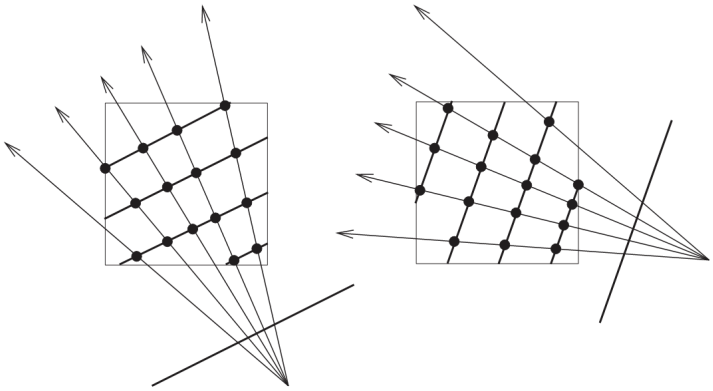


Figure 2.4: Image-aligned 3D texture slicing [34]. The volume is stored as a single 3D texture and subdivided into polygons parallel to the image plane. These polygons are rendered in front to back order, using alpha blending.

corresponds to shear-warp factorization and suffers from the same drawbacks, namely, bilinear interpolation within the slices, and varying sampling rates depending on the viewing direction.

3D texture methods use image-aligned texture slices. In order for these methods to work efficiently, the whole volume needs to be uploaded to the graphics memory as a 3D texture. Using the hardware this texture is then mapped onto polygons parallel to the viewing plane, which are then rendered in back-to-front order using alpha blending (see Figure 2.4). Contrary to 2D texture slicing, where only bilinear interpolation can be used, this method allows for trilinear interpolation supported by the graphics hardware.

A drawback of this method is that the memory of the graphics hardware must be large enough to accommodate the whole volume.

CHAPTER 3

Related Work

Often data can be visually represented in several ways. To visualize relations between these representations a technique called *Linking and Brushing* is frequently used. Brushing is used to select data and perform several operations on it (e.g., highlighting or masking). Linking propagates the selection to the other views, applying the same operation there. Brushing was explored by Becker and Cleveland [1], who developed a system which implemented masking and highlighting. However, the idea of brushing has been examined even earlier. Fisherkeller et al. [12] used the idea of interactively selecting a region in their PRIM-9 system, although they did not call it "brushing". Traditionally brushes in visualization systems are limited to two dimensions, i.e., the brushes operate in display space. To be able to apply a brush to volumetric data, a three-dimensional brush is required. Multidimensional brushes have been studied in several works [16, 25, 44]. Kosara et al. [16] applied linking and brushing to 3D scatterplots, where three dimensions are arbitrarily selected from an n-dimensional dataset. These dimensions can also include spatial coordinates. They used volume rendering to display the 3D scatter plot.

In addition to highlighting selected data points in all views, they can be connected by lines. Collins and Carpendale applied this idea in their *VisLink* system [5], where different 2D views are represented as semi-transparent planes in 3D space, arranged side by side. As soon as data is selected in one plane, the selected data is connected with the same data in adjacent planes by what the authors call *inter-plane edges*. This propagation is recursive, where the level of recursion can be interactively selected. Single source to single target edges are drawn as straight lines. Single source to many target edges are drawn using multiple curves calculated with corner cutting [8, 9].

Other research concludes that relations between representations can be conveyed by using animation. Heer and Robertson [14] used animated transitions to communicate the coherence of different representations of statistical data. They found out, that

staging and staggering the animations can greatly improve user perception. These techniques reduce occlusion, which is one of the recommendations of the authors. Other recommendations are *maximize predictability*, *use simple transitions* and *make transitions as long as needed, but not longer*. They have derived their recommendations from the *Congruence Principle* and *Apprehension Principle* postulated by Tversky et al. [41]. All these design considerations aim at reducing the cognitive load for the user.

Applying animation to volumetric data means moving individual voxels along distinct paths. Because the voxels' coordinates are constantly changing, volume splatting has to be used for rendering. Volume splatting is an object-order rendering method introduced by Westover [47], which traverses the volume voxel-by-voxel and projects each voxel onto the image plane.

CHAPTER 4

Hybrid Volume Rendering

This work employs a hybrid renderer, that uses volume splatting to render animated voxels, and raycasting to render the remainder of the volume. Using this approach, each voxel gets rendered by just one of the renderers. This means, the volume is rendered using raycasting, and splatting is applied to voxels that leave their grid position (see Figure 4.1). We implemented a simple splatter, sufficient for visualizing the voxel animations. However, to get appealing renderings of the volume, a raycaster was used. Nevertheless the principle of volume splatting will be explained in the next section, followed by a more detailed description of the raycasting algorithm.

4.1 Volume Splatting

Object-order algorithms traverse the volume voxel-by-voxel and project them onto the image plane. A more sophisticated approach, *splatting*, introduced by Westover [47], convolves every voxel in object space with a 3D reconstruction filter and accumulates the voxels' contribution on the image plane. Volume splatting is comprised of the following steps:

Volume traversal The way of traversing the volume depends on the compositing method. Simply projecting voxels onto the image plane leads to the wrong compositing order of the projected splats. Usually the volume is traversed in slices in approximate back-to-front order, similar to 2D texture slicing. For more advanced splatting methods, such as *image-aligned sheet buffers* [29], the traversal order is similar to 3D texture slicing.

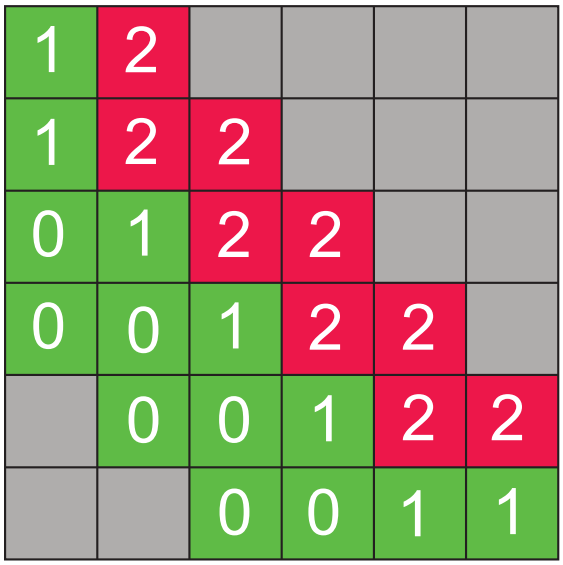


Figure 4.1: This figure shows a slice of the volume where the interpolation parameter $0 \leq t \leq Intensity_{max}$) is set to 1 and the animation is delayed by ascending intensity. This means, that voxels with lower intensity values start moving first. The numbers in the grid represent voxel intensities. Voxels depicted in green have already left their grid position and are therefore rendered by the volume splatter. Voxels depicted in red still remain at their grid positions and are rendered by the raycaster. Voxels in gray have been hidden via the transfer function and are not rendered at all.

Interpolation Splatting derives its efficiency from the use of pre-integrated reconstruction kernels. For simple splatting, the 3D kernel can be pre-integrated into a generic 2D footprint that is stored as a 2D texture. Because of its low computational cost and simplicity, this approach was chosen for this work.

Classification and Shading Typically, splatting uses pre-classification and pre-shading of the volume data. Each voxel stores the resulting RGBα values, which are then multiplied by the footprint before projection. Mueller et al. [30] proposed a method for post-classification and shading in screen space. The gradients are either projected to screen space, or they are computed in screen space using central differencing (see Equation 4.4).

Compositing Compositing is more complicated for splatting than for other volume rendering methods. The easiest compositing approach is called *composite every sample*, where the 2D footprint of the kernel is multiplied by the scalar voxel value, projected to screen space, and blended onto the image plane using graphics hardware [6]. This

leads to visible artifacts, like color bleeding from background objects due to incorrect visibility determination [46]. To remedy this drawback, Westover introduced *sheet-buffer splatting* [47]. The splats are now added to a sheet buffer, instead of getting directly composited to the final image. When all voxels of a slice have contributed to the sheet buffer, the whole sheet buffer is composited onto the image plane. This does solve the bleeding artifacts, but also leads to the same problems encountered by 2D texture slicing methods: Popping artifacts when the slice stack is suddenly changed, and inferior image quality, due to the lack of trilinear interpolation.

Mueller and Yagel [29] proposed to use *image-aligend sheet-buffers*. Here the slices are parallel to the viewing direction, and the contributions of 3D reconstruction kernels between slices are added to the sheet buffer, and the result is composited onto the image plane. This effectively resolves the popping artifacts, however, the intersection of a slice with the 3D reconstruction kernel is computationally expensive.

Since only a basic splatter was used in this work, only a brief overview of volume splatting was given. A comprehensive description of volume splatting can be found in related work [34, 47, 51].

4.2 Raycasting

The other half of the hybrid renderer, is a raycaster, that is used to render voxels, that have not moved from their volume grid positions. As soon as a voxel leaves its grid position, it gets discarded by the raycaster and is rendered by the volume splatter.

Raycasting is an image-order algorithm, that processes the output image pixel-by-pixel and casts rays from each pixel through the volume. This processing order has the disadvantage that the dataset must be traversed once for every ray, resulting in redundant computation (i.e., multiple descents of an octree), but on the other hand enables early ray termination.

4.2.1 The Raycasting Pipeline

Using raycasting the volume is rendered by casting rays from the viewing plane through the volume and sampling it at discrete intervals along the ray. For each ray intersecting the volume, four steps are performed at each sampling position. **Reconstruction** is the process of reconstructing a continuous function from the discrete dataset. This step is necessary, since the volume might be sampled at any position. The **classification** step assigns material properties like color and opacity to the sample obtained by the reconstruction step. The evaluation of the illumination model afterwards is referred to as **shading**. This step often involves the computation of a gradient. **Compositing** determines the contribution of the previously classified and shaded sample to the final image.

The findings of Bruckner [3, p. 19-22] and Engel et al. [11] suggest, that the best results are achieved by the sequence *Reconstruction-Classification-Shading-Compositing*. Therefore this setup is applied in this work. The following sections will describe the steps of the rendering pipeline in more detail.

Reconstruction

In order to be able to sample the volume at arbitrary positions, a continuous volume function $f\colon \mathbb{R}^3 \mapsto \mathbb{R}$ needs to be reconstructed.

Function Reconstruction A point sample can be represented as a scaled Dirac-pulse function. Sampling a signal is equivalent to multiplying it by a grid of Dirac-pulses, one at each sample point, as shown in Figure 4.2 [24]. The Fourier transform of a two dimensional grid of Dirac-pulses, with frequency f_x in x and f_y in y is itself a grid of impulses with period $1/f_x$ in x and $1/f_y$ in y. If we call the grid of Dirac-pulses $k(x, y)$ and the signal $g(x, y)$, then the Fourier transform of the sampled signal, $\widehat{gk}$, is $\hat{g} * \hat{k}$. Since k is a grid, convolving $\hat{g}$ with $\hat{k}$ amounts to duplicating $\hat{g}$ at every point of $\hat{k}$, producing the spectrum shown at the bottom right in Figure 4.2. The copy of $\hat{g}$ centered at zero is the primary spectrum, and the other copies are called *alias spectra*. If $\hat{g}$ is zero outside a given region the signal is called band limited. The alias spectra of a band limited function do not overlap each other if the sampling frequency is chosen high enough (i.e., the Dirac pulses of $\hat{k}$ are sufficiently far away from each other). Then $\hat{g}$ can be recovered by multiplying $\widehat{gk}$ by a box function $\hat{h}$ which is one in the Nyquist region and zero elsewhere. Such a multiplication is equivalent to convolving the sampled data gk with h, the inverse transform of $\hat{h}$. This convolution with h allows us to reconstruct the original signal g by removing, or filtering out, the alias spectra, so we call h a *reconstruction filter*.

Thus, the goal of reconstruction is to extract the primary spectrum and to suppress the alias spectra. Since the primary spectrum comprises the low frequencies, the reconstruction filter is a low-pass filter. The simplest region to which $\hat{g}$ could be limited is the region of frequencies that are less than half the sampling frequency along each axis. This limiting frequency is called the Nyquist frequency and the region the Nyquist region. An ideal reconstruction filter can then be defined to have a Fourier transform that has the value one in the Nyquist region and zero outside it. The inverse Fourier transform of such a box function is the *sinc* function.

Extending the above to three-dimensional signals encountered in volume rendering, the sampling grid becomes a three-dimensional lattice and the Nyquist region a cube. Given this Nyquist region, the ideal convolution filter is the product of three sinc functions:

$$h_I(x, y, z) = (2f_N)^3 sinc(2f_N x) sinc(2f_N y) sinc(2f_N z) \tag{4.1}$$

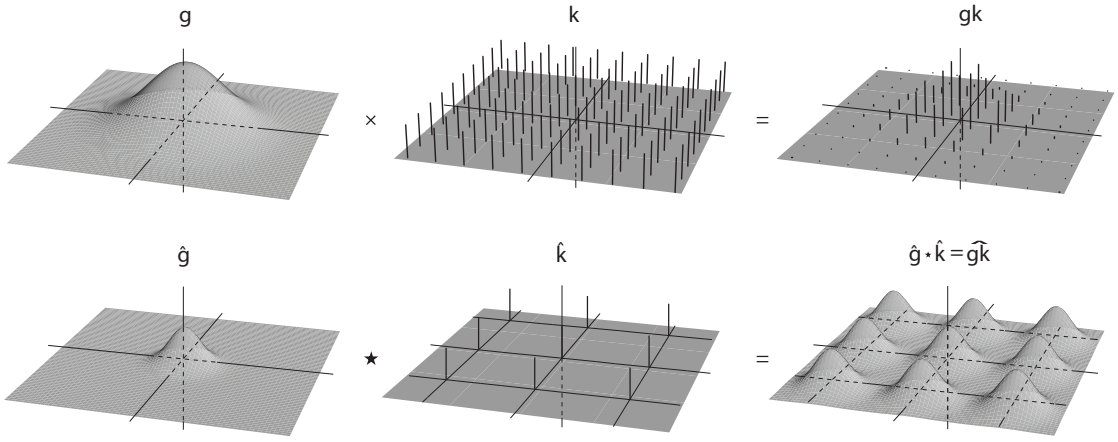


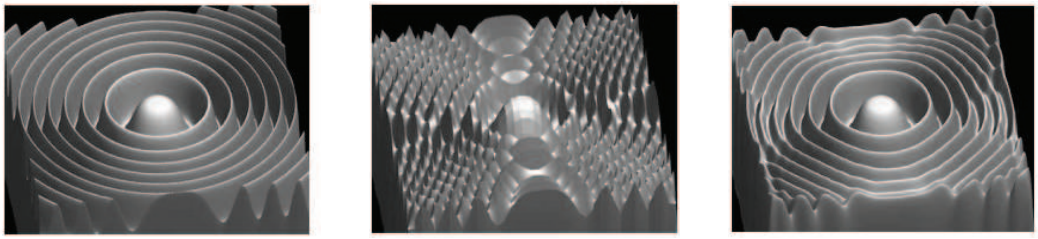
Figure 4.2: Two-dimensional sampling in the spatial domain (top) and the frequency domain (bottom) [24].

Here, f_N is the three-dimensional Nyquist region. Thus, in principle, a volume signal can be exactly reconstructed from its samples by convolving it with h_I. In practice, however, h_I cannot be implemented, because it has infinite extent in the spatial domain. Hence practical reconstruction filters will inevitably introduce artifacts into the reconstructed function. A practical filter takes a weighted sum of a limited number of samples to reconstruct a point. In other words, only samples inside a finite region are taken into account. This region is called the region of support. Filters with a larger region of support have to weight more samples and are computationally more expensive since more samples have to be processed.

The simplest interpolation function is the nearest neighbor function, which returns the value of the sample closest to a given location. Let the point P lie within a cubic cell at location $(x_P, y_P, z_P)^T$. The sample values at the eight corners of this cell are denoted as $S(0,0,0) \ldots S(1,1,1)$. Using the nearest neighbor function the value v_P at location P is given by:

$$v_P = S(round(x_P), round(y_P), round(z_P)) \tag{4.2}$$

The most common interpolation function is the trilinear interpolation function, which is a convex combination of the surrounding 8 samples. This function assumes, that the value varies linearly along each major axis. According to trilinear interpolation, the value v_P at location P is then:



(a) original signal (b) trilinear interpolation (c) windowed sinc filter

Figure 4.3: The original signal (a), reconstructed with trilinear interpolation (b) and a windowed sinc filter (c).

$$
\begin{aligned}
v_P = \; & S(0,0,0) \cdot (1-x_P) \cdot (1-y_P) \cdot (1-z_P) + \\
& S(1,0,0) \cdot x_P \cdot (1-y_P) \cdot (1-z_P) + \\
& S(0,1,0) \cdot (1-x_P) \cdot y_P \cdot (1-z_P) + \\
& S(1,1,0) \cdot x_P \cdot y_P \cdot (1-z_P) + \\
& S(0,0,1) \cdot (1-x_P) \cdot (1-y_P) \cdot z_P + \\
& S(1,0,1) \cdot x_P \cdot (1-y_P) \cdot z_P + \\
& S(0,1,1) \cdot (1-x_P) \cdot y_P \cdot z_P + \\
& S(1,1,1) \cdot x_P \cdot y_P \cdot z_P
\end{aligned}
\tag{4.3}
$$

Marschner and Lobb [24] have examined various reconstruction filters (see Figure 4.3). The best results were achieved with windowed sinc filters. While providing superior reconstruction quality, they are also about two orders of magnitude more expensive than trilinear interpolation. Therefore, when interactivity is required, the trilinear reconstruction is often the preferred method, despite its worse quality.

Gradient reconstruction In addition to the continuous volume function the reconstruction of its first derivative, called the gradient, is also necessary. Since it is an approximation of the normal of an iso-surface, it can be used for the illumination model. The quality of the gradient estimation has considerable impact on the quality of the rendered image. The ideal gradient reconstruction filter is the *cosc* filter, which is the derivate of the *sinc* filter, discussed in the previous section as the ideal reconstruction filter. As with the sinc filter, the cosc filter can not be used as a reconstruction filter due to its infinite extent in the spatial domain.

According to Möller et al. [28] there are four different methods of computing the gradient:

Derivative First (DF) Using this method the derivative is determined by first computing the normals at the grid points and then interpolating these normals.

Interpolation First (IF) The derivative at a ray sample location is calculated from a set of additionally interpolated samples in the neighborhood of the sample location.

Continuous Derivative (CD) This approach uses a derivative filter that is pre-convolved with the interpolation filter. The gradient at the sample location is computed by convolving the volume by this combined filter.

Analytic Derivative (AD) This approach is similar to CD, except the derivative filter is analytically derived from the interpolation filter.

Furthermore, Möller et al. prove that DF, IF and CD are numerically equivalent, and that AD delivers bad results in some cases. According to them, the CD method is more of theoretical interest and they used it mainly for analysis of the normal estimation process. This leaves the DF and IF method for consideration. Although they have shown, that the IF method generally outperforms the DF approach, Bruckner [3] pointed out that with an expensive gradient estimation method the DF method is preferable. Pre-computing the gradients at the grid locations would reduce the computational effort, but also increase the amount of memory needed by three times the size of the volume. Since the hybrid renderer is already a memory-intensive approach, the gradients are rather computed on-the-fly for each cell from the *central differences*, which are given by

$$\nabla f(x,y,z) \approx \frac{1}{2} \cdot \begin{bmatrix} (f(x+t,y,z) - f(x-t,y,z)) \\ (f(x,y+t,z) - f(x,y-t,z)) \\ (f(x,y,z+t) - f(x,y,z-t)) \end{bmatrix}, \quad t = 1 \tag{4.4}$$

where $f(x,y,z)$ is the 3D density function. Trilinear interpolation is then used to calculate the function value and gradient at each resample location.

Classification

Classification is the process of assigning a color and opacity to a reconstructed function value. Transfer functions, usually implemented as lookup tables, are used for this mapping. During rendering, the lookup tables containing color and opacity values are indexed by the reconstructed function value. Levoy first suggested the use of one-dimensional piecewise linear transfer functions [19]. He also used the gradient magnitude for opacity modulation, to enhance regions with high gradients and reduce the opacity of homogeneous regions.

Shading

Although having no physical significance, the Phong illumination model [35] is still common in computer graphics. It is a local illumination model, which only takes direct

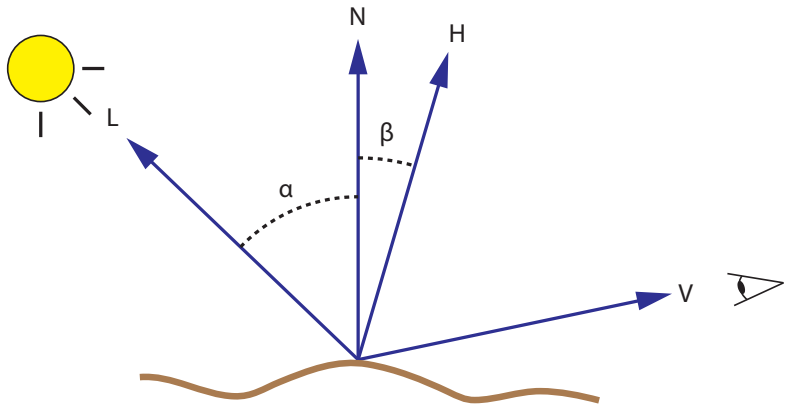


Figure 4.4: Phong illumination model: N is the surface normal at the point for which the illumination model is evaluated. The light vector L points towards the light source, and the view vector V points towards the viewer. H is the half-way vector between L and N.

reflections into account. While this may not be very realistic, it allows illumination to be computed efficiently. The model consists of independent ambient, diffuse and specular terms and employs the light vector, the view vector, and surface normal for computation. The **light vector** L is the normalized vector from a location in space to the light source. In case of a directional light source, this vector is the same for all points in a scene. The **view vector** V is the normalized vector from a location in space along a viewing ray to its origin on the image plane. In case of parallel projection, this vector is the same for all points in a scene. The Phong illumination model was originally designed for surface rendering. In volume rendering, the **surface normal** N is approximated by the normalized gradient.

Figure 4.4 shows an illustration of the Phong illumination model. Additionally, the half-way vector $H = \frac{1}{2}(L + V)$ is displayed. The final light intensity is determined by the three constants $k_{ambient}$, $k_{diffuse}$, and $k_{specular}$, which control the contribution of each term. The shaded color is computed by multiplying the input color (e.g., the classified sample) by the sum of the three terms, as can be seen in Equation 4.5. This equation only holds true under the assumption, that the color of the light source is always white and its color contribution can therefore be disregarded.

$$c_{out} = c_{in} \cdot (I_{ambient} + I_{diffuse} + I_{specular}) \tag{4.5}$$

The ambient term $I_{ambient}$ is constant and simulates the contribution of indirect re-

flections, which are otherwise not accounted for by the model.

$$I_{ambient} = k_{ambient} \tag{4.6}$$

The diffuse term $I_{diffuse}$ is based on Lambert's cosine law, which states that the reflection of a perfect diffuse surface is proportional to the cosine of the angle α between the light vector L and the surface normal N. In other words, the reflection is proportional to the dot product of L and N.

$$I_{diffuse} = k_{diffuse} \cdot \max(L \cdot N, 0) \tag{4.7}$$

The specular term $I_{specular}$ simulates specular reflections by adding a highlight. Blinn [2] proposed to use the half-way vector H to compute the specular term. The half-way vector is the vector halfway between the light vector and the view vector. The specular lighting intensity is then proportional to the cosine of the angle β between the half-way vector H and the surface normal N raised to the power of n, where n is called the specular exponent of the surface and represents its shininess. Higher values of n yield smaller, sharper highlights, whereas lower values result in large and soft highlights.

$$I_{specular} = k_{specular} \cdot \max((H \cdot N)^n, 0) \tag{4.8}$$

Compositing

In raycasting, the volume rendering integral is approximated by applying the *over-operator* [36] in front-to-back order. This means, at each sample location the current color and alpha values of a ray are given by

$$\begin{aligned} c_{out} &= c_{in} + c(s)\alpha(s)(1-\alpha_{in}) \\ \alpha_{out} &= \alpha_{in} + \alpha(s)(1-\alpha_{in}) \end{aligned} \tag{4.9}$$

where c_{in}, α_{in} are the color and opacity values already accumulated by the ray. s is the reconstructed function value and $c(s)$ and $\alpha(s)$ are the classified color and opacity values derived from the transfer functions. The advantage of using the front-to-back formulation of the over-operator is the possibility of early ray termination. This is, a ray can be terminated as soon as it has accumulated full opacity (i.e., $\alpha_{out} = 1$).

4.3 Trajectory Rendering

Animated voxels travel between volume representations on predefined trajectories. Instead of using these trajectories as an animation path, they can be rendered themselves to visualize the connection between two representations. The trajectories can be linear,

quadratic or cubic Bézier splines, and are colored according to the transfer function. These splines are approximated by line strips, which have to be pre-computed on the CPU.

4.3.1 Illumination of Lines

To render illuminated streamlines, cylinders could be used to draw the line segments and light them using graphics hardware. Alternatively, simple line segments as graphical primitives would reduce geometric complexity and therefore speed up rendering considerably. Unfortunately line segments have no distinct normal vector. Thus it is impossible to directly apply a shading model like Phong shading for the illumination of a point P on a line. Let L be the light direction, V the viewing direction and R the unit reflection vector (the vector in the $L-N$-plane with the same angle to the surface normal as the incident light). Then light intensity I at a given point P is given by

$$\begin{aligned} I &= I_{ambient} + I_{diffuse} + I_{specular} \\ &= k_a + k_d L \cdot N + k_s (V \cdot R)^n. \end{aligned} \tag{4.10}$$

Choosing the normal vector N as the one that is coplanar to the tangent vector T and the light direction L, $L \cdot N$ can be expressed without N [50]:

$$L \cdot N = \sqrt{1 - (L \cdot T)^2}. \tag{4.11}$$

$V \cdot R$ can be rewritten without R in a similar way, yielding

$$V \cdot R = (L \cdot T)(V \cdot T) - \sqrt{1 - (L \cdot T)^2}\sqrt{1 - (V \cdot T)^2} \tag{4.12}$$

To exploit graphics hardware for the illumination, the texture matrix is loaded with L and V:

$$M = \frac{1}{2} \begin{pmatrix} L_1 & V_1 & 0 & 0 \\ L_2 & V_2 & 0 & 0 \\ L_3 & V_3 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \tag{4.13}$$

Now $L \cdot T$ in Equation 4.11 is set to $2t_1 - 1$ and $V \cdot T$ in Equation 4.12 is set to $2t_2 - 1$. With t_1 and t_2 running from 0.0 to 1.0 in both coordinates, the results of the equations are stored in a 2D texture map. Next the texture coordinates of P are set to the normalized tangent vector. This way OpenGL calculates the inner products of the Phong equation with the help of the texture matrix, yielding the correct illumination color in P as texture color. See Zöckler et al. [50] for more details.

CHAPTER 5

Animated Transitions

This work employs two methods to connect different volume representations. The first, *trajectory rendering*, was already discussed in Section 4.3 and the second, *animation*, will be discussed in this chapter.

Each voxel of a volume can be interpreted as an n-dimensional attribute vector:

$$v = (a_1, a_2, \ldots, a_n)^T \tag{5.1}$$

Some examples for voxel attributes are the grid coordinates (x, y, z), the intensity I and the gradient magnitude m. The gradient is calculated for each voxel and is a vector, that points in the direction of the greatest rate of increase, and its magnitude is the greatest rate of change. Volumetric data can be illustrated in multiple ways. The most common representations are 2D slices orthogonal to one of the major axes, a 3D view rendered by one of the techniques described in Section 2.2, and histograms for intensity and gradient magnitude, as well as a 2D histogram combining these two voxel attributes. This work focuses on the 3D and the histogram representations. The 3D representation lives in the spatial domain, whereas the histograms live in the 2D domain. In order to implement animated transitions between these views, the histograms have to be placed on a plane in 3D space. Each representation has its own local coordinate system, in which each voxel has local coordinates. For the 3D view, the voxels' grid coordinates are used. The coordinates of a voxel in the 2D histogram are simply its intensity and its gradient magnitude. In case of 1D histograms, the horizontal axis represents either the intensity or the gradient magnitude, and the vertical axis is the (logarithmic) bin position. This bin position is the only coordinate that cannot be mapped to one of the voxels' attributes and thus has to be calculated for each voxel. The bin position of a voxel is the number of voxels present in the according bin after the voxel was added.

5.1 Principles for Animation

> "Smooth interactive animation is particularly important because it can shift a user's task from cognitive to perceptual activity, freeing cognitive processing capacity for application tasks." - *Robertson (1991)* [38, p. 5]

Over the last decades, extensive research has been conducted in the field of animation. Tversky et al. [41] performed a skeptical analysis of animation and its alleged superiority to static graphics for "conveying the workings of complex systems". According to them, animations are often too complex or too fast to be accurately perceived. However they made an exception for animated transitions in visualizations. They also point out, that the drawbacks of animation can be overcome with the aid of interactivity. Interactivity allows the user to arbitrarily control the animation by moving forward and backward or pausing it at any time. They propose two principles that specify conditions for effective animation. Their *Congruence Principle* states "the structure and content of the external representation should correspond to the desired structure and content of the internal representation" and their *Apprehension Principle* states "the structure and content of the external representation should be readily and accurately perceived and comprehended". According to those principles, data is best presented in its inherent dimensions, as long as it is easily comprehensible. Hence, animation should be well suited for conveying the concept of change over time. Adhering to those principles, however, does not make animation superior to static graphics per se. The usefulness of animation is also highly dependent on the number and complexity of the objects being animated, and the path they describe. When the features of the moving objects are not relevant, they can be represented by simple colored dots. This visual simplification, as well as moving the objects along simple trajectories, helps reducing the cognitive load for the user significantly. However, volumetric datasets are comprised of millions of voxels (volume elements), leading to unavoidable overlap during the animation. Minimizing this overlap is a major issue, which can be addressed by reducing the number of objects moving simultaneously, and by minimizing the overlap of their individual trajectories. The calculation of these trajectories and minimizing the number of simultaneously moving objects are described in the next section and in Section 5.3, respectively.

5.2 Interpolation Methods

This section describes the details of the mathematical tools used to facilitate animation. Let us start by briefly reviewing the theoretical background. A linear combination of points P_1, P_2, ..., P_n and weights λ_1, λ_2, ..., λ_n given by

$$P_0 = \sum_{i=1}^{n} \lambda_i P_i, \quad \sum_{i=1}^{n} \lambda_i = 1, \tag{5.2}$$

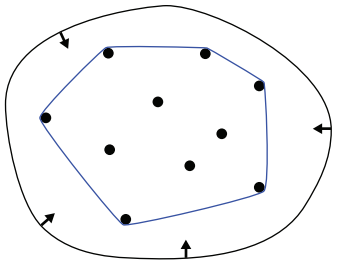


Figure 5.1: The convex hull (blue) can be visualized by imaging an elastic band stretched out around all points. When the band is released, it assumes the shape of the convex hull.

is called an *affine combination*, when the sum of all weights λ_i is 1. The *affine hull* of a set S of points is the set of all affine combinations of a finite subset of S. For example, line AB is the affine hull of points A and B, plane ABC is the affine hull of non collinear points A, B, and C.

A more restricted subset of affine combinations are *convex combinations*, where the coefficients λ_i not only sum up to 1, but are also non-negative. A convex combination is given by:

$$P_0 = \sum_{i=1}^{n} \lambda_i P_i, \quad \sum_{i=1}^{n} \lambda_i = 1, \quad \lambda_i \geq 0. \tag{5.3}$$

The definition of the *convex hull* is analogous to the affine hull. The convex hull of points A and B is the segment $[AB]$, and triangle ABC is the convex hull of points A, B, and C. An analogy for the convex hull in the plane is shown in Figure 5.1.

Since each voxel moves on its own *trajectory*, it has to be calculated individually. This trajectory is the path a voxel describes on its way from one representation to another one. The computation of that path is done on-the-fly by interpolating intermediate values between the endpoints of the animation. Depending on the number of these endpoints, different methods have to be applied. When the animation has two endpoints, the methods described in subsection 5.2.1 are employed. Using three or more endpoints for the interpolation requires methods described in subsection 5.2.2.

5.2.1 Convex Combination of Two Points

Throughout this section the interpolation parameter is denoted as t and is in the interval $[0, 1]$. Whenever $t > 0$ or $t < 1$, the voxels are on the move, otherwise they are residing in one of the representations mentioned earlier. A convex combination of two points P_1

and P_2 using $1-t$ and t as their weights respectively, lets the voxels travel on a straight line between those points.

$$P_0 = (1-t)P_1 + tP_2$$

To make the voxels travel on a curved trajectory, parametric curves can be utilized. In general, a parametric curve is a function of one independent parameter, denoted as t. A particular example is a curve $P(t)$ that is defined by a set of *control points* P_i and *blending functions* $f_i(t)$, $i = 1, \ldots, n$ [39, ch. 5], given by:

$$P(t) = \sum_{i=1}^{n} P_i w_i(t) \tag{5.4}$$

Each point on this curve is computed as a weighted sum of all control points. This means that each point (of the curve) is influenced by every control point according to the assigned blending function. A blending function defines the weight of the control point at each point of the curve. A value of 0 indicates that the control point is not affecting the point on the curve. If the blending function reaches 1, the curve is going through the control point. An example of blending functions is shown in Figure 5.2, where this behavior can be observed.

Bézier Curves

In case of *Bézier curves* the set of blending functions - one for each control point - is given by the *Bernstein polynomials* [39, ch. 2]

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i, \quad t \in [0,1] \tag{5.5}$$

where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ is the binomial coefficient.

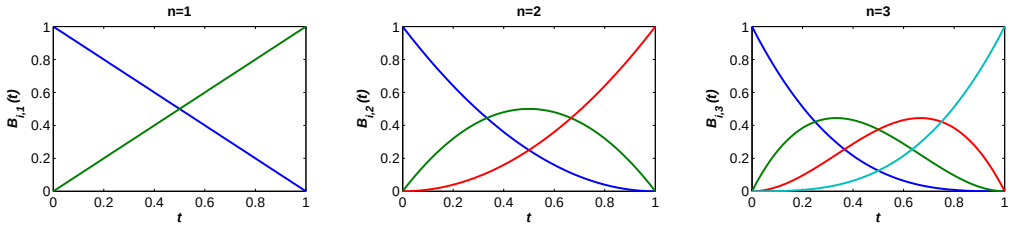
The sum of the Bernstein polynomials of degree n is equivalent to the *Binomial theorem*, given by:

$$\sum_{i=0}^{n} \binom{n}{i} a^{n-i}b^i = (a+b)^n \tag{5.6}$$

After substituting a and b by $(1-t)$ and t respectively, then one has:

$$\sum_{i=0}^{n} B_i^n(t) = ((1-t)+t)^n \equiv 1 \tag{5.7}$$

According to Equation 5.3 this fact and that $B_i^n(t) \geq 0$ (as can be verified in Figure 5.2), makes the Bézier curve a convex combination of its control polygon. This provides the Bézier curve with some of its most important properties:



(a) n=1 (b) n=2 (c) n=3

Figure 5.2: The Bernstein polynomials $B_i^n(t)$ for (a) $n = 1$, (b) $n = 2$, and (c) $n = 3$. The blue, red, green and cyan lines correspond to the blending functions of the first, second, third and forth control point respectively.

- It always passes through its first and last control points, and is tangent to the control polygon at these points.
- It can be transformed (translated, rotated, scaled, sheared) by performing these operations on the control points.
- It lies within the convex hull of the control points.

Finally, when substituting $w_i(t)$ in Equation 5.4 by Equation 5.5, one gets the equation for the Bézier curve:

$$C(t) = \sum_{i=0}^{n} P_i \binom{n}{i} (1-t)^{n-i} t^i, \quad t \in [0,1] \tag{5.8}$$

Evaluation of Bézier Curves The hands-on approach to determine a curve point $C(t)$ at time t would be the evaluation of Equation 5.8. Apart from being inefficient, this solution also suffers from numerical instability caused by raising floating-point numbers to higher powers. The most common algorithm to evaluate Bézier curves is the *De Casteljau algorithm* [8]. Basically, Equation 5.8 is rewritten as recursive linear interpolations, or more precisely recursive *convex combinations*. This way the evaluation is reduced to basic arithmetic operations and becomes numerically stable. A graphic depiction of the algorithm along with a short description can be found in Figure 5.3.

Using De Casteljau's algorithm to evaluate a polynomial curve of degree n, still has a computational complexity of $\mathcal{O}(n^2)$. In an effort to reduce this cost, several evaluation schemes for curves based on the Bernstein polynomial have been proposed. A comparison of these alternative approaches was done by Delgado and Peña [10]. They also introduce an algorithm of their own [9], which has a complexity of $\mathcal{O}(n)$. Although their approach seems to be superior to De Casteljau's algorithm in terms of complexity, let us take a closer look at the number of operations needed to evaluate a Bézier

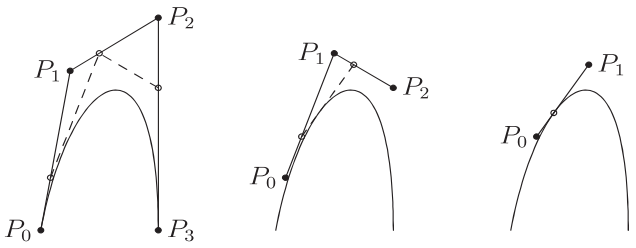


Figure 5.3: De Casteljau's algorithm for Bézier curves: Each line segment of the control polygon is subdivided with the ratio $\frac{1-t}{t}$ and the resulting points are connected. The process is repeated until one arrives at a single point. This is the point of the curve for the given parameter t [48].

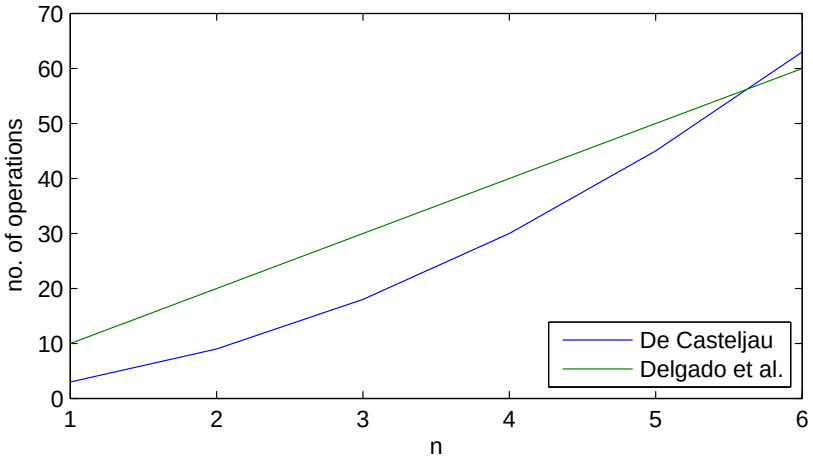


Figure 5.4: A comparison of De Casteljau's to Delgado's [9] algorithm in terms of computational complexity for a degree n Bézier curve.

curve of degree n. The De Casteljau algorithm needs $\frac{3n(n+1)}{2}$ and according to Delgado and Peña [9], Delgado's algorithm needs $10n$ basic arithmetic operations. Figure 5.4 reveals, that Delgado's algorithm is only faster for $n \geq 6$. Moreover, for $n \leq 3$ De Casteljau's algorithm is approximately twice as fast. Since the maximum degree used in this application is $n = 3$, De Casteljau's algorithm is preferable to Delgado's approach.

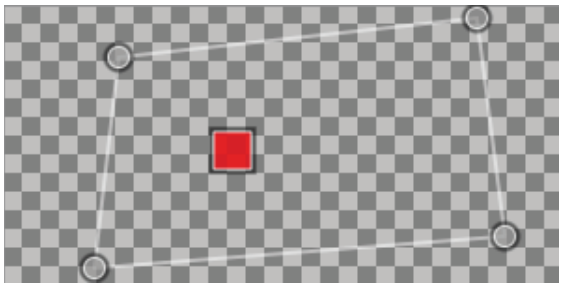


Figure 5.5: Each of the polygon's circular vertices represents an endpoint of the animation. When moving the center of mass, represented by the red square, the weights of the vertices are changed accordingly.

5.2.2 Convex Combination of Three or More Points

The last section described convex combinations of two points, which correspond to two animation endpoints. Adding endpoints to the animation requires adding points to the interpolation. When using more than two endpoints, a simple slider is not enough to interactively change the parameters for the interpolation. The parameters, also referred to as weights, need to be determined by moving a vertex inside a closed 2D-polygon (see Figure 5.5), whose vertices represent the animation endpoints. These weights are then used to interpolate voxel positions for the animation. In order for this procedure to work, a bijective mapping between the weights and the vertex positions is required. In other words, for a given polygon a certain combination of vertex weights leads to exactly one point coplanar to the polygon, and vice versa (see Figure 5.6).

Barycentric Coordinates

The barycentric coordinates of a point specify the center of mass of the weights placed at the vertices of a *simplex*. A simplex is the simplest possible polygon for a given dimension. Adding a vertex to a simplex also expands the simplex to a higher dimension. For instance, a 1-simplex is a line segment, a 2-simplex is a triangle, and a 3-simplex is a tetrahedron. So an n-dimensional simplex has $n + 1$ vertices. As a consequence, the points forming a simplex are always linearly independent, and a point inside of a simplex is uniquely determined by its barycentric coordinates. Since this implementation makes use of barycentric coordinates in the plane, the only simplex taken into account is the triangle. For a triangle formed by the points P_1, P_2, P_3 and their according weights λ_1, λ_2, λ_3, any point P_0 in the plane of this triangle is given by:

$$P_0 = \sum_{i=1}^{3} \lambda_i P_i, \quad \sum_{i=1}^{3} \lambda_i = 1 \tag{5.9}$$

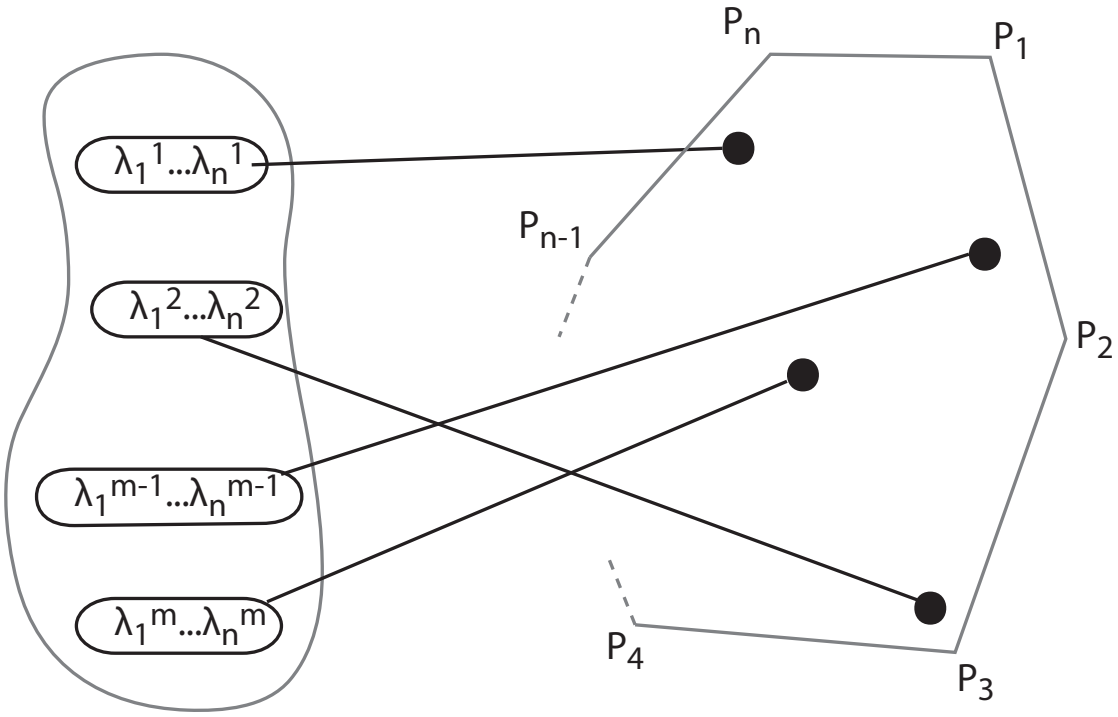


Figure 5.6: Bijective mapping: A set of weights $\lambda_1 \dots \lambda_n$ always maps to exactly one point inside the polygon $P_1 \dots P_n$ and vice versa.

The weights λ_i are the barycentric coordinates relative to the points of the triangle. If all weights are non-negative, then P_0 lies inside the triangle. Furthermore, if one of the weights is 0, then P_0 lies on the opposite edge of the triangle. When two of the weights are 0, the third weight becomes 1 and places P_0 on the corresponding vertex of the triangle.

The aforementioned bijective property is needed, because each point must have unique weights. Otherwise obtaining the weights as described in Figure 5.5 is not possible. In order to determine the barycentric coordinates of a point, its Cartesian coordinates have to be transformed with respect to a triangle, using the following equation:

$$\begin{aligned} \lambda_1 &= \frac{(y_2-y_3)(x-x_3)+(x_3-x_2)(y-y_3)}{(y_2-y_3)(x_1-x_3)+(x_3-x_2)(y_1-y_3)} \\ \lambda_2 &= \frac{(y_3-y_1)(x-x_3)+(x_1-x_3)(y-y_3)}{(y_2-y_3)(x_1-x_3)+(x_3-x_2)(y_1-y_3)} \\ \lambda_3 &= 1-\lambda_1-\lambda_2 \end{aligned} \tag{5.10}$$

Here, x and y are the Cartesian coordinates of a point coplanar to the triangle, and x_i and y_i are the Cartesian coordinates of the triangle's points. Barycentric coordinates can

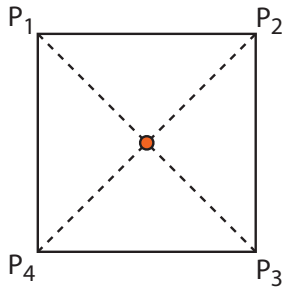


Figure 5.7: Points inside a polytope are not uniquely determined by their barycentric coordinates. For example, the center of the square can be described as $\frac{1}{2}(P_1 + P_3)$ or as $\frac{1}{2}(P_2 + P_4)$ or as $\frac{1}{4}(P_1 + P_2 + P_3 + P_4)$

be extended from a simplex to a *polytope*, which is an n-dimensional polygon with more than $n + 1$ vertices. However the vertices of a polytope are not linearly independent, which leads to points, that are not uniquely determined by their barycentric coordinates. For example, the center of a square can be described as the midpoints of both diagonals, as shown in Figure 5.7.

A common way to deal with arbitrary polygons in 2D is to triangulate them first, and apply barycentric coordinates on each simplex. However the results depend on the choice of triangulation and contain unnecessary artifacts. An overview of methods for obtaining unique barycentric coordinates for polytopes can be found in [13, 45].

Mean Value Coordinates

Among others, Floater et al [13] proposed a way to generalize barycentric coordinates to 2D polytopes, called *mean value coordinates*. Let α_i, $0 < \alpha_i < \pi$, be the angle at P_0 in the triangle $[P_0, P_i, P_{i+1}]$, defined cyclically (see Figure 5.8). The weights

$$\lambda_i = \frac{\omega_i}{\sum_{j=1}^{k} \omega_j}, \quad \omega_i = \frac{\tan(\alpha_{i-1}/2) + \tan(\alpha_i/2)}{\|P_i - P_0\|} \tag{5.11}$$

are coordinates for P_0 with respect to $P_1 \dots P_k$. These weights can then be used to interpolate the voxel positions using Equation 5.3. When used on a triangle, mean value coordinates are equivalent to barycentric coordinates.

5.3 Staggered Animation

When starting and ending times of all voxels are the same, only the voxels closest to the viewer are visible and occlude the rest. Using transparency alone is not enough to

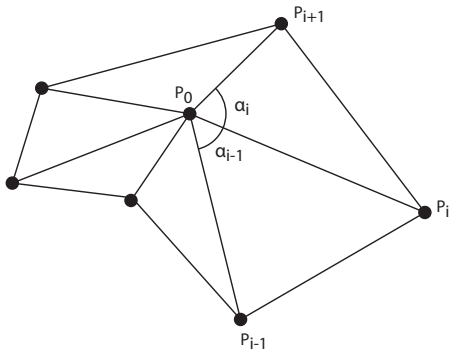


Figure 5.8: Mean value coordinates.

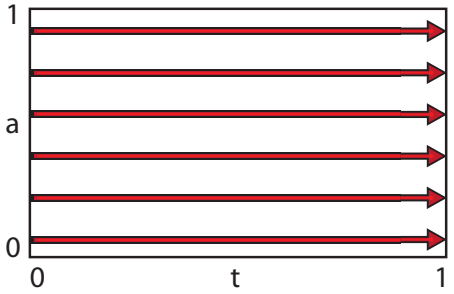


Figure 5.9: Non delayed animation: The red arrows represent moving voxels, $t \in [0, 1]$ is the interpolation parameter, and a is one of the normalized voxel attributes

clarify the animation, due to the sheer number of voxels. To overcome this problem either the overall number of voxels or the number of voxels being moved simultaneously has to be reduced. Decreasing the number of voxels is not the goal of this work, which leaves us with decreasing the number of simultaneously moving voxels. In their work about animations in statistical data graphics [14], Heer and Robertson had to deal with a similar occlusion problem. According to them, issuing small delays in movement separated the items' starting (and ending) times, leading to small but noticeable decreases in the amount of overlap. This idea was taken and extended by making the delay dependent on one of the normalized voxel attributes. These attributes are *intensity*, *gradient magnitude*, and *grid coordinates*. This means that voxels with a low-valued attribute start moving earlier whereas voxels with high intensities start moving later. This order can easily be reversed, causing voxels with a high-valued attribute to move first.

For the interpolation a parameter t is used. Without delaying the starting time, all voxels start moving as soon as t is greater than 0 and end their movement when t equals to 1, which can be observed in Figure 5.9.

Let t_a be the interpolation parameter used for moving voxels with a certain attribute

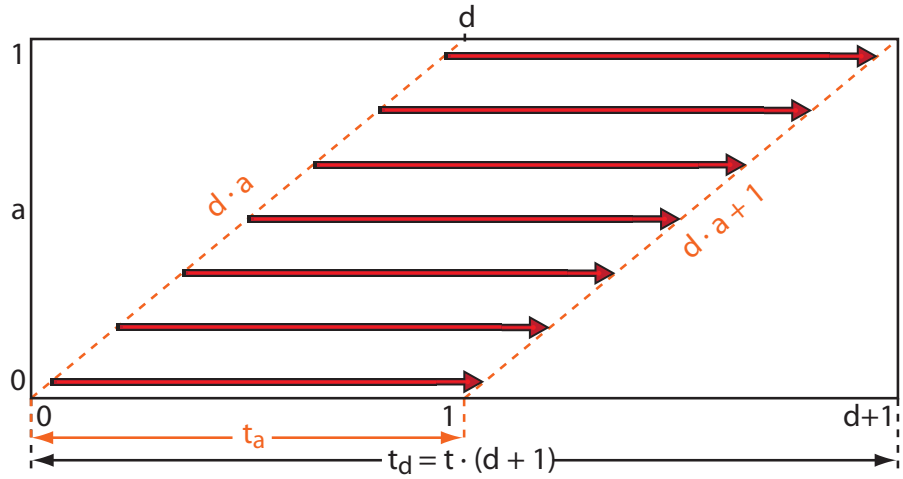


Figure 5.10: $t_d = t(d+1)$ is the delayed interpolation parameter, with t in $[0,1]$. a denotes the normalized voxel attribute used to scale the maximum delay d. Moving voxels are depicted as red arrows. Their movement starts and stops, as soon as t_d equals $d \cdot a$ and $d \cdot a + 1$ respectively.

value. Without adding a delay, t_a equals t for all values of a. Whereas adding a delay separates the t_a's into staggered subintervals of t (see Figure 5.10). In order to accommodate these subintervals, t needs to be scaled. This delayed and scaled interpolation parameter is given by

$$t_d = d \cdot a + t_a, \quad d \geq 0, \tag{5.12}$$

where a is the normalized voxel attribute and d denotes the maximum delay imposed on the voxels' starting times. When $d = 0$, no delay is added and all voxels move simultaneously. For $d > 0$ voxels are delayed according to their attribute a. In order to calculate the interval of t_d, the earliest starting time and the latest ending time need to be computed. The earliest starting time obviously is 0 and evaluating Equation 5.12 for $t_a = 1$ and $a = 1$, yields the latest ending time of $d+1$. Hence, $t_d \in [0, d+1]$ and since $t \in [0,1]$, t_d in Equation 5.12 can be substituted by $t(d+1)$ yielding:

$$t(d+1) = t_a + d \cdot a \tag{5.13}$$

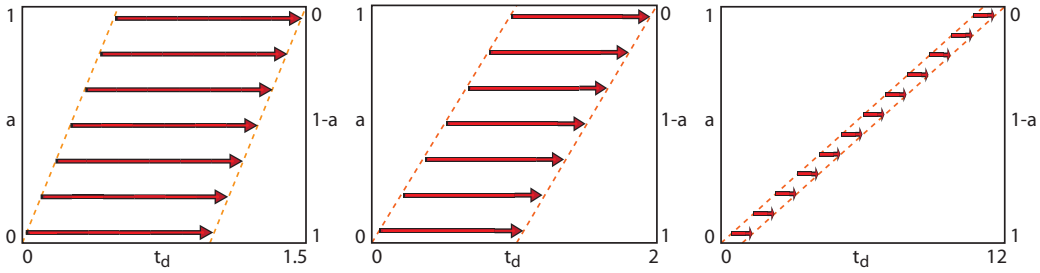
Solving this equation for t_a allows us to express t_a in terms of the interpolation parameter t, yielding:

$$t_a = t(d+1) - d \cdot a \tag{5.14}$$

Because this equation can yield values outside the interval $[0,1]$, the result has to be clamped to a lower and upper bound of 0 and 1, respectively. This is necessary, because

t	a	t_a
0.0	0.0	-d
1.0	0.0	1.0
0.0	1.0	0.0
1.0	1.0	d+1

Table 5.1: This table shows the lower and upper bound of the interpolation parameter t_a, which is used to move voxels with attribute value a (see Equation 5.15).



(a) d=0.5 (b) d=1 (c) d=11

Figure 5.11: Moving voxels are represented by red arrows. Figures (a) to (c) illustrate examples for different values of the delay parameter d, used in Equations 5.14 and 5.15.

t_a is used to parameterize a convex combination (see Equation 5.3). Using Equation 5.14, the starting times for the voxels are ascending according to the chosen attribute a . This order can easily be reversed by substituting a by $1-a$. The equation for descending order is then given by:

$$t_a = t(d+1) - d \cdot (1-a) \tag{5.15}$$

5.3.1 Optimal Delay

Animation is basically a series of consecutive frames. In order to be smooth, the images shown in two consecutive frames should not change significantly. This is especially important, when using interactivity, where the user can arbitrarily navigate through the animation, and pause/resume it at any time. Since the animation is controlled by an interpolation parameter t, the difference between two consecutive frames is proportional to the difference of the two values of t, used to compute them. Let n_{max} be the number of possible values for t, then the smallest *step size* of t is $\Delta t_{min} = \frac{1}{n_{max}}$. In case of a 32 bit float, where 1 bit is used for the sign, 7 bits are used for the exponent and 24 bit for the coefficient, n_{max} is 2^{24}. As a consequence, the smallest representable difference

a	a_{max}
x coordinate	volume width
y coordinate	volume height
z coordinate	volume depth
intensity	max intensity
gradient magnitude	max grad. magnitude

Table 5.2: This table lists the voxel attributes a and the corresponding number of possible values a_{max}. For the intensity and gradient magnitude a_{max} is not necessarily the numerical limit of the data type, but rather the highest value occurring in the volume.

is $\Delta t_{min} = 2^{-24} \approx 5.960 \cdot 10^{-8}$. Figure 5.10 shows, that introducing a delay d, scales the interpolation interval to $t_d = t(d+1)$, containing all the subintervals, needed to animate all distinct attribute values. t remains in the interval $[0, 1]$ and gets subdivided into subintervals of size $\frac{1}{d+1}$. Since n_{max} is the number of available animation steps for the interval of t, the number of steps n for such a subinterval is given by:

$$n = \frac{n_{max} + d}{d + 1} \tag{5.16}$$

The maximum delay d_{max} depends on the lowest number of steps $n = n_{min}$ required for a subinterval. Rewriting Equation 5.16, this maximum delay can be written as:

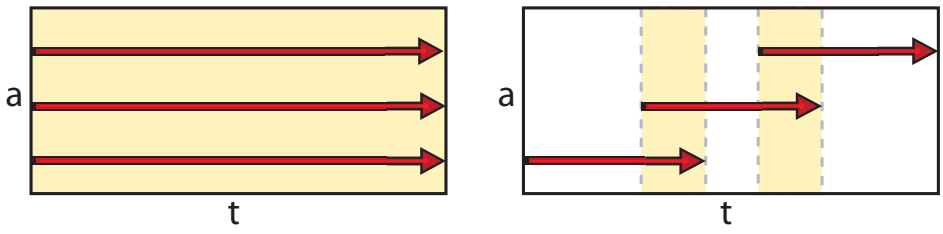
$$d_{max} = \frac{n_{max} - n_{min}}{n_{min} - 1} \tag{5.17}$$

The optimal value of d is found, when the subintervals of t of two consecutive values of attribute a do not overlap and there is no gap between them. In order to be able to quantify this overlap, the number of possible values for each voxel attribute, must be taken into account. In general, this number a_{max} is different for all voxel attributes. A list of all voxel attributes and their according a_{max} can be found in Table 5.2. Using a_{max} and delay d, the subinterval overlap of two consecutive values of a is given by:

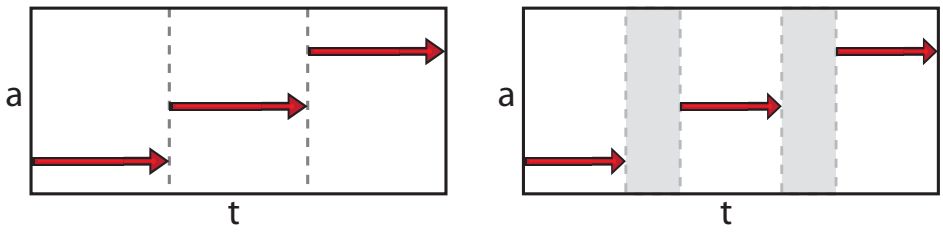
$$o = 1 - \frac{d}{a_{max}} \tag{5.18}$$

The overlap can be interpreted as the percentage of an interval, overlapped by its preceding or succeeding interval. In other words, when $0 < o \leq 1$, two consecutive intervals overlap. If $o < 0$, there is a gap between the intervals. If $o = 1$, the two (and as a consequence, all) intervals coincide. Finally, when $o = 0$ there is no overlap and no gap between them. This behavior is illustrated in Figure 5.12. Using Equation 5.18, the delay can be expressed in terms of the desired overlap:

$$d = (1 - o)a_{max} \tag{5.19}$$



(a) $o = 1$: full overlap (b) $0 < o < 1$: partial overlap



(c) $o = 0$: no overlap, no gap (d) $o < 0$: negative overlap, or gap

Figure 5.12: Illustration of the interval overlap o defined in Equation 5.18.

The goal is to minimize occlusion during the animation process. An overlap of 1 means, that all voxels travel at once, an overlap less than 0 means, that there are gaps in the animation, where no voxels move at all. This would be a waste of available animation steps. So the optimal value for the overlap is 0, where voxels start moving as soon as their attribute-wise predecessors stopped. Evaluating Equation 5.19 for an overlap of 0 yields the optimal delay for a given attribute a:

$$d_{opt} = a_{max} \tag{5.20}$$

Depending on the chosen attribute, or rather a_{max}, d_{opt} could exceed d_{max}, defined in Equation 5.17, and as a consequence reduce the number of available animation steps for a subinterval to a value below the desired minimum. Given that even for the float data type, there are 2^{24} animation steps available, this will hardly ever be the case, but for the sake of completeness the optimal delay should be rewritten as:

$$d_{opt} = \min(a_{max}, \frac{n_{max} - n_{min}}{n_{min}}) \tag{5.21}$$

This ensures an optimal exploitation of the animation steps for the subintervals. As far as this delaying approach goes, the number of simultaneously traveling voxels, has been minimized. Still, all voxels sharing the same attribute value move at the same time. For example, let the delay be based on one of the grid coordinates of the voxels. Then the number of voxels moving simultaneously is at least the number of voxels of a slice

perpendicular to the selected axis. This also holds for the remaining voxel attributes, intensity and gradient magnitude.

5.3.2 GUI Controls for the Interpolation Parameter

Using an optimal delay with no overlap causes problems, when using a simple slider. There are 2^{24} animation steps and s possible slider states. Thus, a single slider step equals $\frac{2^{24}}{s}$ animation steps. Optimally, one slider step should correspond to one animation step. Applying an optimal delay reduces the available interval to $\frac{1}{d+1}$. If the interval becomes smaller than $1/s$, steps are being skipped, due to the low resolution of the user interface. This problem could be circumvented using the mouse wheel or cursor keys on the slider. However, this makes the control sequential and can hardly be used to traverse the whole interpolation interval. A good choice of a user interface for the interpolation parameter would probably be a jog dial, as used for professional video editing systems.

5.4 Parameter Transfer Function

The staggered animation approach does not take the distribution of the voxel attributes into account. All voxels, regardless of the relative frequency of their attribute value, use the same amount of time to complete their transition. Attributes like the grid coordinates are equally distributed, meaning, each value of the attribute has the same frequency throughout the volume. This can easily be verified by the fact, that each slice perpendicular to a selected axis has the same number of voxels. For these attributes, an adaptive voxel speed is not really necessary. Intensity and gradient magnitude, on the other hand, are unevenly distributed attributes. For these attributes, conditioning the traversal speed of the interpolation interval on the frequency of the attribute values would be preferable. It makes sense to spend more time on frequently occurring voxel values, and less time on rarely occurring voxel values. This can be realized by applying a so-called *parameter transfer function (PTF)* to the interpolation parameter t. This function yields a transferred interpolation parameter t' and is given by:

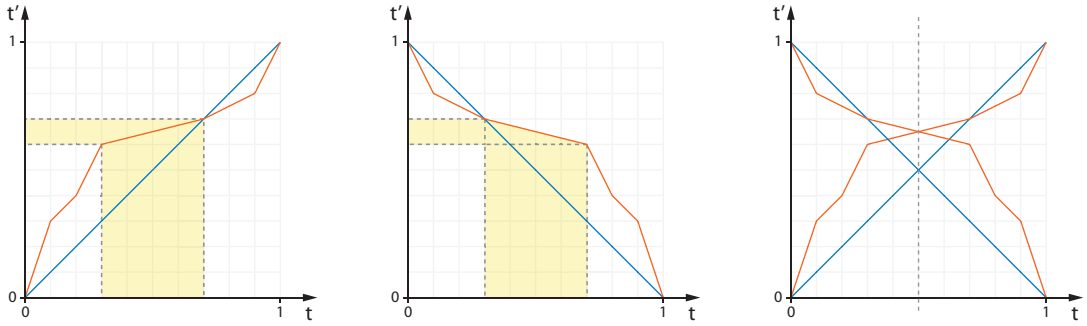
$$t' = f(t), \quad f\colon [0,1] \mapsto [0,1] \tag{5.22}$$

This function is also used to switch between ascending and descending delay order. For ascending order, $f(t)$ has to be strictly monotonic increasing, which means:

$$a < b \Rightarrow f(a) < f(b) \quad \forall a, b \in [0,1] \tag{5.23}$$

For descending order, $f(t)$ has to be strictly monotonic decreasing, or formally:

$$a > b \Rightarrow f(a) > f(b) \quad \forall a, b \in [0,1] \tag{5.24}$$



(a) Ascending (b) Descending (c) Transformation

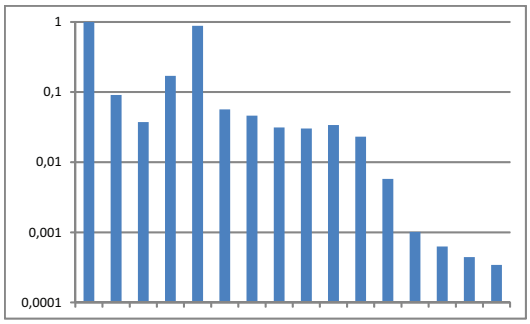
Figure 5.13: Figures (a) and (b) show the same two parameter transfer functions for ascending and descending order respectively. A PTF accelerates or decelerates voxels, when a subinterval of t with constant slope is mapped to a larger or smaller subinterval of t', respectively. For example, in the areas highlighted in yellow, a subinterval of t of size 0.4 is mapped to a subinterval of t' of size 0.1, effectively slowing down voxels to a quarter of the original speed represented by the blue line. (c) Geometrically, an ascending PTF can be converted to a descending PTF and vice versa, by reflecting it about the line parallel to the t'-axis and intersecting the t-axis at $t = 0.5$.

An ascending PTF $f_a(t)$ can be transformed into a descending PTF $f_d(t)$ by simply substituting its parameter t by $1 - t$, yielding:

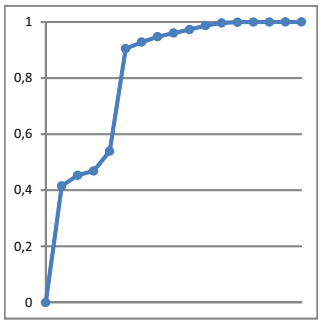
$$\begin{aligned} f_a(t) &= f_d(1-t) \\ f_d(t) &= f_a(1-t) \end{aligned} \tag{5.25}$$

This simple transformation makes it unnecessary to define the PTF for ascending and descending order separately. Figure 5.13 shows a comparison of the same PTFs in ascending and descending order and a geometric interpretation of the transformation. $f(t)$ has to be strictly monotonic increasing/decreasing, otherwise the value of t' would remain unchanged for subintervals of t and the animation would stop.

For each unevenly distributed voxel attribute, like intensity and gradient magnitude, it is desirable to define an appropriate PTF. Because of its relation to the frequency distribution of an attribute, a good PTF can be automatically generated. Figure 5.14 shows the frequency distribution and the cumulative distribution function (CDF) of the density values of a human head data set. The slope of the CDF of an attribute is steeper for more frequent values. This is the opposite behavior of the desired PTF. Hence, the PTF of an attribute can be derived from its CDF through inversion. The inversion can only be done, if the CDF is strictly increasing, otherwise its inverse would not be unique. Since all voxel attributes are discrete, so are their CDFs. Nonetheless, discrete functions can be inverted as well, by simply swapping the coordinates of their data points. The result is the discrete attribute-driven PTF. To avoid sudden jumps in the animation, the



(a) Intensity histogram



(b) A cumulative distribution function (CDF)

Figure 5.14: (a) shows a logarithmic histogram of a human head, and (b) shows the corresponding cumulative distribution function. The highest intensity for this dataset is 4096. To illustrate the discrete nature of the empirical CDF, the class width of the histogram was set to 256, resulting in 16 bins. Usually the class size is 1, nevertheless the values "between" the bins need to be interpolated.

PTF needs to be smoothed. On this account, linear interpolation is applied between the data points, yielding a piecewise linear function. The CDF of evenly distributed attributes has a constant slope of 1, and therefore the according attribute-driven PTF is the unaltered interpolation parameter $t' = t$. Attribute-driven PTFs automatically slow down the movement of frequent attribute values, and accelerate the coarser ones. This is the desired behavior in general, but in some cases adjustments are necessary. For example, a substantial part of volumetric data is composed of empty space and is usually transparent. Slowing down these voxels should be avoided, which can be done by adjusting the attribute-driven PTFs manually.

CHAPTER 6

Implementation

The implementation of our visualization prototype was done as a number of plug-ins for VolumeShop [4], a volume visualization framework implemented by Bruckner et al. VolumeShop provides a complete OpenGL and GLSL setup, functionality for loading and traversing of volumetric data, and interactive transfer functions. The plug-ins were implemented in Visual Studio 2008 using C++, and OpenGL and GLSL were used to program the graphics hardware. Using this framework significantly reduced the implementation effort and allowed us to focus on the implementation of animated transitions.

6.1 Implemented Plug-Ins

The renderer performing the animated transitions must be capable of rendering single voxels on arbitrary spatial coordinates. Being an object-order method (see Section 2.2.2) makes volume splatting the ideal candidate. In order to deliver appealing rendering results of the volume, a volume splatter must render the voxels in back-to-front order. This either requires a resorting of the voxels any time the viewing parameters change, or three copies of the volume in memory (see Chapter 4 for details). So it is either a pre-processing step on the CPU or a considerable increase in memory consumption. On the other hand, when using the splatter solely for voxel animation, simple compositing using the OpenGL blending function `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` is sufficient. This is why a raycaster, included in the framework, is being used to render the volume, and the animated voxels are rendered using a very basic volume splatter.

6.1.1 Volume Splatter

The voxels are loaded into a *vertex buffer object* and transferred to graphics memory once. The actual animation and the shading takes place in the vertex shader. The fragment shader is responsible for drawing the splats.

Vertex Shader The interpolation of the voxel coordinates takes place in eye coordinates, i.e., after transforming the coordinates of the start- and end-points of the animation with the respective *ModelView* matrix. The interpolation function can be a Bézier function of first, second or third degree (see Section 5.2.1).

Instead of using `GL_QUAD` the voxels are rendered using `GL_POINT_SPRITES`, which reduces the number of necessary coordinates per voxel from four to one. The size of the point sprites is set in the vertex shader using `gl_PointSize`. The color and opacity of each voxel is derived from the transfer function, which is implemented as a simple 1D texture, indexed by the normalized voxel intensity. Since the color is constant throughout a single point sprite, shading also takes place in the vertex shader.

Fragment Shader To avoid quadratic point sprites with constant opacity, the opacity channel of each point is multiplied by a 2D Gaussian kernel, given by:

$$G(x, y) = \frac{1}{(2\pi\sigma^2)} \cdot e^{-\frac{(x-0.5)^2+(y-0.5)^2}{2\sigma^2}} \tag{6.1}$$

Here x, y are the normalized coordinates inside a single point sprite, given by `gl_PointCoord` in the fragment shader. The circular 2D Gaussian function is a bivariate normal distribution of two uncorrelated variates x and y, and σ is the *standard deviation*. Since the same kernel is applied to each voxel, the computational effort can be significantly reduced by pre-computing the kernel. The result of the pre-computation is stored in a texture, which is then used as a lookup table indexed by `gl_PointCoord`.

6.1.2 Trajectory Renderer

Instead of moving voxels along trajectories, the trajectories themselves are rendered as line strips. Therefore, a neighborhood of voxels is selected interactively by the user from either the 2D histogram (using a 2D neighborhood) or the volume (using a 3D neighborhood). The selected voxels are highlighted in both representations and are connected by their trajectories. In order to render the trajectories, the vertices of each trajectory need to be pre-computed on the CPU and sent to the GPU as `GL_LINE_STRIP`s. Each trajectory gets its color from the transfer function and therefore has the same color as its corresponding voxel. In order to be able to apply illumination, a normal is required. Unfortunately, a line does not have a single normal, but rather a normal plane. Therefore, the vector coplanar to the tangent and the light vector is chosen as the normal. To

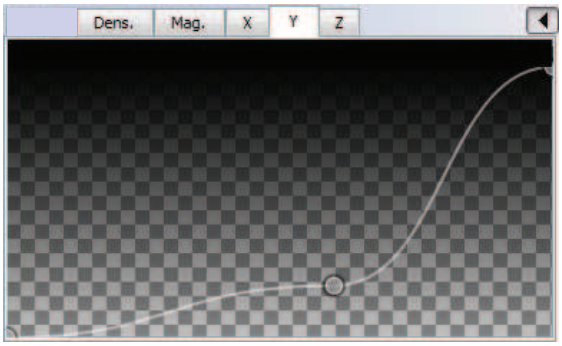


Figure 6.1: GUI element to define parameter transfer functions for each of the voxel attributes. In this example the majority of the interpolation interval is spent animating voxels with low y-coordinates.

be able to utilize the graphics hardware without pre-computing the normal of each line segment of each trajectory, the illumination parameters have to be precomputed into a texture. Here each color channel corresponds to one of the three illumination parameters (ambient, diffuse and specular) used in the Phong equation. A brief description of this method, introduced by Zöckler et al. [50], can be found in Section 4.3.

6.1.3 Parameter Transfer Function

Besides the renderers, a component to define the parameter transfer functions, described in Section 5.4, had to be implemented. These functions allow the user to control the amount of time spent in certain subintervals of the interpolation parameter. In some cases it is desirable to spend more time in an interval containing more voxels, and less time in intervals containing a smaller number of voxels. Figure 6.1 shows the GUI element used to define the PTFs.

CHAPTER 7

Results

The main goal of this work is to help users gain insight into volumetric data. Since animation is very useful for understanding statistical data [14], we want to apply the same idea to volumetric data. Animating volumetric data is a major challenge due to the huge number of objects to be animated at once. Moving all voxels simultaneously is not an option, thus staggered animations, introduced by Heer and Robertson [14], are implemented. Furthermore, to keep the number of simultaneously moving objects at a minimum, non-contributing voxels are hidden. These measures help reducing the cognitive load for the user. The figures in this chapter depict animations, that use the volume as the starting point.

7.1 Interpolation Methods

The voxels' trajectories can be linear, quadratic and cubic Bézier splines (see Section 5.2.1). The animation is controlled by an interpolation parameter $0 \leq t \leq 1$, where $t = 0$ and $t = 1$ represent the start- and end-point of the animation, respectively. The animation path can be calculated using linear interpolation (Figure 7.1), quadratic interpolation (Figure 7.2), or cubic interpolation (Figure 7.3).

7.2 Staggered Animation

The animation can be delayed using one of the inherent voxel attributes. These attributes are the grid coordinates (x, y, z), density and gradient magnitude. The voxels can either be delayed in ascending or descending attribute order, meaning, that voxels with low or high attribute values start moving first. Figure 7.4 shows the results, when the delay is based on the voxels' densities. In ascending order, voxels with *low*

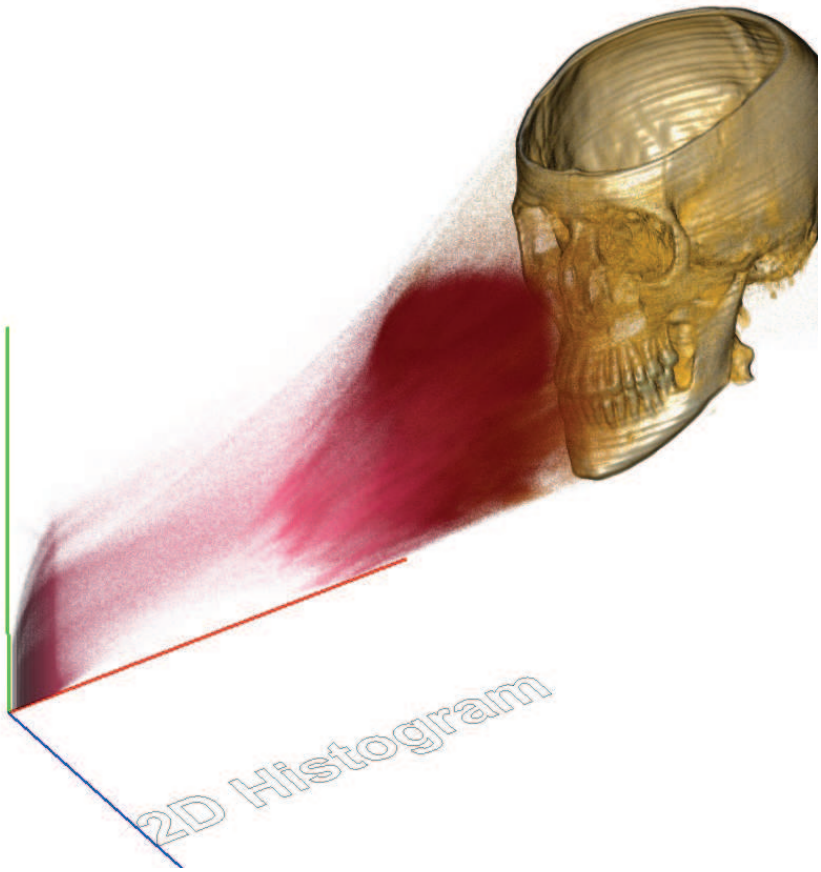


Figure 7.1: The screen shot was taken for an interpolation parameter $t = 0.2$. Two points are required when using linear interpolation to calculate the animation paths. The red and green axes are the normalized voxel density and gradient magnitude, respectively. The blue axis was added for spatial orientation.

density start moving first, filling the density/gradient histogram horizontally from *left to right*. Using descending order, voxels with *high* density move earlier, and the histogram is filled horizontally from *right to left*. In Figure 7.5 the gradient magnitude is used to delay the voxels. In ascending order, voxels with *low* gradient magnitude start moving first, filling the density/gradient histogram horizontally from *bottom to top*. In contrast, when using descending order, voxels with *high* gradient magnitude start moving first, and the histogram is filled from *top to bottom*. Figures 7.6, 7.7, and 7.8 show the results when the delay is based on the voxels' coordinates. Depending on ascending or descending order, voxels with lower or higher coordinate values start moving first, respectively. Regardless of the selected order, all voxels of the same slice move simultaneously. Using an optimal delay (see Section 5.3.1), only a single slice moves at each

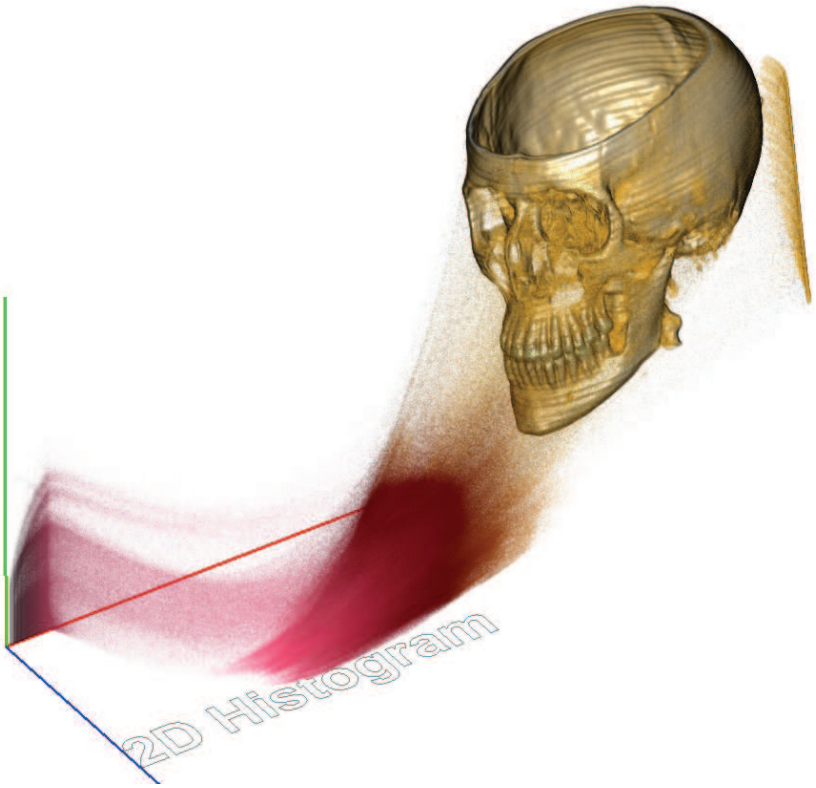


Figure 7.2: The screen shot was taken for an interpolation parameter $t = 0.2$. In addition to the two points used for linear interpolation, a third point is required for quadratic interpolation. The position of the voxel in the histogram locally translated by $(0, 0, 1)^T$ was used. The red and green axes are the normalized voxel density and gradient magnitude, respectively. The blue axis was added for spatial orientation.

point in time. The quality of certain animations (especially those delayed by grid coordinates) depends heavily on the chosen camera position and the arrangement of the two representations. In Figure 7.9, the voxels are animated from the volume to the *gradient magnitude histogram* in ascending gradient magnitude order. A bin size of 32 was used for the logarithmic gradient magnitude histogram. Since the transfer function calculates colors based on the voxels' intensities, voxels with the same gradient magnitude values (but different densities) have different colors. In Figure 7.10, the voxels are animated from the volume to the *density histogram* in ascending density order. This means, that voxels with low density start moving first. A bin size of 1 was used for the logarithmic density histogram. Setting the delay parameter d to a sufficiently large value, separates voxels with same intensities into groups. Figure 7.11 shows the first two images of an image sequence of a non-delayed animation (delay $d = 0$), where all voxels move si-

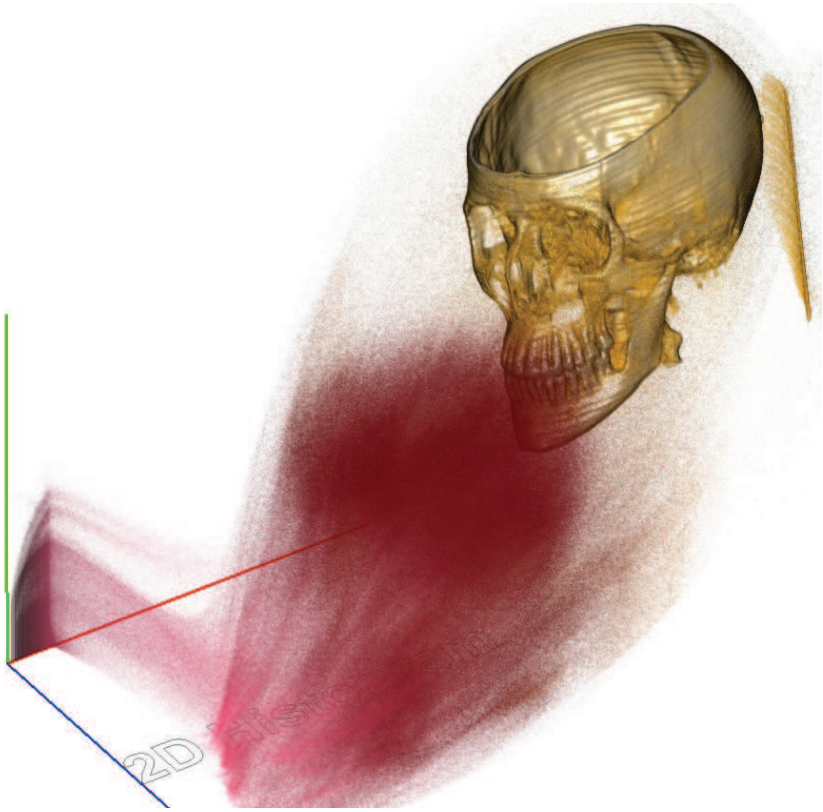
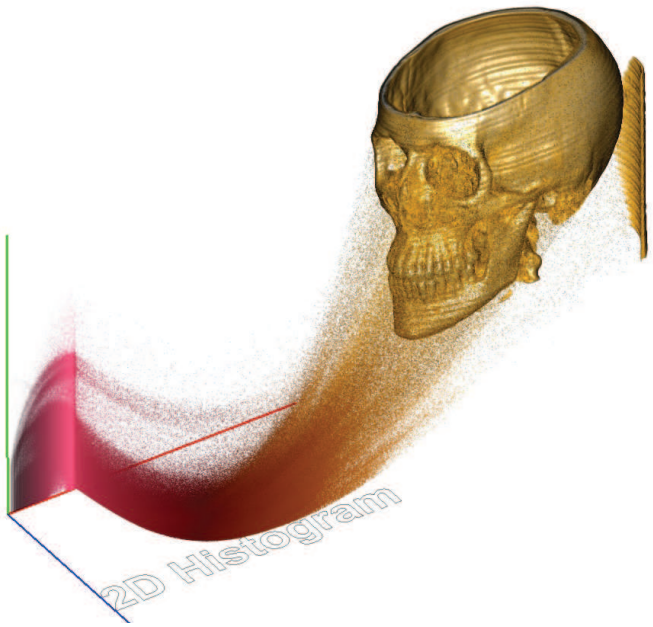


Figure 7.3: The screen shot was taken for an interpolation parameter $t = 0.2$. In addition to the three points used for quadratic interpolation a forth point is required for cubic interpolation. The additional point is the voxel grid coordinate minus the coordinates of the volume center. This makes the voxels travel away from the center of the volume and effectively avoids trajectories running through the volume. The red and green axes are the normalized voxel density and gradient magnitude, respectively. The blue axis was added for spatial orientation.

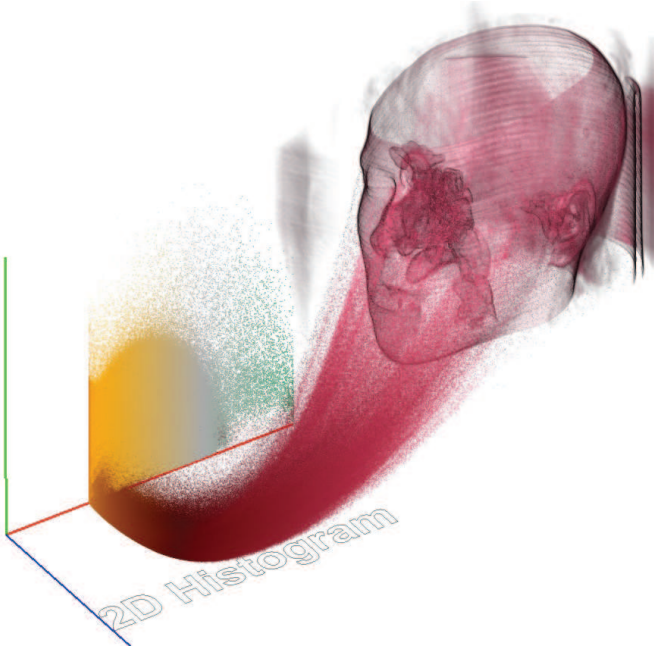
multaneously. The remainder of the image sequence is shown in Figure 7.12. In this case no order can be applied. Finally, Figure 7.13 shows the first two images of an image sequence of an animation using an optimal delay. The last two images of this sequence are shown in Figure 7.14. Using an optimal delay only voxels with the same attribute value travel simultaneously. These are in this example, all the voxels of the same slice.

7.3 Trajectory Rendering

When rendering trajectories of the voxels, a neighborhood of voxels has to be selected from the volume. The center and diameter of this neighborhood can be interactively selected by the user. The voxels contained in this neighborhood are then connected in both representations by their trajectories, which are shaded using the transfer function. Figure 7.15 shows an example of trajectory renderings with different neighborhood sizes. Here, the arcs in the density/gradient magnitude histogram represent transitions between regions of different density (e.g., bones and soft tissue). Voxels with low gradient magnitude reside in a homogeneous neighborhood, whereas voxels with high gradient magnitude reside in a heterogeneous neighborhood. Adjusting the transfer function accordingly, allows the user to select a neighborhood from the interior of the volume. See Section 4.3 for details.

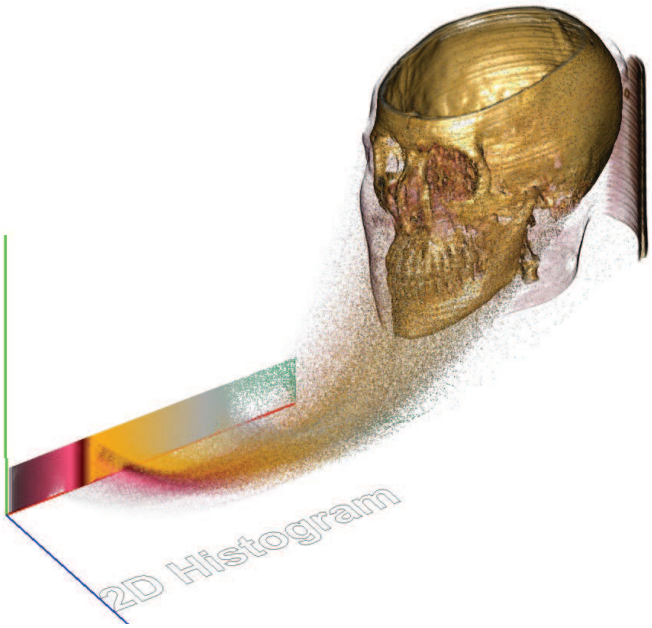


(a) Ascending density

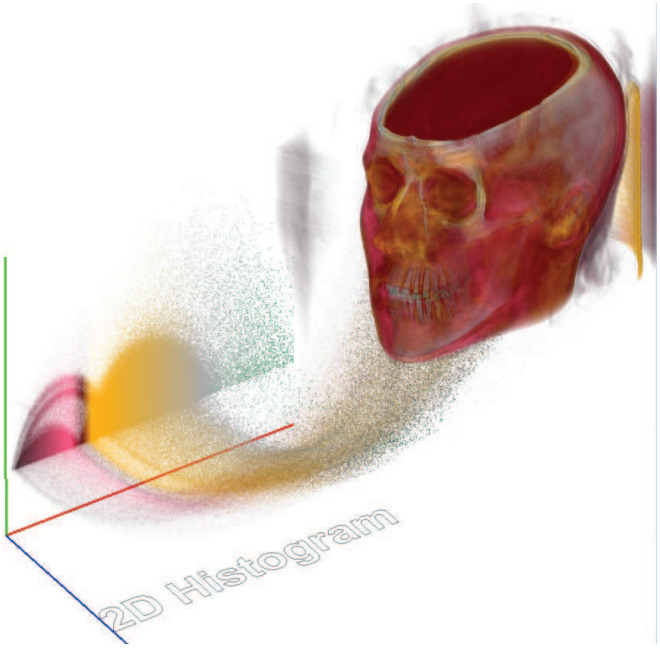


(b) Descending density

Figure 7.4: (a) Voxels with low density start moving first. This can be observed in the density/gradient histogram, which is filled horizontally from left to right. (b) Voxels with high density move earlier, and the histogram is filled from right to left.

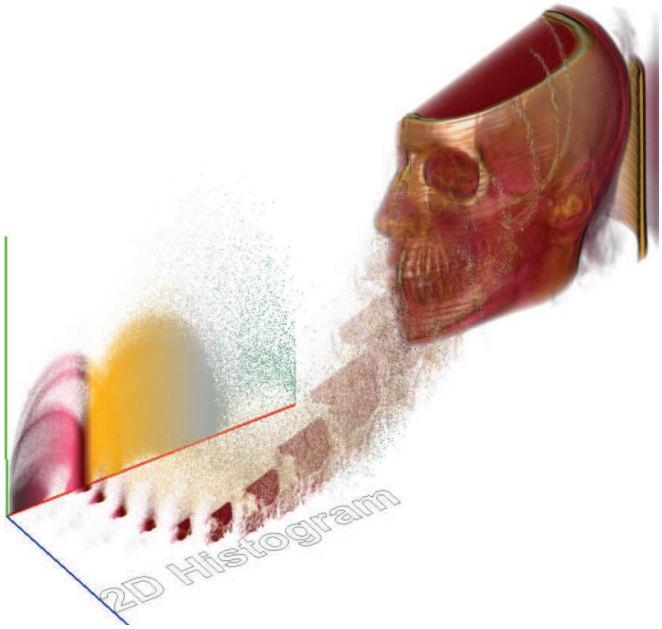


(a) Ascending magnitude

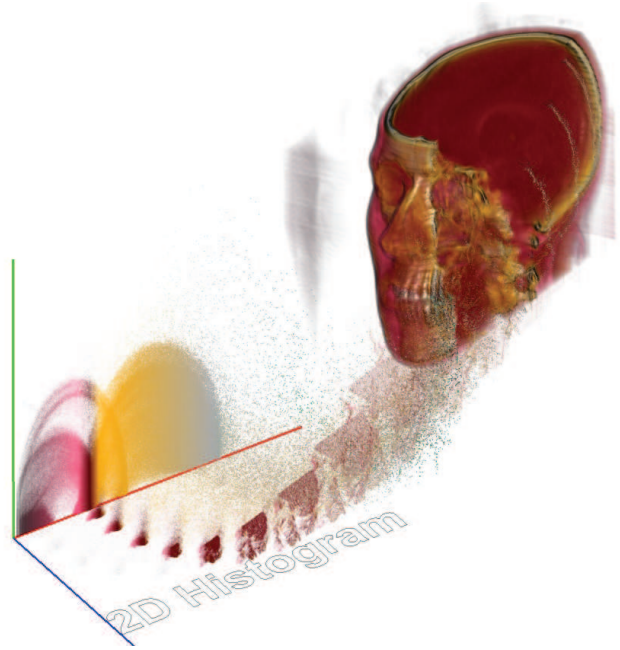


(b) Descending magnitude

Figure 7.5: (a) Voxels with low gradient magnitude start moving first. This can be observed in the density/gradient histogram, which is filled vertically from bottom to top. (b) Voxels with high gradient magnitude move earlier, and the histogram is filled from top to bottom.

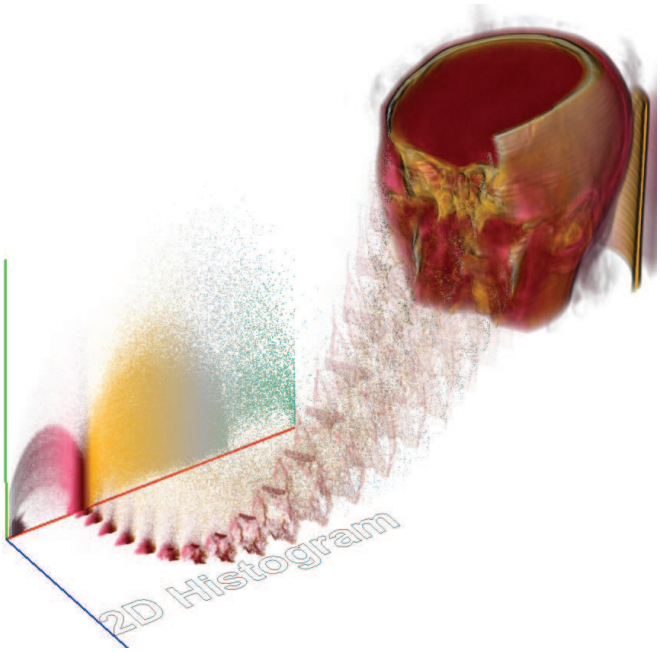


(a) Ascending x-axis

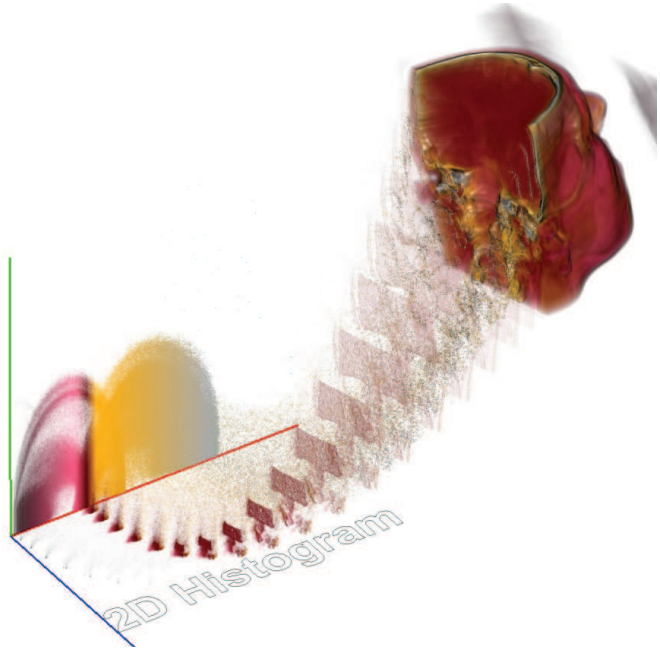


(b) Descending x-axis

Figure 7.6: Voxels with (a) lower or (b) higher x-coordinates start moving first. Voxels of the same slice move simultaneously. Setting the delay to $d = x_{max}$ only a single slice moves at each point in time.

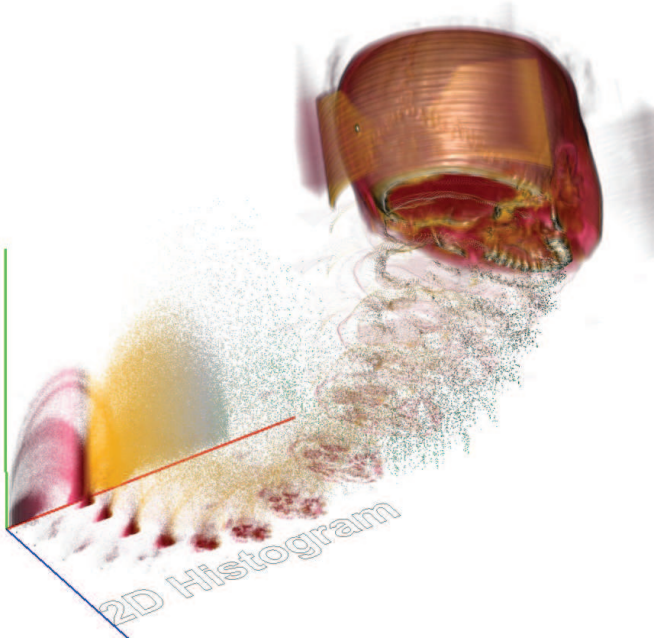


(a) Ascending y-axis

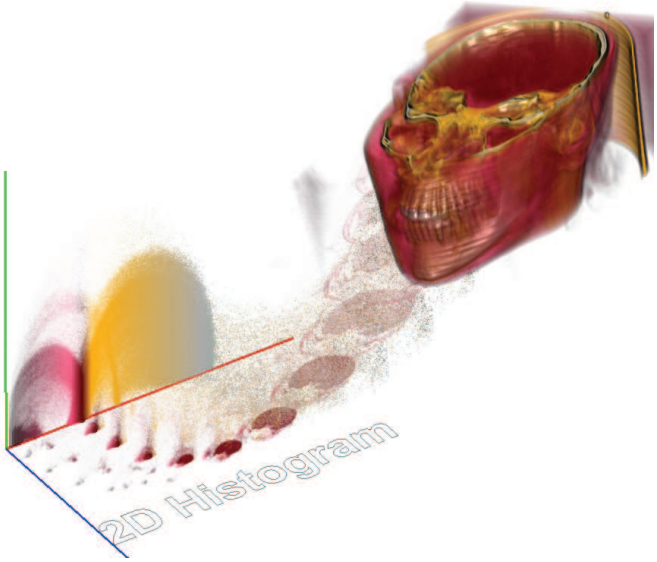


(b) Descending y-axis

Figure 7.7: Voxels with (a) lower or (b) higher y-coordinates start moving first. Voxels of the same slice move simultaneously. Notice, that the volume has been transformed differently in (a) and (b) to give a better view on the animation. When setting the delay to $d = y_{max}$, only a single slice moves at each point in time.



(a) Ascending z-axis



(b) Descending z-axis

Figure 7.8: Voxels with (a) lower or (b) higher z-coordinates start moving first. Voxels of the same slice move simultaneously. Notice, that the volume has been transformed differently in (a) and (b) to give a better view on the animation. When setting the delay to $d = z_{max}$, only a single slice moves at each point in time.

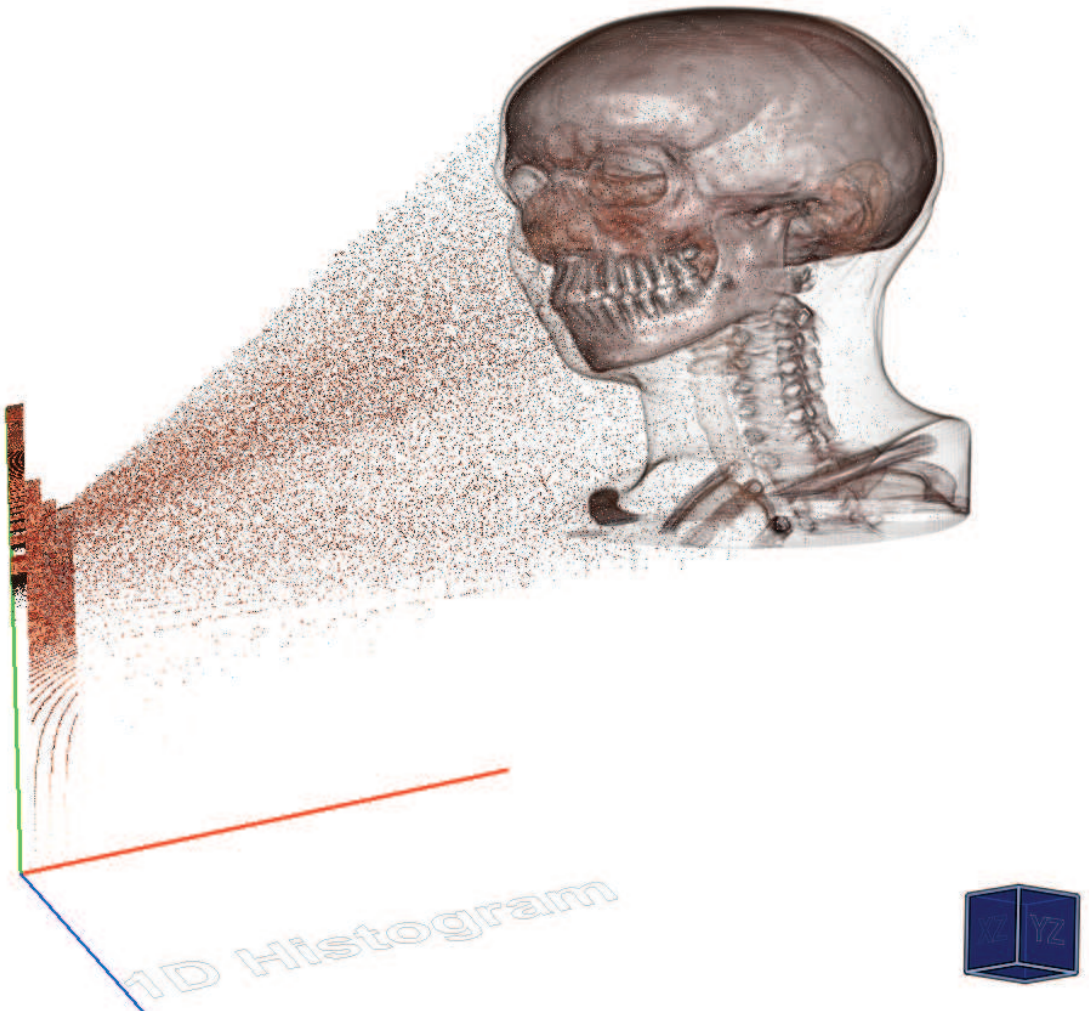


Figure 7.9: The voxels are animated from the volume to the gradient magnitude histogram in ascending gradient magnitude order. This means, that voxels with low gradient magnitude values move first. A bin size of 32 was used for the logarithmic gradient magnitude histogram. Since the transfer function calculates colors based on the voxels' intensities, voxels with the same gradient magnitude values (but different densities) have different colors.

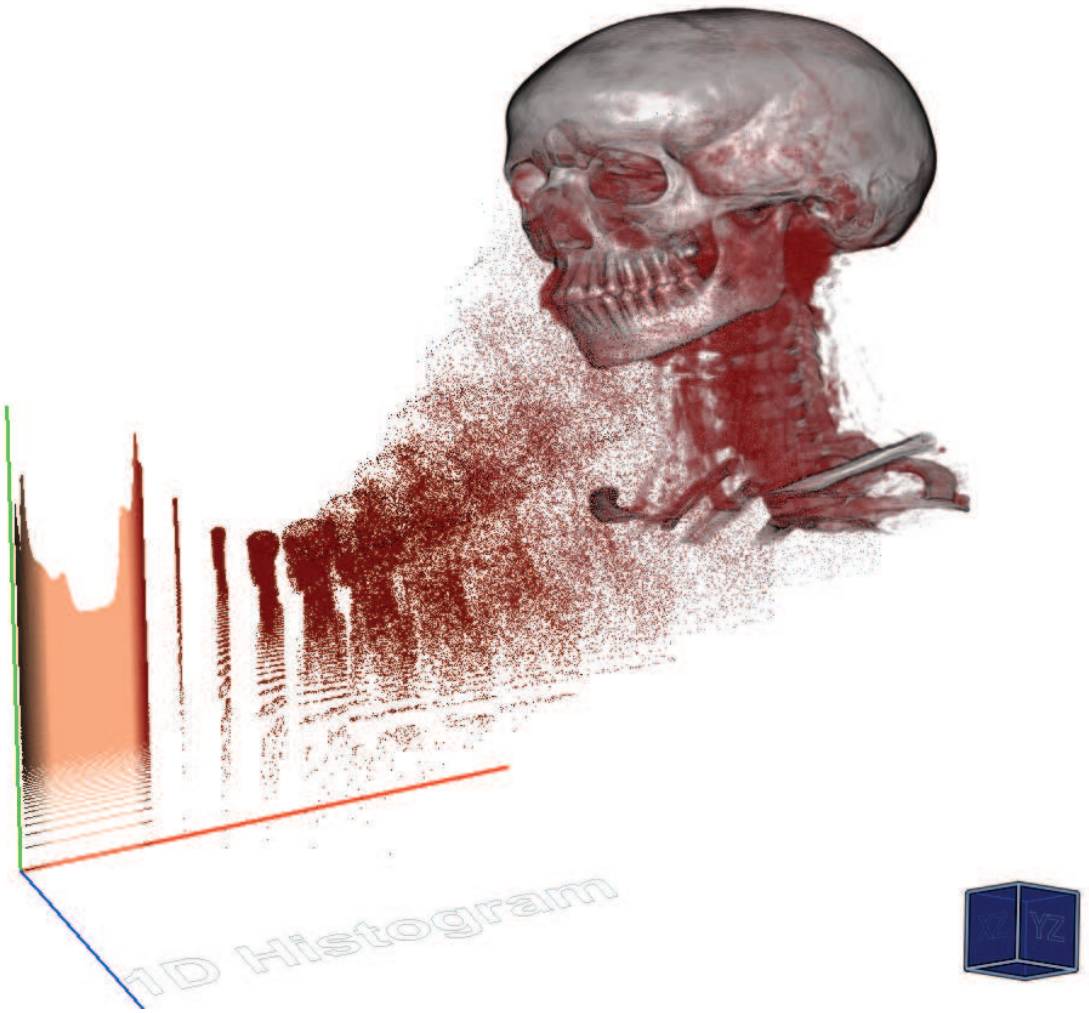
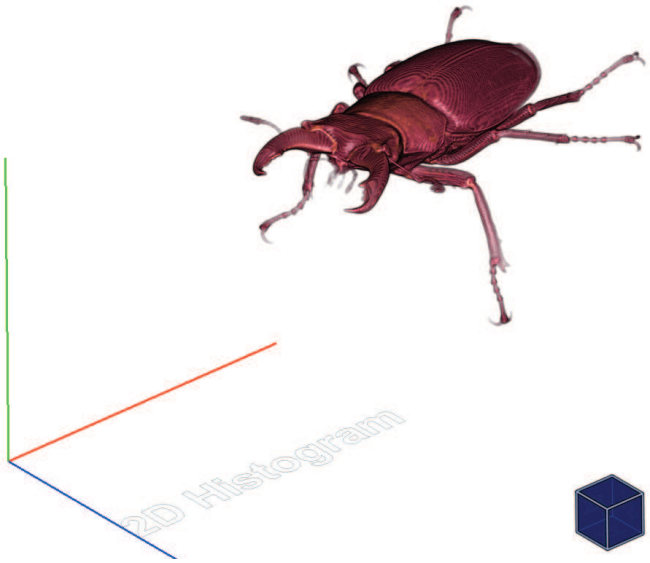
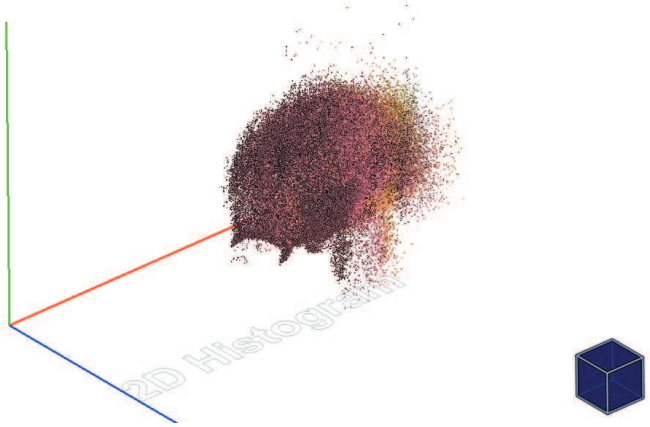


Figure 7.10: The voxels are animated from the volume to the density histogram in ascending density order. This means, that voxels with low density start moving first. A bin size of 1 was used for the logarithmic density histogram. Setting the delay parameter d to a sufficiently large value, separates voxels with same intensities into groups.

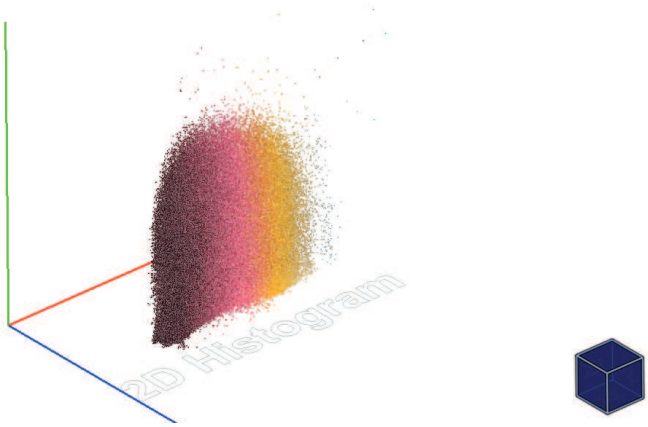


(a) t=0.0

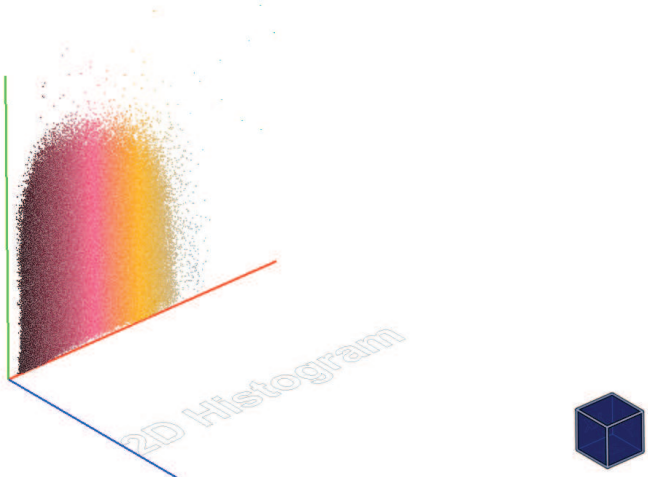


(b) t=0.3

Figure 7.11: When the delay is 0, all voxels start and end their movement at the same time. (a) and (b) show the first two images of the image sequence. The last two images of the sequence are shown in Figure 7.12.

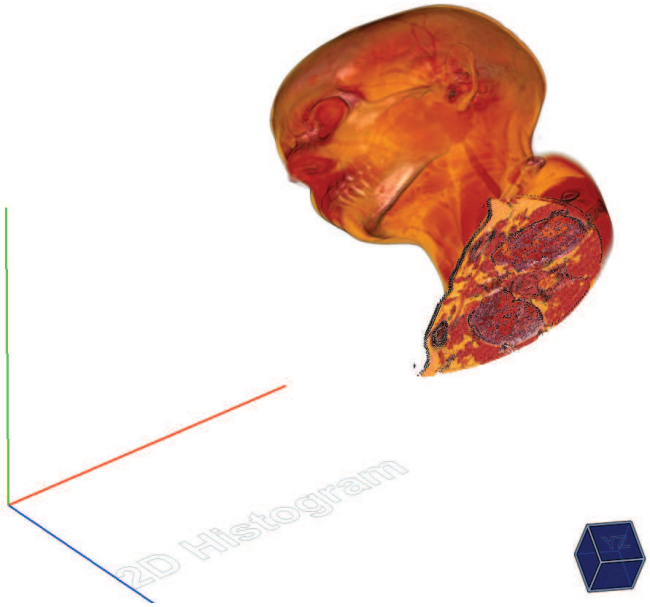


(a) t=0.6

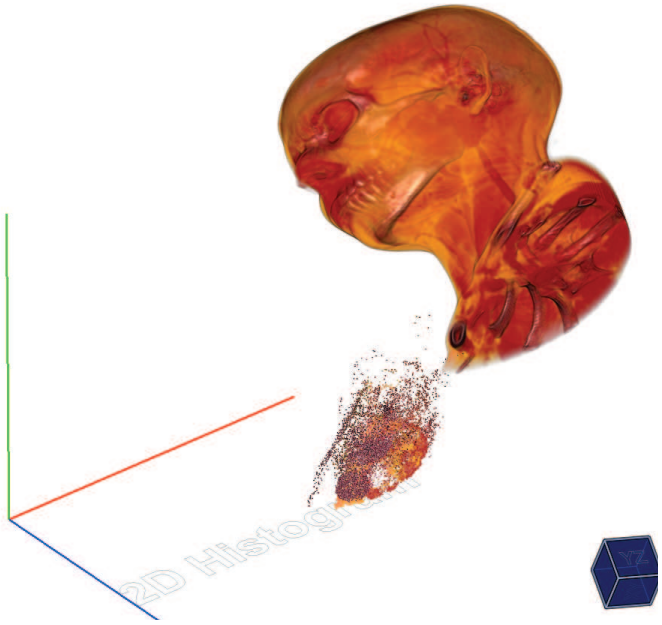


(b) t=1.0

Figure 7.12: When the delay is 0, all voxels start and end their movement at the same time. (a) and (b) show the last two images of the image sequence. The first two images of the sequence are shown in Figure 7.11.

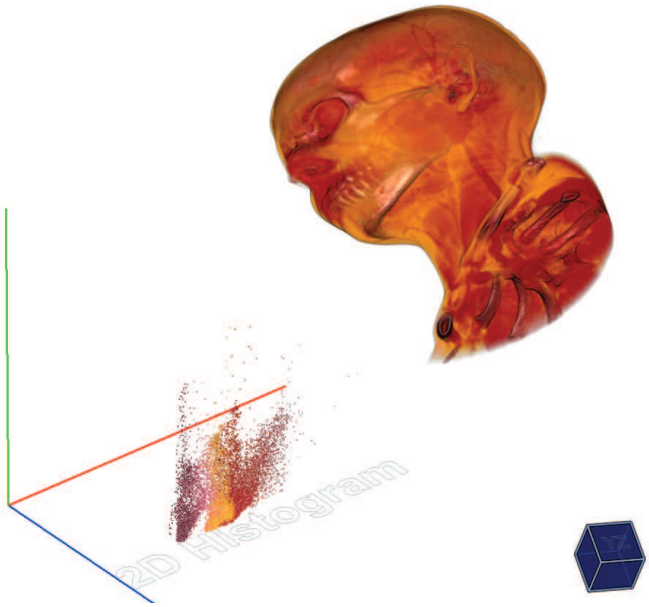


(a) t=0.0001

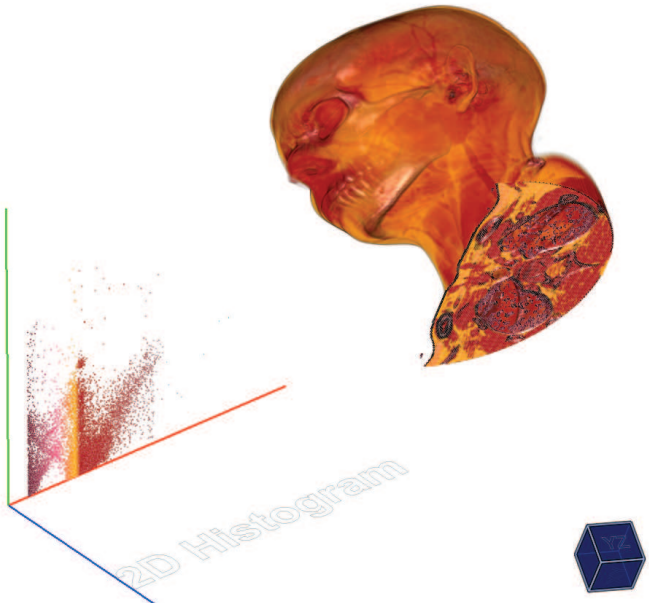


(b) t=0.002

Figure 7.13: When using an optimal delay d_{opt}, only one slice moves at a time. Each slice uses a subinterval of the interpolation parameter of size d_{opt}^{-1}. Since the human head has an optimal delay of 166 for the z-coordinate ($z_{max} = 166$), each slice uses a subinterval of $1/166 \approx 0.006$. (a) and (b) show the first two images of the image sequence. The last two images of the sequence are shown in Figure 7.14.

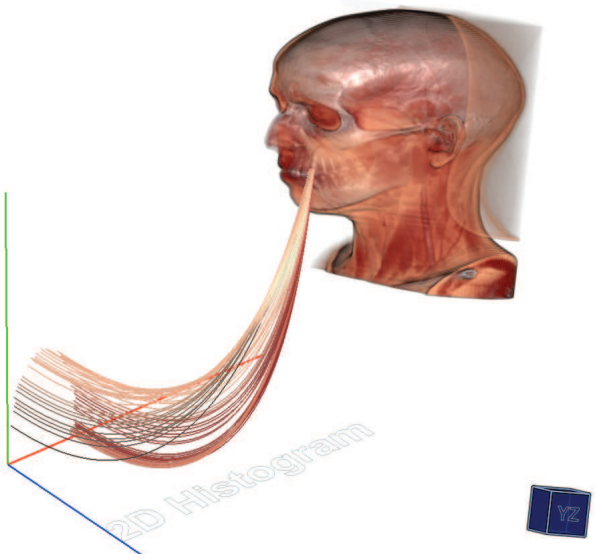


(a) t=0.004

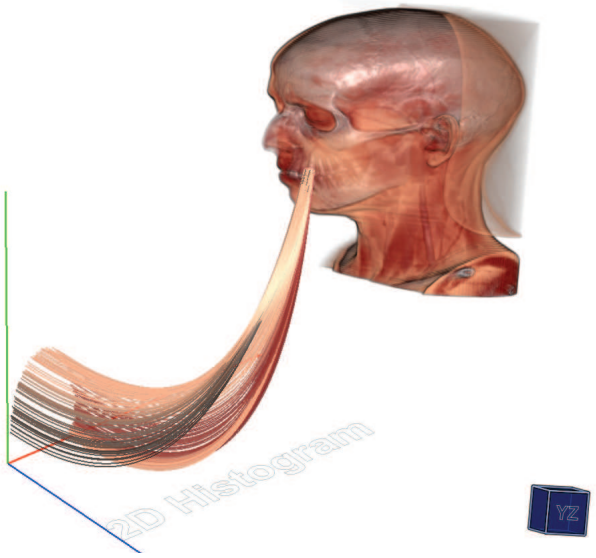


(b) t=0.006

Figure 7.14: When using an optimal delay d_{opt}, only one slice moves at a time. Each slice uses a subinterval of the interpolation parameter of size d_{opt}^{-1}. Since the human head has an optimal delay of 166 for the z-coordinate ($z_{max} = 166$), each slice uses a subinterval of $1/166 \approx 0.006$. (a) and (b) show the last two images of the image sequence. The first two images of the sequence are shown in Figure 7.13.



(a) Neighborhood diameter = 5 voxels



(b) Neighborhood diameter = 10 voxels

Figure 7.15: The trajectory of each voxel of a selected neighborhood is rendered, visually connecting the same voxel in the two representations. It has the same color and transparency as the corresponding voxel. The size and location of the neighborhood can be changed interactively, using the mouse cursor. The arcs in the density/gradient magnitude histogram represent transitions between regions of different density (e.g., bones and soft tissue). Voxels with low gradient magnitude reside in a homogeneous neighborhood, whereas voxels with high gradient magnitude reside in a heterogeneous neighborhood.

CHAPTER 8

Summary

8.1 Introduction

Volumetric data consists of volume elements (voxels), arranged on a 3D equidistant grid. Each voxel has 3D coordinates and an intensity. Additionally, for each voxel, a surface normal can be approximated by the gradient. The gradient is a vector, that points in the direction of the greatest rate of intensity increase. Several representations can then be used to visualize the volumetric data. The most common form are slices perpendicular to one of the major axes. Furthermore, 3D renderings are used to literally gain insight into volumetric data. A more abstract representation of volumetric data are histograms. Here only the intensity and/or gradient magnitude are used yielding three possible histograms: 1D intensity histogram, 1D gradient magnitude histogram, and a 2D intensity/gradient magnitude histogram.

To gain insight into the data, the correlations between these representations need to be examined. In particular the correlation between the 3D rendering and the histograms are of interest. Usually this connection is shown via *Linking and Brushing* techniques, where voxels that are selected in one view also get highlighted in the other ones.

8.2 Animated Transitions

We extended the idea of Linking and Brushing, so instead of highlighting voxels, we moved them from one representation to another one. When animating voxels individually, their paths need to be calculated. This was done using Bézier curves of first, second, or third degree. Bèzier curves were calculated on the GPU using corner-cutting. After examining multiple algorithms, the De Casteljau algorithm was selected, due to its high performance on low degree curves.

For the animation an interpolation parameter $0 \leq t \leq 1$ is used, where 0 and 1 represent the starting and ending point of the animation respectively. The voxels' position is then calculated as a convex combination of the coordinates of the start and the end point. Without any further modifications, changing t automatically changes the position of every voxel, which leads to high occlusion. Delaying the start and end times reduces the number of simultaneously moving objects, and as an effect reduces the number of overlapping voxels. The delay can be based on one of the voxels' normalized attributes: grid coordinates, intensity, or gradient magnitude. For example, voxels with low intensity start moving first, whereas voxels with high intensity start moving later. This delayed interpolation parameter t_a is given by

$$t_a = t(d+1) - d \cdot a, \quad d \geq 0, \tag{8.1}$$

where d is the delaying factor and a is the normalized voxel attribute. If $d = 0$, the interpolation parameter remains unchanged. Using this approach, voxels with the same attribute value move at the same time (e.g., all voxels of the same slice). In order to avoid overlapping of voxels with consecutive attribute values, the delay must not be set too small. On the other hand, to avoid gaps between those voxels, the delay must not be set too high. The optimal delay corresponds to the maximum value of a voxel attribute. In case of the x-coordinate the optimal delay is the volume width, and for the intensity it is the difference between the highest and the lowest occurring intensity.

In addition to delaying the animation, we also implemented a so-called *parameter transfer function*, which allows us to spend more or less time in certain subintervals of the volume. This is particularly useful, if the user wants to spend more time on intervals with many voxels and less time on empty subintervals. This parameter transfer function can be derived from the cumulative distribution function of an attribute through inversion. Of course, this is only useful for unevenly distributed attributes like intensity and gradient magnitude.

8.3 Hybrid Volume Rendering

In order to render the volume as well as individual voxels, a volume splatter was used. For a volume splatter to produce correct renderings, either the volume needs to be sorted in back-to-front order for each frame, or three stacks of slices (one for each major axis) have to be stored in graphics memory. The former requires constant pre-sorting on the CPU, and the latter requires three times the amount of memory. Therefore a hybrid rendering approach was chosen. Here, the volume gets rendered by a raycaster, and as soon as voxels leave their grid position, they are rendered as simple point sprites in no particular order. To get better results the alpha channel of each point sprite is multiplied by a 2D Gaussian kernel. Each voxel is either rendered as point sprite or by the raycaster, but never twice. This approach still requires twice as much memory as a

raycaster, but produces appealing renderings of the volume without constantly sorting of the volume.

8.4 Conclusion

We have presented different alternatives to *Linking and Brushing* for volumetric data. It is difficult to assess the value of animation in revealing correlations between different representations. Due the ephemeral nature of animation, a voxel cannot be displayed in more than one place at once. Depending on the selected delaying attribute this makes it hard to determine the origin and destination of a single voxel. Using this delaying approach, the number of simultaneously moving objects can only be reduced to a certain degree. Voxels with the same attribute value will always move at the same time.

Additional methods to reduce the number of animated objects could be implemented, including windowing and volume segmentation. The animation would then be applied to the selection only. This would drastically reduce the number of simultaneously moving objects and minimize occlusion in the process. Furthermore, acceleration and deceleration at the starting point and the ending point, respectively, could be added.

Bibliography

[1] R. A. Becker and W. S. Cleveland. Brushing scatterplots. *Technometrics*, pages 127–142, 1987.

[2] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the International IEEE Conference on Computer graphics and interactive techniques*, pages 192–198, 1977.

[3] S. Bruckner. Efficient Volume Visualization of Large Medical Datasets. Master's thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2004.

[4] S. Bruckner and M. E. Gröller. VolumeShop: An Interactive System for Direct Volume Illustration. In *Proceedings of the International IEEE Conference on Visualization*, pages 671–678, 2005.

[5] C. Collins and S. Carpendale. VisLink: Revealing Relationships Amongst Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, pages 1192–1199, 2007.

[6] R. Crawfis and N. Max. Direct volume visualization of three-dimensional vector fields. In *Proceedings of the IEEE workshop on Volume visualization*, pages 55–60, 1992.

[7] B. Csebfalvi. *Interactive Volume-Rendering Techniques for Medical Data Visualization*. PhD thesis, Institute of Computer Graphics and Algorithms, Vienna University of Technology, 2001.

[8] P. de Casteljau. Outillages méthodes calcul. Technical report, Citroën, Paris, 1959.

[9] J. Delgado and J. M. Peña. A linear complexity algorithm for the Bernstein basis. In *International Conference on Geometric Modeling and Graphics*, pages 162–167, 2003.

[10] J. Delgado and J. M. Peña. On efficient algorithms for polynomial evaluation in CAGD. In *Monografías del Seminario Matemático García de Galdeano 31*, pages 111–120, 2004.

[11] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001.

[12] M. A. Fisherkeller, J. H. Friedman, and J. W. Tukey. PRIM-9: An Interactive Multi-dimensional Data Display and Analysis System. In *ACM Pacific '75*, pages 140–145, 1975.

[13] M. S. Floater. Mean value coordinates. *Computer Aided Geometry Design*, pages 19–27, 2003.

[14] J. Heer and G. G. Robertson. Animated Transitions in Statistical Data Graphics. *IEEE Transactions on Visualization and Computer Graphics*, pages 1240–1247, 2007.

[15] R. Kosara. Visualization Criticism - The Missing Link Between Information Visualization and Art. In *Proceedings of the International IEEE Conference on Information Visualization*, pages 631–636, 2007.

[16] R. Kosara, G. N. Sahling, and H. Hauser. Linking Scientific and Information Visualization with Interactive 3D Scatterplots. In *Proceedings of the International IEEE Conference on Computer Graphics, Visualization and Computer Vision*, pages 133–140, 2004.

[17] P. Lacroute and M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the International IEEE Conference on Computer graphics and interactive techniques*, pages 451–458, 1994.

[18] A. Law and R. Yagel. An optimal ray traversal scheme for visualizing colossal medical volumes. In *Proceedings of the International IEEE Conference on Visualization in Biomedical Computing*, pages 43–52, 1996.

[19] M. Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, pages 29–37, 1988.

[20] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, pages 245–261, 1990.

[21] M. Levoy. Volume rendering by adaptive refinement. *Visual Computing*, pages 2–7, 1990.

[22] E. Lindholm, M. J. Kilgard, and H. Moreton. A user-programmable vertex engine. In *Proceedings of the International IEEE Conference on Computer graphics and interactive techniques*, pages 149–158, 2001.

[23] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[24] S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. In *Proceedings of the International IEEE Conference on Visualization*, pages 100–107, 1994.

[25] A. R. Martin and M. O. Ward. High Dimensional Brushing for Interactive Exploration of Multivariate Data. In *Proceedings of the International IEEE Conference on Visualization*, pages 271–, 1995.

[26] N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, pages 99–108, 1995.

[27] M. Meissner, J. Huang, D. Bartz, K. Mueller, and R. Crawfis. A practical evaluation of popular volume rendering algorithms. In *Proceedings of the IEEE symposium on Volume visualization*, pages 81–90, 2000.

[28] T. Möller, R. Machiraju, K. Mueller, and R. Yagel. A comparison of normal estimation schemes. In *Proceedings of the International IEEE Conference on Visualization*, pages 19–26, 1997.

[29] K. Mueller and R. Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *Proceedings of the International IEEE Conference on Visualization*, pages 239–245, 1998.

[30] K. Mueller, T. Möller, and R. Crawfis. Splatting without the blur. In *Proceedings of the International IEEE Conference on Visualization*, pages 363–370, 1999.

[31] K. Mueller, N. Shareef, J. Huang, and R. Crawfis. High-Quality Splatting on Rectilinear Grids with Efficient Culling of Occluded Voxels. *IEEE Transactions on Visualization and Computer Graphics*, pages 116–134, 1999.

[32] N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In *Fourth International Workshop on Volume Graphics*, pages 197–242, 2005.

[33] S. Parker, M. Parker, Y. Livnat, P.-P. Sloan, C. Hansen, and P. Shirley. Interactive ray tracing for volume visualization. *IEEE Transactions on Visualization and Computer Graphics*, pages 238 –250, 1999.

[34] H. Pfister. Hardware-Accelerated Volume Rendering. In *The Visualization Handbook*, chapter 11, pages 229–260. Elsevier, 2004.

[35] B. T. Phong. *Illumination for computer-generated images*. PhD thesis, The University of Utah, 1973.

[36] T. Porter and T. Duff. Compositing digital images. *SIGGRAPH Computer Graphics*, 18:253–259, 1984.

[37] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, 2000.

[38] G. G. Robertson, J. D. Mackinlay, and S. K. Card. Information visualization using 3d interactive animation. In *Proceedings of the SIGCHI conference on Human factors in computing systems: Reaching through technology*, pages 461–462, 1991.

[39] T. W. Sederberg. Computer Aided Geometric Design Course Notes. `http://www.tsplines.com/educationportal.html`, last retrieved on 24.08.2011.

[40] K. R. Subramanian and D. S. Fussell. Applying space subdivision techniques to volume rendering. In *Proceedings of the International IEEE Conference on Visualization*, pages 150–159, 1990.

[41] B. Tversky, J. Bauer Morrison, and M. Betrancourt. Animation: can it facilitate? *International Journal of Human-Computer Studies*, pages 247–262, 2002.

[42] T. van Walsum, A. J. S. Hin, Versloot J., and F. H. Post. Efficient Hybrid Rendering of Volume Data and Polygons. In *Advances in Scientific Visualization*, pages 83–96. Springer-Verlag, 1992.

[43] F. Vega-Higuera, P. Hastreiter, R. Fahlbusch, and G. Greiner. High performance volume splatting for visualization of neurovascular data. In *Proceedings of the International IEEE Conference on Visualization*, pages 271–278, 2005.

[44] M.O. Ward. XmdvTool: integrating multiple methods for visualizing multivariate data. In *Proceedings of the International IEEE Conference on Visualization*, pages 326–333, 1994.

[45] J. Warren, S. Schaefer, A. Hirani, and M. Desbrun. Barycentric coordinates for convex sets. *Advances in Computational Mathematics*, pages 319–338, 2007.

[46] L. Westover. Interactive volume rendering. In *Proceedings of the Chapel Hill workshop on Volume visualization*, pages 9–16, 1989.

[47] L. Westover. Footprint evaluation for volume rendering. *SIGGRAPH Computer Graphics*, pages 367–376, 1990.

[48] Wikipedia. De Casteljau's algorithm. `http://en.wikipedia.org/wiki/De_Casteljau's_algorithm`, last retrieved on 24.08.2011.

[49] R. Yagel and Z. Shi. Accelerating volume animation by space-leaping. In *Proceedings of the International IEEE Conference on Visualization*, pages 62–69, 1993.

[50] M. Zöckler, D. Stalling, and H.-C. Hege. Interactive visualization of 3D-vector fields using illuminated stream lines. In *Proceedings of the International IEEE Conference on Visualization*, pages 107–113, 1996.

[51] M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA volume splatting. In *Proceedings of the International IEEE Conference on Visualization*, pages 29–36, 2001.

List of Figures

List of Tables