Illustrative Rendering of
White Matter Fiber Bundles

by
R.J.G. Otten

Supervisors:
dr. A. Vilanova Bartroli (TU/e)
dr.ir. H.M.M. van de Wetering (TU/e)

Eindhoven, November 9, 2009
Abstract

Since the 1970s the science and technology behind magnetic resonance imaging (MRI) has steadily become more advanced. With today’s MRI technology, we can scan the human brain and record enough data to reconstruct tissue structures of the brain in a virtual 3D environment. These kind of medical visualizations allow for exploratory analysis of the scan, which can be a useful tool for the study of anatomy or pathology.

However, such visualizations can quickly become too complex to manage for the viewer, resulting in reduced insight. This is especially a problem when dealing with large and dense sets of data, such as those referencing the entire white matter of the brain in whole-brain visualizations. A common approach to make more sense of such large data sets is to cluster the reconstructed white matter into bundles with anatomic meaning. While such clustering methods alleviate the issue somewhat, they generally continue to visually represent the bundle by its individual tissue structures, which is still cluttered.

We investigate a novel visualization method by which to render such bundles, based on illustrative rendering. Illustrative rendering specializes in producing stylized images which allow for a great deal of abstraction. We aim to use guidelines and techniques from the traditional fields of hand drawn illustration to provide a more abstract illustrative visualization of the clustered white matter. The rendering technique will be accompanied by a focus & context technique based on the paradigm of exploded views. It interactively shifts the positions of contextual bundles in a non-obtrusive way, making room for viewing the focused bundle fully.

This visualization enables us to see and work with the data on the higher level of bundles, rather than on that of the individually visualized white matter tissue.

Keywords

illustrative rendering, DTI, MRI, fibers, tracts, fiber bundles
Contents

Abstract iii

Contents vi

List of abbreviations vii

1 Introduction 1
  1.1 Terminology 2
  1.2 Motivation 2
  1.3 Objectives 3

2 DTI data and visualization 4
  2.1 Diffusion Tensor Imaging 4
  2.2 Visualization techniques for fiber tracts 5
    2.2.1 Clustering fiber tracts into bundles 6
    2.2.2 The merit of illustrative rendering 8

3 Related work 9
  3.1 Techniques employed in hand drawn medical illustration 9
  3.2 Automated illustrative techniques for fiber tract bundles 11

4 An illustrative rendering method for fiber bundles 14
  4.1 General overview 14
    4.1.1 Generating the required components 16
  4.2 Generating hint lines 16
    4.2.1 Constructing a halo 17
  4.3 Extracting a silhouette and outline from fiber halos 21
    4.3.1 The dilation operator 21
    4.3.2 Dilating the fiber halos into a silhouette 23
    4.3.3 Adding an outline to the silhouette 25
    4.3.4 Self-occluding fiber bundles and contours 26
  4.4 Alternate hint line representations 32
    4.4.1 Lighting 32
    4.4.2 Line thickness 33
  4.5 A GPU-based implementation 36
    4.5.1 Global structure of the pipeline 36
    4.5.2 Rendering halos with a geometry shader 37
List of abbreviations

API  Application Programming Interface
BMIA  Biomedical Image Analysis
CPU  Central Processing Unit
DTI  Diffusion Tensor Imaging
GPGPU  General-Purpose computation on Graphics Processing Units
GPU  Graphics Processing Unit
MRI  Magnetic Resonance Imaging
OpenGL  Open Graphics Library
RAII  Resource Acquisition Is Initialization
RGB  Red Green Blue
SLI  Scalable Link Interface
STL  Standard Template Library
UML  Universal Modeling Language
VTK  Visualization Tool Kit
Chapter 1

Introduction

“As the Chinese say, 1001 words is worth more than a picture.”
- John McCarthy

Man has always been a visually oriented creature. Starting with the cave drawings of early man and up to the modern era have we communicated through visual aids. It is in our nature to extract information through the interpretation of images. In modern times we coined the saying that ‘a picture is worth a thousand words’ to reflect this notion. It is however, when we have it recite the likes of the Complete Works of William Shakespeare, that the cracks start appearing. Pictures have a limited capacity to store information just as words do. Comprehending a picture becomes ever harder as it grows in complexity, until we reach some point of ‘critical mess’ where the meaning is just completely lost.

Encoding massive amounts of information into an image in a comprehensible way, requires a very specific school of thought. Figuring out how to efficiently make use of the space in an image and how to create insightful illustrations is in fact an art: The art of the professional illustrator. The art of illustration is exceptional at abstracting and representing the essence of things, while leaving out the annoying ‘not really interested in that now’-details, that tend to clutter an image and make it incomprehensible. The creed ‘less is more’ comes to mind. Naturally, the same holds true for medical illustrations. From sketches of the 15th century Renaissance to illustrations of today, schematic illustrations are an important part of medical literature: illustrations are so clearer than real life photographs and give better insight into the illustrated subject than photographs would.

Since magnetic resonance imaging (MRI) arrived in the 1970s, the methods by which we can extract data through this technology has been steadily improving. Today we have advanced MRI technology such as diffusion tensor imaging (DTI). Computer graphics have not lagged behind and our options to visualize and work with these data have likewise expanded. However, we see them beginning to buckle under the sheer volume of data. Wanting to keep a clear overview is one of the reasons clustering is applied to such data, but even then a crowded whole-brain visualization tends to clutter and become incomprehensible.

For this thesis, information removal and focus and context in hand drawn illustrations were studied. Illustrations from professional medical illustrators were examined and counsel from experts was sought. The goal of this thesis is to devise a new visualization technique from the principles and ideas learned. A technique that can be automatically applied to produce a visualization of the human brain’s white matter based on tracts extracted from medical DTI data sets. The envisioned result should be capable of providing stylized illustrations of the
INTRODUCTION

1.2

The work presented in this thesis spans a number of research areas, each using its own jargon. Here we will introduce a few common terms used throughout the thesis.

The research challenge we are trying to solve is based on the visualization of a three dimensional set of data, also called a volume, obtained from making a magnetic resonance imaging (MRI) scan of the human brain. Specifically the portion of the brain referred to as the white matter, which connects the various sections of the brain’s grey matter. Think of the brain as a network of computers: each section of grey matter being a computer and the white matter being the network cables that patch them all together.

A common method of visualizing such a data set involves the three dimensional reconstruction of the scanned tissue as a set of dense lines. Commonly referred to as streamlines (following a particular flow or stream in a volume), in the context of reconstructing data from an MRI scan they are referred to as fibers (as in the fibers of a tissue sample). The particular field of research involved with creating these kind of reconstructions is tractography, which is why fibers are sometimes also referred to as tracts or as, somewhat tautological, fiber tracts. When clustering is performed on fibers, either automated or by hand, the resulting clusters are often referred to as bundles or fiber bundles.

The work presented centers around the development of a specific rendering technique for such bundles. In the context of computer graphics, the term ‘rendering’ stands for the production of a picture through some form of underlying process that colors picture elements, commonly referred to as pixels. This does not strictly say anything about the type of picture produced. In many cases the goal is to create a picture that is life like, as if we would have taken a photograph. This process is referred to as photorealistic rendering. Our work focuses on the opposite. We try to produce schematic illustrations which are heavily stylized to emphasize clarity and comprehensibility. For a long time, the production of such stylized images has been referred to as non-photorealistic rendering. Apparently the term was coined by Georges Winkenbach and David Salesin in a 1994 paper [37] and has received criticism for the poor choice of words ever since. A more contemporary reference to the same field of research uses the term illustrative rendering: producing pictures that are stylized after illustrations hand drawn by illustrators.

Therefore, the rendering technique we present in this thesis is called an illustrative rendering technique.

1.2 Motivation

Medical volumes of white matter can quickly become complex beasts to visualize in a comprehensible fashion. A multitude of filtering techniques can be used to cut down the visual complexity. Some options involve cutting planes or slabs, thresholds on several parameters of the visualization, etc. The use of clustering algorithms too can segment the data into smaller components that can, for instance, be selectively filtered out. All these mean a large part of contextual data is invariably lost.
1.3 Objectives

Technical illustrators and medical illustrators have mastered the art of simplification, being able to leave out details that lead to cluttering and incomprehensibility, while leaving enough details to propel the intention of the illustration. We are interested in employing their domain knowledge to develop an illustrative rendering technique that provides clean and comprehensible schematic illustrations of the bundles in the white matter of the brain.

With such a technique it should be possible to comprehensibly visualize the entire white matter. This will in turn allow us to develop a focus & context technique for navigating the white matter and producing quality pictures of its structure where particular parts are highlighted and also a suitable context of the remaining white matter remains.

The main objective of this thesis is the development of an illustrative rendering technique that can be used to browse and interact with a comprehensible, clustered visualization of white matter of the human brain in an exploratory fashion.

First we look to existing techniques for schematic drawings in medical illustration. We subsequently distill those into an illustrative rendering technique. Since the technique should be used interactively, we create a hardware-accelerated implementation of the rendering technique, refitting the algorithms to run on the graphics card.

A secondary objective is the development of a focus & context system that can be paired with the rendering technique to navigate clustered visualizations of the white matter. We should be able to focus on individual clusters while retaining the other clusters as an unobtrusive context.

Lastly we would like to integrate the rendering technique with DTITool: an existing software framework designed to process MRI volumes that is in development at the Biomedical Image Analysis (BMIA) research group at the Eindhoven University of Technology. To this end, a prototype of the technique should be based on the same line of development tools: the C++ language and the QT and VTK libraries.
Chapter 2

DTI data and visualization

This chapter contains necessary background information on diffusion tensor imaging, the methods by which fiber tracts are formed and visualized and explains clustering and its relation to the development of our illustrative rendering technique for fiber bundles.

2.1 Diffusion Tensor Imaging

Diffusion tensor imaging (DTI) is a type of magnetic resonance imaging (MRI) technique that measures the diffusion of water molecules.

A water molecule suffers constant random collision with other molecules, which is a phenomenon called Brownian motion. In pure water (figure 2.1a) this phenomenon is described by an isotropic Gaussian distribution of water molecules [36]. In more fibrous, organized tissue (figure 2.1b), the Gaussian distribution is anisotropic: it is larger in directions that follow the inner structure of the sampled tissue.

DTI is a tensor-based MRI technique, which capitalizes on this particular quality by recording the covariance matrices of the diffusion’s measured Gaussian distributions in so-called diffusion tensors [36]. Through analysis of these tensors we can gain improved insight into the orientation of the fibrous, organized tissue that was scanned.

![Figure 2.1: Diffusion of water molecules](image)

Although DTI has existed for a comparatively short time, it sees quite some use in the field thanks to the increased capabilities it offers. It is an important tool in studying the anatomy and pathology of the brain’s white matter. However, in contrast to weighted techniques,
the data is harder to interpret directly and requires more significant processing to turn into suitable visualizations.

2.2 Visualization techniques for fiber tracts

One of the most popular types of pre-processing to perform on a DTI data set is a fiber tracking operation, from the field of tractography. The additional data on orientation present in a DTI data set is used to plot fibers through the data set volume.

In its simplest form, we select a given point in the data volume as a seed, analyze the main diffusion direction at that point and step to the next data point in that direction. We repeat this simple process until we hit some kind of stop criterion. Stop criteria can include a given maximum length or minimum linear anisotropy. More advanced methods exist, but discussing these is outside the scope of this thesis.

The result of this reconstruction is a three dimensional set of polylines that is commonly used to render an interpretation of the structure of the brain’s white matter. An important note we make here is that fibers are indeed merely an abstracted interpretation of the actual structure. A fiber does not represent an actual physical neural tract. These are too small to be measured by DTI, but thanks to their coherence fiber tracking can still be used to produce fibers that each represent multiple neural tracts in the direct vicinity of the fiber.

Figure 2.2 shows common ways in which such fibers are represented: streamlines (polylines), streamtubes and streamsurfaces.

Streamlines
Rendering the reconstructed fibers using a direct line primitive is probably the most common method by which fibers are visualized and one of the fastest. However, they tend to clutter quickly when large amounts of them are drawn. Streamtubes used to be favored over polylines for their capability to be properly lit, which reduces this clutter by adding additional three dimensional insight. However, with current graphics hardware it has become possible to properly apply lighting and even real-time shading to polylines\[26\].

Streamtubes
Streamtubes are an extension of the simple streamline. Rather than a direct line primitive, a polygonal mesh is used for the representation. The name comes from the fact that the mesh is tube-shaped, which creates ‘lines’ with an actual thickness. Various types of lighting can easily be applied to a tube’s surface, which combined with the thickness mitigates the effect of tangled line crossings. One can clearly discern which tube overlaps the other when tubes cross.

While the increased amount of geometry make them slower to render than polylines, streamtubes allow for more data to be embedded into the visualization. Tubes can be given elliptical cross sections, where the two axes of the ellipsoid are controlled by the secondary and tertiary direction of diffusion. This can give visual feedback on both the amount of diffusion and the direction as the tube also twists along with the secondary and tertiary directions that change the orientation of the cross section\[31, 12\]. They are hyperstreamlines\[12\].
Streamsurfaces

Streamsurfaces [38] take a different approach altogether and construct a surface erected along the primary and secondary directions of diffusion in areas where anisotropy is planar, rather than linear. They can be successfully married to fiber-based streamtubes [35], fluidly transferring from a region of linear anisotropy rendered by tubes to a region of planar anisotropy rendered by surfaces.

Color is most commonly used to indicate properties such as diffusion direction \((r, g, b) \leftarrow (x, y, z)\), for instance) or scalar value derived from one of various kinds of anisotropy [36].

A true geometric representation helps with occlusion related problems. Lighting and shading help with three dimensional insight in general. However, none of the above rendering methods truly resolve the issue of clutter, caused by the sheer amount of complex, entangled 3D data being represented all at once.

Figure 2.2: Several ways to visualize fiber tracts

2.2.1 Clustering fiber tracts into bundles

If fibers are reconstructed and subsequently visualized individually, the display quickly becomes cluttered. As the data set becomes larger and more complex, it becomes ever increasingly difficult to obtain insight, to the point where the visualization ends up largely useless. This is especially true in cases such as figure 2.3 where the entire white matter is being treated.
2.2 VISUALIZATION TECHNIQUES FOR FIBER TRACTS

Figure 2.3: A fiber reconstruction of the entire white matter, threshold applied to the anisotropy to ignore noise.

It is possible to cluster such a tangled mass of fibers into bundles with anatomical meaning, either by hand or automated based on some selection criteria [25]. Current automated processes are not perfect and still require manual configuration and tuning.

Figure 2.4: A possible result of clustering fibers into bundles, taken from Moberts et al. [25]

A possible result we show in figure 2.4 is a visualization where the fibers have been split off into separate bundles, identified by different colors. This partly resolves the issue of clutter: at least areas with many bundles present will seem less cluttered now that individual fibers
can be identified and attributed to the individual bundles. However, it does little for the inherent complexity: everything is really still a large tangled mass, just a tangled mass in different colors.

### 2.2.2 The merit of illustrative rendering

What we want to use to resolve the issues of clutter and inherent complexity is a method of representing a fiber bundle, where we are no longer bothered by individual fibers, but can reason about fiber bundles. We do not want to bother with the path of each individual fiber. As we previously stated: a fiber is only an abstract interpretation of orientation of tissue in its direct area. It is merely a concept to help us reason about the white matter tissue.

Therefore, why not allow the viewer to reason about the fiber bundles as a whole? We want to develop an illustrative rendering technique that allows us to make more obvious the bundle structure and go away from the single fiber visualization.

A *schematic illustration* of a fiber bundle, where we abstract from the fibers running inside the bundle, would offer a significantly cleaner image, reducing the amount of clutter resulting from densely packed fibers as for instance figure 2.4 still shows. In addition, the use of clear bounding shapes as common in schematic illustrations can greatly enhance the visual separation of the clustered data. It not only convenes the subdivision of the data into individual bundles clearly, but also convenes clearly which fibers come together to form a cluster.
Chapter 3

Related work

In this chapter we take a look at existing illustrative rendering techniques applicable to fiber tracts. We study the problem from the view point of a medical illustrator and compare existing solutions to the techniques used by artists to produce clear and comprehensible illustrations.

3.1 Techniques employed in hand drawn medical illustration

To establish an automated illustrative rendering technique we can not ignore the insight and view of the artist, who has been dealing with the issue of producing quality images for years. Since we are looking to render white matter, we look at how white matter or fibrous tissue in general is commonly illustrated by medical illustrators.

In practice there seem to be two distinct branches we can identify. On the one side illustrators produce highly detailed, almost realistic artwork. Often this kind of artwork portrays anatomy of the body or its organs in diagrammatic sections, accompanied by legends indicating the individual components. On the other side illustrators produce artwork that is closer to schematics or diagrams. This is often the case when an illustration has a very specific goal. Such is for instance the case with diagnostic or pathologic images indicating disease or injury. This kind of artwork is also used often in the illustration of guidebooks for surgical procedures, another good example.

The second category of schematic artwork is what we are interested in. It matches our goal of reducing clutter by removing unnecessary detail and enhancing the aspect of the larger fiber bundle structures.

First, we take a look at a few general properties of schematic artwork. Take for example, figure 3.1 depicting the human brain. Note the use of large, clear shapes where detailed artwork would fill in details. There is a clear contrast between filling and line in these shapes, leaving no mistake as to the structure of the object being depicted.

Color is used with reservation, with most of the image consisting of a soft tan color. This is in contrast with regular visualization approaches, where the full range of available colors is often used to encode additional information into the image. A two-tonal model like that in this image is often encountered: outlines and contours are drawn in a darker color that heavily contrasts with the background (which is almost exclusively white or near white) and the fill color of the shape itself is a lighter color that contrasts much less with the background.

Typically, the hues of both colors will still match closely, making one a darker (or lighter) version of the other and connecting both as belonging to the same object. Stronger color
is used to highlight a few key areas of interest, where similarly colored structures indicate a relationship. Side features that are only there for context are generally set in subdued, softer tones.

Finally, lighting and shading is non-existent. For figure 3.1a, which illustrates a cross section, one could argue this is logical: a cross-section is after all flat. Figure 3.1b is not a flat cross section, but lighting and shading is still absent.

![Figure 3.1: Two schematic illustrations of the human brain, by Patrick J. Lynch, medical illustrator; C. Carl Jaffe, MD, cardiologist, released under the Creative Commons Attribution 2.5 License 2006 (http://creativecommons.org/licenses/by/2.5/)](image)

We are not interested in completely leaving out all detail, only in leaving out superfluous detail. How should we handle the removal of detail regarding the orientation and direction of individual fibers, while retaining enough contextual information regarding the fibers to visualize the orientation of fiber bundles?

An example is given is given by a hand drawn legend for overview shots of rendered DTI fibers from [36] in figure 3.2. Here we see a few evenly spaced lines hinting at a fiber bundle’s internal structure. When we compare this with figure 3.1b, we note a recurring pattern: the cranial nerves in the center of that image (colored in blue) use a similar technique, where emphasis remains on a few of the curving areas and the remainder of the individual fibers is not drawn. A few quickly sketched lines remain to hint at the structure: hint lines, if you will.
3.2 AUTOMATED ILLUSTRATIVE TECHNIQUES FOR FIBER TRACT BUNDLES

In fact, we can find something like this when we look at typical illustrations of other fibrous tissue as well. Take for instance the muscle groups depicted in figure 3.3, where we see the same sketchy lines. Here we note that a subtle level of light fall has been used to give a 3D bulging look to the muscle. The lighting is subtle though and there are no shadows cast, which keeps everything simple and clean.

3.2 Automated illustrative techniques for fiber tract bundles

Little work has been done in the field of illustrative rendering of fiber-based data. Illustrative rendering for the brain in general, combined with fiber-based data, has been used in work done by Born et al. [4] who integrate contours and half-tone shading of a transparent dome.
of gray matter with a tract-based visualization and activation hot spots.

Though the end result is pleasing to the eye, we note that the fiber tracts are little more than a byproduct for context, given form by simple polygonal streamtubes. Of course, they endeavored to research the integration of fiber tracts into an existing illustrative rendering method [19] for brain activation areas, not to solve the complexity of a fiber representation of the brain white matter.

Only very recently, in parallel to the research and development for the masters project that this thesis belongs to, has illustrative rendering been used for large volumes of fiber tracts. It is the only occurrence we were able to find.

Figure 3.4 shows what can be produced by the method of Everts et al. [14]. They aimed to use principles from handmade pen-and-ink drawings to cleanly visualize all fibers in a data set, foregoing any kind of information removal. It mimics the style of pen art in a realtime 3D environment. As they themselves note; the technique is informally accepted as “beautiful” and their visualizations “show better depth relation and structure” than typical streamline or streamtube visualizations.

![Figure 3.4: Illustrative visualization of a subset of DTI fiber tracts with depth dependent halos, taken from Everts et al. [14]](image)

One fairly fundamental issue their technique does not yet solve is that it suggests that drawn lines represent actual physical fiber tracts in the brain. Something that DTI can not guarantee, for reasons we have noted in section 2.2. The method also shows lines without context, which they suggest might be remedied by pulling in and rendering volumetric data to augment the technique. Selection (and thus identification) of visible subsets is another area in which the technique can be improved, according to the feedback they have received.

Another potential weakness of this pen-and-ink technique that we identify, is that in areas where many lines end up passing by closely in parallel, the area will still be rendered as an indecipherable mess of black, losing any kind of hint at the structure in that place. (Note the black areas in figure 3.4 as example of this, if you will.)

The first three issues are key issues our own illustrative rendering technique should address (and for a large part has addressed). The fourth we noted ourselves, because it is the main issue our rendering technique should resolve: incomprehensibility of the image due to the density of data in the visualization.
We make note that only a few well chosen lines are necessary to effectively describe a surface [22]. This includes the bounding surface of a volume, such as a bundle of fiber tracts. Therefore, we strive to leave out as much superfluous fibers as possible, precisely to prevent the kind of ‘massive wall of lines’-clutter we referenced above from occurring.

The idea of using wide open spacing to prevent such clutter has shown up in algorithms that place evenly spaced streamlines in 2D images produced through Line Integral Convolution (LIC) [20, 21] or trace evenly spaced streamlines in three dimensional space [23]. In these cases even spacing is used both to prevent areas devoid of lines from appearing, as well as to prevent areas from becoming too dense with lines and becoming cluttered.

Appel et al. [1] used an effect coined haloed lines to clear clutter due to crossing lines. The halo they place around a line refers to a polygon that (when the scene has been rendered) surrounds the line in question, like in the traditional sense of the word a halo is an optical effect where a ring of light seems to surround an object. Appel et al. used the technique with success to reduce clutter on images of various kinds of wire frame grids stacked together. While the grids are wire frames, the added halos allow them to treat the grids as opaque surfaces that can occlude underlying surfaces.

Everts et al.[14] use a revised version of this technique, though rather reservedly as they only want to show separation between lines and do not want to actually remove any occluded lines.
Chapter 4

An illustrative rendering method for fiber bundles

To establish an automated illustrative rendering technique for a white matter fiber bundle, we have looked at both hand drawn illustrations in section 3.1 and (parts of) existing automated solutions in section 3.2. From the former we have learned that schematically rendered fiber bundles should a) consist of large, clear shapes, b) have hard, well-defined boundaries, c) hint subtly at their inner details, d) be drawn in soft colors, and e) sparingly employ light fall, for them to form comprehensible, clear visualizations.

In this chapter, we will describe such an automated illustrative rendering method for fiber bundles. We begin with a general overview of the involved components and then go into detail.

4.1 General overview

We want to base our illustrative representation of a fiber bundle on the principles of hand drawn schematic illustrations as previously summarized in this chapter and discussed in full in section 3.1. To sketch the components our rendering technique requires, we begin by summarizing the kind of input we expect our rendering algorithm to take and the desired type of output. We find the following:

**Input** The fiber bundle we will process consists of a list of *polylines*: connected series of line segments. Each polyline in the list represents one fiber in the fiber bundle. When a fiber bundle is constructed (for instance by tracking fibers through a volume of DTI data) it is usually dense, with many fibers packed close together.

**Desired output** The schematic illustrative rendering we will produce may only contain a few hint lines, indicating the general flow of the component fibers in the bundle. It needs an outlined silhouette to convey the overall shape of the bundle, rather than rely on the combined effect of all the fibers in the bundle to convey that shape. A subtle shading or lighting effect could be added to hint at the three dimensional nature of the fiber bundle.
4.1 GENERAL OVERVIEW

We notice two things. Firstly, our input data consists of a dense set of lines, whereas our desired rendition should only contain a few sparse lines. Secondly, we have no clearly defined boundary that can be used as a silhouette. Indeed, we only have a set of lines available to us. Figure 4.1 shows the three things that we need to extract from a) this dense set of polylines,

Figure 4.1: Components of our illustrative rendering technique
namely b) a sparse set of hint lines, c) a silhouette, and d) an outline, which all have to be combined into e) the resulting illustrative rendering of the fiber bundle.

### 4.1.1 Generating the required components

There are several different ways by which we can form these components. Some approaches for generating a silhouette, or any kind of solid surface or volume for a set of polylines rely on pre-processing steps that generate intermediate geometry for later rendering [13, 24]. Instead we propose a novel method that does not require generating any form of intermediate geometry.

The rendering technique we have devised generates all three components in order, using the pipeline model shown in figure 4.2. The first processing step in the pipeline culls fibers from the bundle to a point where only a reasonably small, well-spaced amount of them remain in a rendered image to serve as hint lines. The second processing step in the pipeline takes this output image and uses it to generate both a silhouette and an outline.

![Figure 4.2: Pipeline of processing steps](image)

Figure 4.2: Pipeline of processing steps

The remainder of this section contains an in-depth description of both processing steps. Section 4.2 describes the first step, whereas, section 4.3 describes the second step. Section 4.4 adds details regarding a few different options in rendering the hint lines that remain after culling, while section 4.5 contains notes on implementing the algorithms that form our rendering method on a GPU.

### 4.2 Generating hint lines

The first processing step converts a fiber bundle into a rendered image of a more sparse set of hint lines. When a lot of fibers cross over each other or pass closely in parallel, an image of a fiber bundle becomes terribly cluttered. We want to cull enough of the fibers to unclutter the image, while leaving enough lines to hint at the bundle’s overall flow and orientation of its fibers.

We accomplish this by adding an opaque halo around each polyline, performing a kind of hidden line removal akin to the technique used Appel et al. [1] we referred to in section 3.2. Actual inspiration for this method came from the VolumeFlies [6, 32, 33] framework, where a similar hidden surface removal takes place based on splatting techniques.
The basic idea behind our approach is to simply render all the polylines, augmented with suitably wide halos. Each newly drawn polyline will cull parts of neighboring polylines that are behind its halo. This gradually enforces open space to be created between polylines by iteratively hiding parts of older lines that are too close to newly inserted lines. See figure 4.3 for an illustration of the idea.

![Figure 4.3: Culling fibers using halos](image)

The rendered polylines that survive the culling procedure serves as a good spread of hint lines. How far these lines are spread and how many remain is directly related to the radius $r_{halo}$ (see figure 4.4) of the halos we augmented the polylines with. This makes the density of the hint lines a parameter that can be easily configured to the user’s taste.

With the halos added to our fiber bundle’s fibers, we can render the scene to an image and get a result such as shown in figure 4.2b.

The actual process that creates the halos is 2D in nature, but takes place in 3D world space. A halo can be created in a 3D plane perpendicular to both the view vector and the fiber polyline segments. This plane will of course warp to keep itself aligned with all segments of the fiber. However, after projection onto the screen, it will result in a proper screen-aligned halo surrounding the projected line.

The remainder of this section describes the halo construction process.

**4.2.1 Constructing a halo**

To augment a fiber with a halo, we add a pair of polygonal *fins* that run along the fiber polyline. We cap these fins with half-circles, which gives the halo a nice rounded trim. The fins are segmented: each segment in the polyline receives its own pair of *fin segments*. Figure 4.4 illustrates the structure if we were to take the surface curving along the fiber in 3D space and roll it out into a flat 2D representation.

![Figure 4.4: Overview of the structure of a fiber halo](image)

Building a section of a fin is accomplished as follows. We take the cross product between
the view vector \( \vec{v}_{\text{eye}} \) and the tangent \( \vec{t} \) of the relevant polyline segment. This results in a view-perpendicular normal vector \( \vec{n} \). The tangent and this normal now erect the local 2D plane we referred to earlier.

It is easy to construct the two fin segments for a section of the halo. We treat \( \vec{n} \) and \( \vec{t} \) as an orthogonal base with which the segments align. We use a parameterized width given by radius \( r_{\text{halo}} \) to determine how far the fin extends along \( \vec{n} \). The procedure is repeated for every line segment \( \ell \) in every fiber polyline \( f \) of the bundle to build the entire fin.

To create an end cap for the fin on the first or last vertex of the fiber, we render two quarter circles each with one halfway point at the 45 degree angle. (This is a level of detail that suffices for our needs, though it would be easy to render more points in between.) We use the same coordinate base of \( \vec{n} \) and \( \vec{t} \) to construct these quarter circles.

Algorithm 4.1 and figure 4.5 illustrate the procedure. Note that \text{makeQuadrilateral} is an abstract function that represents the notion of placing a quadrilateral, a polygon with four vertices and four sides into the scene. It takes four vertex positions in the correct winding order as its arguments. Degenerate normals are handled by adding a small vector \( \vec{\epsilon} \) perpendicular to \( \vec{v}_{\text{eye}} \) to the tangent (and then normalizing the resulting vector again) prior to taking the cross product. We use the operator \( \times \) to signify this.

Algorithm 4.1 Creating a fin

\begin{verbatim}
Require: \( \vec{v}_{\text{eye}} \) is normalized
for all fiber \( f \in \text{bundle} B \) do
    for all line segment \( \ell \in f \) do
        \( P_0 \leftarrow \text{start}(\ell) \)
        \( P_1 \leftarrow \text{end}(\ell) \)

        \{Get the tangent \( \vec{t} \}\}
        \( \vec{t} \leftarrow \text{normalize}(P_1 - P_0) \)
        \{Get normal vector \( \vec{n} \), taking into account degenerate cases.\}
        \( \vec{n} \leftarrow \vec{v}_{\text{eye}} \times \vec{\epsilon} \quad \vec{t} \)

        \{Construct the fins with given width \( w \}\}
        \text{makeQuadrilateral}(P_0, P_1, P_1 + r_{\text{halo}} \vec{n}, P_0 + r_{\text{halo}} \vec{n})
        \text{makeQuadrilateral}(P_0, P_1, P_1 - r_{\text{halo}} \vec{n}, P_0 - r_{\text{halo}} \vec{n})

        \{Deal with the caps on the fin\}
        if \( \ell \) is first segment in \( f \) then
            \text{makeQuadrilateral}(P_0, P_0 + r_{\text{halo}} \vec{n}, P_0 + r_{\text{halo}} \text{normalize}(\vec{n} - \vec{t}), P_0 - r_{\text{halo}} \vec{t})
            \text{makeQuadrilateral}(P_0, P_0 - r_{\text{halo}} \vec{n}, P_0 + r_{\text{halo}} \text{normalize}(\vec{n} - \vec{t}), P_0 - r_{\text{halo}} \vec{t})
        end if
        if \( \ell \) is last segment in \( f \) then
            \text{makeQuadrilateral}(P_1, P_1 + r_{\text{halo}} \vec{n}, P_1 + r_{\text{halo}} \text{normalize}(\vec{n} + \vec{t}), P_1 + r_{\text{halo}} \vec{t})
            \text{makeQuadrilateral}(P_1, P_1 - r_{\text{halo}} \vec{n}, P_1 + r_{\text{halo}} \text{normalize}(\vec{n} + \vec{t}), P_1 + r_{\text{halo}} \vec{t})
        end if
    end for
end for
\end{verbatim}
4.2 GENERATING HINT LINES

This initial approach introduces gaps when the fiber polyline bends, as we can see in figure 4.6a: the individual segments do not match up their end points yet. We can remedy this problem by computing the tangents on the level of the vertices, instead of the line segments.

In this new situation we deal with four points \( P_0 \) to \( P_3 \) for each line segment. \( P_1 \) and \( P_2 \) are the start and end point of the current segment we will render, while \( P_0 \) and \( P_3 \) belong to the adjacent segments. We can compute the tangents \( t_1 \) and \( t_2 \) at \( P_1 \) and \( P_2 \) respectively as follows.

\[
\begin{align*}
\vec{t}_1 &= \text{normalize} \left( \frac{(P_1 - P_0) + (P_2 - P_1)}{2} \right) \\
&= \text{normalize}(P_2 - P_0) \\
\vec{t}_2 &= \text{normalize} \left( \frac{(P_2 - P_1) + (P_3 - P_2)}{2} \right) \\
&= \text{normalize}(P_3 - P_1)
\end{align*}
\]

In short: the tangent of the vertex is the average of the tangents of the line segments the vertex joins. Taking the average of these tangents results in a continuous halo fin, as seen

Figure 4.5: Creating a fin segment.

(a) Regular segment

(b) Capped segment

Figure 4.6: Different results with different tangents

(a) Using the line segment tangents

(b) Using the vertex tangents

In short: the tangent of the vertex is the average of the tangents of the line segments the vertex joins. Taking the average of these tangents results in a continuous halo fin, as seen
in figure 4.6b. Quite conveniently, this way of computing a tangent is also a more stable approximation of the true tangent of the original continuous fiber: the tract of which our polyline is a discrete representation.

The adjusted algorithm 4.2 uses the computed vertex tangents to create smooth fins. It gracefully handles the cases when previous and next line segments are not available, by falling back to the original line segment tangent from algorithm 4.1. The special cases where a previous and next segment are not available, also take care of rendering the end caps.

**Algorithm 4.2 Creating a smooth fin**

```plaintext
for all fiber \( f \in \text{bundle } B \) do
    for all line segment \( \ell \in f \) do
        \( P_0 \leftarrow \text{previous}(\ell) \)
        \( P_1 \leftarrow \text{start}(\ell) \)
        \( P_2 \leftarrow \text{end}(\ell) \)
        \( P_3 \leftarrow \text{next}(\ell) \)

        {If a previous segment exists, use it to compute the vertex tangent}
        if \( P_0 \) exists then
            \( \vec{t}_1 \leftarrow \text{normalize}(P_2 - P_0) \)
            \( \vec{n}_1 \leftarrow \vec{v}_{\text{eye}} \times \epsilon \vec{t}_1 \)
        else
            \( \vec{t}_1 \leftarrow \text{normalize}(P_2 - P_1) \)
            \( \vec{n}_1 \leftarrow \vec{v}_{\text{eye}} \times \epsilon \vec{t}_1 \)

        {No previous segment exists: render a start cap}
        \( \text{makeQuadrilateral}(P_1, P_1 + r_{\text{halo}} \vec{n}_1, P_1 + r_{\text{halo}} \text{normalize}(\vec{n}_1 - \vec{t}_1), P_1 - r_{\text{halo}} \vec{t}_1) \)
        \( \text{makeQuadrilateral}(P_1, P_1 - r_{\text{halo}} \vec{n}_1, P_1 + r_{\text{halo}} \text{normalize}(-\vec{n}_1 - \vec{t}_1), P_1 - r_{\text{halo}} \vec{t}_1) \)
        end if

        {If a next segment exists, use it to compute the vertex tangent}
        if \( P_3 \) exists then
            \( \vec{t}_2 \leftarrow \text{normalize}(P_3 - P_1) \)
            \( \vec{n}_2 \leftarrow \vec{v}_{\text{eye}} \times \epsilon \vec{t}_2 \)
        else
            \( \vec{t}_2 \leftarrow \text{normalize}(P_2 - P_1) \)
            \( \vec{n}_2 \leftarrow \vec{v}_{\text{eye}} \times \epsilon \vec{t}_2 \)

        {No next segment exists: render an end cap}
        \( \text{makeQuadrilateral}(P_2, P_2 + r_{\text{halo}} \vec{n}_2, P_2 + r_{\text{halo}} \text{normalize}(\vec{n}_2 + \vec{t}_2), P_2 + r_{\text{halo}} \vec{t}_2) \)
        \( \text{makeQuadrilateral}(P_2, P_2 - r_{\text{halo}} \vec{n}_2, P_2 + r_{\text{halo}} \text{normalize}(-\vec{n}_2 + \vec{t}_2), P_2 + r_{\text{halo}} \vec{t}_2) \)
        end if

        \( \text{makeQuadrilateral}(P_1, P_2 + r_{\text{halo}} \vec{n}_2, P_1 + r_{\text{halo}} \vec{n}_1) \)
        \( \text{makeQuadrilateral}(P_1, P_2 - r_{\text{halo}} \vec{n}_2, P_1 - r_{\text{halo}} \vec{n}_1) \)
    end for
end for
```
Bleed through

A small remaining problem we will make note of here is the issue of bleed through: small parts of a halo lying just behind another halo may occasionally poke through the surface of the halo that is closer to the viewpoint. In other words; they can bleed through. The reverse can also happen: small parts of fibers that should be visible have small parts drawn over by a fin closer to the viewpoint. We take some inspiration from Busking’s cone splats [6], and add a slope to the fins and caps that constitute our halos.

A negative ramp in depth gradually pulls the fins towards the user and combats the first type of bleed through. A positive ramp in depth gradually makes the fins recede away from the viewer and combats the second type of bleed through. This can be used to tweak resulting images and improve their quality. (The resulting algorithm simply adds a small fraction of $\vec{v}_{\text{eye}}$ to the outlying points of the fins (where $p \pm r_{\text{halo}} \vec{n}$) and will not be shown here for brevity’s sake.)

4.3 Extracting a silhouette and outline from fiber halos

The second step in our rendering technique constructs a silhouette (see figure 4.1) from the intermediate image produced by the halo generation. Figure 4.2b shows that the halos already form a thin, silhouette-like surface when viewed, but it is still too thread-like: the halos are too thin to form a coherent silhouette surface. We also see a few small holes dotting the surface, which we do not want.

While we could increase the width of the halos, this would in turn also reduce the remaining number of hint lines to the point where not enough information remains to accurately represent the three-dimensional form of the bundle. Instead, we need a method to ‘fatten’ the rendered halos to give the silhouette a bit more body and also close any small holes that may remain, without further affecting the remaining hint lines.

Given that we are now working in image space, we can make use of image processing techniques. We use techniques from mathematical morphology [15] to fatten our rendered halos, generating a more present silhouette with holes closed up. Later we add an outline as well.

The remainder of this section describes the morphology operation we will use and shows how we can extend this into an algorithm that builds a silhouette and outline. The later parts will extend this further and show how we can include not only a circumfering outline, but also additional contour lines that help emphasize the three dimensional shape.

4.3.1 The dilation operator

The specific operator from morphology that we are interested in, is called dilation. Dilation is an operator that, put simply, extends the border of an image by the shape of a structuring element. In other words; it fattens an image. Figure 4.7 shows the effect of dilating a square by a disc-shaped structuring element.
The opposite of dilation is *erosion*, which reduces the border. An operator which can be used to close holes; the *closing* operator, is formed by first dilating and then eroding. Of course, since we actually want to keep the fattened image of our rendered halos after closing holes, we only perform the dilation and forego the erosion.

A formal notation that expresses dilation of an object $A$ in an image $E$ by a symmetrical structuring element $B$ can be given using set theory as

$$A \oplus B = \{z \in E | B_z \cap A \neq \emptyset \}$$ (4.5)

Here $E$ represents the $\mathbb{Z}^2$ space of all pixels in a binary image. $A$ is a subset of $E$, $A \subseteq E$, containing only those pixels of $E$ that are ‘on’ or active. $B$ is another subset of $E$ containing the shape of the structuring element. The dilation of an object $A$ in an image $E$ by structuring element $B$ is equal to all pixels $z$ in image $E$, where the translation $B_z$ of structuring element $B$ to center on $z$ shares at least one active pixel with displayed object $A$. (Formally: the intersection of $A$ and $B_z$ is not empty.)

In layman’s terms: the structuring element is swept over the image and if at any point the area covered by the structuring element contains an active pixel, the pixel on which the structuring element is centered is set in the output.

The use of a separate $A$ and $E$ as in the formal notation is bothersome for our purposes. We introduce the notion of a rasterized image $I$, representing both. Rather than placing all active pixels in a separate set $A$, every pixel coordinate $p$ in rasterized image $I$ has a value $I_p$, which is either 0 (inactive, only part of $E$) or 1 (active, part of both $A$ and $E$). In other words:

$$E = I$$
$$A = \{p \in I | I_p = 1 \}$$ (4.7)

The dilated output image is represented by rasterized image $I'$. Analogous to the above formal notations, $B_p$ is the structuring element $B$ translated to $p$. This translates quite
EXTRACTING A SILHOUETTE AND OUTLINE FROM FIBER HALOS

4.3

straight forward into the small segment of pseudo-code presented in algorithm 4.3, on which we will base the remainder of our silhouette extraction process. Image 4.8 illustrates a simple case of the algorithm using a square structuring element.

Algorithm 4.3 A simple dilation operation

\begin{algorithm}
  \textbf{for all} pixel \( p \in \text{image} \ I \) \textbf{do}
  \begin{algorithmic}
    \State \( I_p' \leftarrow 0 \)
    \textbf{for all} pixel \( b \in \text{structuring element} \ B_p \) \textbf{do}
    \begin{algorithmic}
      \If {\( I_b = 1 \)}
      \State \( I_p' \leftarrow 1 \)
      \EndIf
    \EndFor
  \EndFor
\end{algorithmic}
\end{algorithm}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{image}
\caption{Illustrating a simple dilation operation: black is active (1), white is inactive (0), grey is newly made active.}
\end{figure}

4.3.2 Dilating the fiber halos into a silhouette

Now we want to apply the dilation operator to our intermediate image of fiber halos and use it to add more body to the halos and close up any halos. This should result in a processed image where the halos have grown into a coherent silhouette, enveloping and retaining the few hint lines that remained in the intermediate image.

The base dilation operation we describe in algorithm 4.3 operates on a binary image: each pixel has either a 0 or a 1 value. The image of fibers with halos we obtained from the hint line generation described in section 4.2 is not binary. If we want to perform a dilation, we need a method to interpret whether a pixel is 1 (part of a fiber or its halo) or 0 (part of the background).
A straight forward approach that will suffice for now is to look at the color of a pixel $I_p$. If it matches a given background color $c_{\text{back}}$, we assume it is 0: part of the background. Otherwise it is 1: part of a fiber or halo. If an output pixel $I'_p$ would receive the value 1 and input pixel $I_p$ is $c_{\text{back}}$, it is a freshly dilated pixel that should be given the halos’ fill color $c_{\text{fill}}$. Otherwise it should keep whatever color was already there, so that any already drawn hint lines are not overwritten. The result is algorithm 4.4.

**Algorithm 4.4** A dilation operation for fiber halos

\[
\begin{align*}
  c_{\text{back}} & \leftarrow \text{background color} \\
  c_{\text{fill}} & \leftarrow \text{fill (halo) color} \\

  \text{for all pixel } p \in \text{image } I \text{ do} \\
    I'_p & \leftarrow I_p \\
    \text{if } I_p = c_{\text{back}} \text{ then} \\
      \text{for all pixel } b \in \text{structuring element } B_p \text{ do} \\
        \text{if } I_b \neq c_{\text{back}} \text{ then} \\
        \quad I'_p & \leftarrow c_{\text{fill}} \\
        \text{end if} \\
      \text{end for} \\
    \text{end if} \\
\end{align*}
\]

What remains now, is the selection of an appropriate structuring element $B$ to sweep over rasterized image $I$. Rather than the square in our preceding example figure 4.8, we define the structuring element as a disc with a radius $r_{\text{sil}}$ (‘sil’ for ‘silhouette’). This will give a smoothly rounded silhouette, rather than the hard discontinuities a square would give at sharp edges and pointed corners.
4.3.3 Adding an outline to the silhouette

We can now add an outline to the newly generated silhouette in $I'$ by dilating it again to $I''$, using a structuring element with a smaller radius $r_{line}$ and line color $c_{line}$ instead of fill color $c_{fill}$. Note figure 4.9.

![Figure 4.9: Applying dilation to create a bundle’s outlined silhouette](image)

Rather than keep two separate dilations, we will now combine both into one single compound operation. The reason for this might not yet be clear, but it is a prerequisite and a step up to the extension of the outline generation that will follow.

![Figure 4.10: A compound dilation where the lighter color is the silhouette, the darker color is the outline and these cases are separated based on radius](image)

We can combine both dilations into the single compound operation given in algorithm 4.5. As figure 4.10 shows, it uses a disc-shaped structuring element with a radius $r = r_{sil} + r_{line}$. We simply base the color that will be used on a comparison between radius $r_{sil}$ and the measured Euclidean distance $|\hat{b} - \hat{p}|$ (where we interpret both pixel coordinates as 2D vectors). If we find any active pixel $I_b$ with $|\hat{b} - \hat{p}| \leq r_{sil}$, fill color $c_{fill}$ should be used. If we find any active
pixel $I_b$ with $r_{\text{sil}} < |b - p| \leq r$, $c_{\text{line}}$ should be used. Precedence is important here: $c_{\text{fill}}$ takes precedence over $c_{\text{line}}$.

**Algorithm 4.5** A combined silhouette and outline dilation operation for fiber halos

- $c_{\text{back}}$ ← background color
- $c_{\text{fill}}$ ← fill (halo) color
- $c_{\text{line}}$ ← line (hint line & outline) color

for all pixel $p \in$ image $I$ do

$I_p' \leftarrow I_p$

if $I_p = c_{\text{back}}$ then

$c \leftarrow c_{\text{back}}$

for all pixel $b \in$ structuring element $B_{p}$ do

if $I_b \neq c_{\text{back}}$ then

if $|b - p| \leq r_{\text{sil}}$ then

$c \leftarrow c_{\text{fill}}$

else if $c \neq c_{\text{fill}}$ then

{Precedence: only when another pixel $b$ has not set $c_{\text{fill}}$ already}

$c \leftarrow c_{\text{line}}$

end if

end if

end for

$I_p' \leftarrow c$

end if

end for

4.3.4 Self-occluding fiber bundles and contours

The previous approach given in algorithm 4.5 neatly creates an outlined silhouette in one single compound dilation. However, outlines alone may not be enough when dealing with more complexly shaped fiber bundles. The outlines we have created so far only show the circumference of the bundle. This can become bothersome when the bundle suffers from self-occlusion during rendering. This significantly reduces the clearness of the image.

In these cases it is favorable to add some additional contour lines. The additional contour lines convene the three dimensional nature of the fiber bundle much better [10, 22]. Compare figure 4.11a with circumfering outlines to figure 4.11b with contours added.
EXTRACTING A SILHOUETTE AND OUTLINE FROM FIBER HALOS

4.3

(a) Circumfering outline only
(b) Outline with further contours

Figure 4.11: Contours improve images of self-occluding fiber bundles

The remainder of this section describes how we can extend algorithm 4.5 to generate these additional contours. We begin by explaining how we detect self-occlusion and then follow with the required changes to the algorithm.

Detecting occlusion

When a fiber bundle occludes itself, that means one or more fibers pass in front of one or more other fibers. With only the flat 2D image $I$ produced by the first processing step, we have no way of detecting this notion of ‘passing in front’. This requires depth information. Therefore we assume the first processing step creates two outputs: an image $I$ containing colors and an accompanying depth buffer $D$ recording the depth of each pixel.

Because each pixel $p$ only has a single depth, this does not give us a way to detect multiple layers passing over one another. However, it does allow us to detect differences in depth between pixels under structuring element $B_p$. This is in fact analogous to how image-space edge detection can be used to draw outlines and contours [7, 11, 28]. We simply combine the principle with dilation into one single pass.

When a difference between $D_p$ and $D_b$ of a pixel $b \in B_p$ exceeds a given threshold $t$, we assume $b$ belongs to a different fiber halo passing in front or behind (depending on the sign of the difference). We define the operator $\prec t$, where $a \prec t b$ means $D_b - D_a < t$ or ‘$a$ is in front of $b$’. To handle background pixels with this operator as well, we assume that for any background pixel $c$ it holds that $D_c = \infty$. Furthermore, we define that $\infty \prec t \infty$ is false.
Identifying occlusion cases and applying the correct pixel rendering

We need to cover two different cases for a pixel $p$. Either there is a pixel $b \in B_p$ where $b <_t p$ or there is not such a pixel.

At a first glance this seems fairly straightforward again. However, we should realize that many pixels $b \in B_p$ may be closer to the viewer pixel $p$. We need to look for a pixel $b$ which is not only closer to the viewer than $p$, but which is closest to the viewer of all pixels $b \in B_p$. In other words: pixel $b \in B_p$ with the minimum $D_b$.

Previously in algorithm 4.5 we always give precedence to coloring pixel $p$ with the silhouette color $c_{fill}$ when a pixel $b$ is found within radius $r_{sil}$. Depth now plays a role in deciding precedence, meaning this condition can no longer be upheld in that form.

We now explicitly handle the pixels in our structuring element divided in two categories, recording the minimum depth encountered in each category.

1. $Bsil_p$ contains the pixels within $r_{sil}$, that belong to the silhouette. From $Bsil_p$ we obtain the minimum silhouette depth $D_{sil_p}$.

2. $Bcon_p$ contains the pixels outside $r_{sil}$, that belong to contours. From $Bcon_p$ we obtain the minimum contour depth $D_{con_p}$.

The precedence between silhouette and contour can be handled correctly using these minimum depths. Firstly there are the naive cases where only one of either recorded depth is $<_t p$. The more complex case where both are $<_t p$ can now compare both depths with themselves.

Take the scenario displayed in figure 4.12 as an example. We have a pixel $b1 \in Bsil_p <_t p$ and a pixel $b2 \in Bcon_p <_t p$. We know that the local part of $H1$ at $b1$ passes in front of $p$. We know that the local part of $H2$ at $b2$ passes in front of $p$. Because $b2 <_t b1$, we know that $H2$ passes in front of $H1$. Therefore the result of the dilation should have pixel $p$ on a contour line.

Would it have held that $b1 <_t b2$, then $H1$ would have passed in front of $H2$ or $H1$ would have been ‘at the same depth’ as $H2$. In both cases the result should have pixel $p$ in a silhouette. This is clearly the case when $H1$ would pass over $H2$. In the other case, the depth does not have any significance. We can fall back to the old precedence rule from algorithm 4.5. We give precedence to the silhouette over the contour all the same.
Algorithm 4.6 shows the adapted algorithm. It has switched to a depth buffer comparison and implements the new precedence rules we’ve established. The interesting part is the second block of pseudo-code where depth values are compared. The main switching logic checks if we found both a minimum contour and minimum silhouette depth exceeding $t$ and then uses a comparison between both depths to see if $p$ gets to be part of the silhouette or the contours, as explained previously. Only if we did not find any halo closer to the viewer where $p$ should take part in the silhouette or the contour, does $p$ keep its original color and depth values in $I'$ and $D'$ respectively.

With a suitable threshold $t$ algorithm 4.6 produces an image such as in figure 4.11b, which is the final image that our pipeline would output to the screen. Note that with lower thresholds the contours would become more pronounced with more showing up, whereas with higher thresholds less would be shown. With $t = \infty$ we would reobtain the situation depicted in figure 4.11a. The effect of varying $t$ is further shown in figure 4.13.
(a) $t = 2$ has more pronounced contours  
(b) $t = 10$ has less contours

Figure 4.13: Varying threshold $t$
Algorithm 4.6 A combined silhouette and contour dilation operation for fiber halos

\[ c_{back} \leftarrow \text{background color} \]
\[ c_{fill} \leftarrow \text{fill (halo) color} \]
\[ c_{line} \leftarrow \text{line (fiber) color} \]

\textbf{for all} pixel \( \vec{p} \in \text{image} \) \textbf{do}

\{Set the initial minimum depths to infinity\}
\[ D_{sil} \leftarrow \infty \]
\[ D_{con} \leftarrow \infty \]

\{Record both types of minimum depth\}
\textbf{for all} pixel \( b \in \text{structuring element} \) \( B_p \) \textbf{do}

\{Assume that min(\( \infty \), \( \infty \)) simply returns \( \infty \)\}
\textbf{if} \( |\vec{p} - b| \leq r_{sil} \) \textbf{then}
\[ D_{sil} \leftarrow \min(D_{sil}, D_b) \]
\textbf{else}
\[ D_{con} \leftarrow \min(D_{con}, D_b) \]
\textbf{end if}

\textbf{end for}

\{Compare depths and decide which wins\}
\textbf{if} \( D_{sil} < t_p \land D_{con} < t_p \) \textbf{then}
\textbf{if} \( D_{con} < t \) \( D_{sil} \) \textbf{then}
\[ D'_p \leftarrow D_{con} \]
\[ I'_p \leftarrow c_{line} \]
\textbf{else}
\[ D'_p \leftarrow D_{sil} \]
\[ I'_p \leftarrow c_{fill} \]
\textbf{end if}
\textbf{else if} \( D_{sil} < t \) \( p \) \textbf{then}
\[ D'_p \leftarrow D_{sil} \]
\[ I'_p \leftarrow c_{fill} \]
\textbf{else if} \( D_{con} < t \) \( p \) \textbf{then}
\[ D'_p \leftarrow D_{con} \]
\[ I'_p \leftarrow c_{line} \]
\textbf{else}
\[ D'_p \leftarrow D_p \]
\[ I'_p \leftarrow I_p \]
\textbf{end if}

\textbf{end for}
4.4 Alternate hint line representations

We have so far discussed the basic form of our rendering technique that produces an illustrative rendering out of a dense fiber bundle. In this section, we will look at the remaining hint lines that are represented as uniformly colored polylines in the basic form of the technique. We offer ideas for two alternate representations that use lighting and variable line thickness.

4.4.1 Lighting

We have seen examples of artists using a very subtle gradient in a silhouette to suggest lighting on a three dimensional shape. We cannot apply this directly to the silhouette, because it is in actuality not a surface. It is only perceived as such by the viewer. Instead, we can apply some subtle lighting to the lines themselves.

In the regular Phong lighting model, surfaces are lit based on the angle between the light vector and their surface normal according to equation 4.8, where $\mathbf{n}$ represents the surface normal, $\mathbf{l}$ is the light vector pointing from the lit point to the light source, $\mathbf{v}_{\text{eye}}$ is the view vector pointing from the lit point towards the eye and $\mathbf{r}$ is the reflection of $\mathbf{l}$ in $\mathbf{n}$. The $c_i$ are RGB constants representing the ambient, diffuse and specular color values, whereas $\alpha$ represents the shininess of the specular component.

\[
\mathbf{i} = \mathbf{c}_a + \mathbf{c}_d(\mathbf{l} \cdot \mathbf{n}) + \mathbf{c}_s(\mathbf{v}_{\text{eye}} \cdot \mathbf{r})^\alpha \tag{4.8}
\]

When working with a fiber, we do not have surfaces with normals available. Rather we have an infinite amount of normals all lying in the plane perpendicular to the line segment for each point on the fiber. To obtain a correct lighting, we have to pick a vector $\mathbf{n}$ that maximizes $(\mathbf{l} \cdot \mathbf{n})$ and $(\mathbf{v}_{\text{eye}} \cdot \mathbf{r})$ [26]. Rather than performing an explicit calculation of such an optimal vector, we substitute the following for $(\mathbf{l} \cdot \mathbf{n})$ and $(\mathbf{v}_{\text{eye}} \cdot \mathbf{r})$ [2, 26, 30]:

\[
\mathbf{l} \cdot \mathbf{n} = \sqrt{1 - (\mathbf{l} \cdot \mathbf{t})^2} \tag{4.9}
\]
\[
\mathbf{v}_{\text{eye}} \cdot \mathbf{r} = (\mathbf{l} \cdot \mathbf{n}) \sqrt{1 - (\mathbf{v} \cdot \mathbf{t})^2 - (\mathbf{l} \cdot \mathbf{t}) (\mathbf{v}_{\text{eye}} \cdot \mathbf{t})} \tag{4.10}
\]

This enables us to compute a lighting for our fiber polylines based solely on $\mathbf{v}_{\text{eye}}$, $\mathbf{l}$ and $\mathbf{t}$. Of these, $\mathbf{v}_{\text{eye}}$ and $\mathbf{l}$ are given constants, which leaves us with tangent $\mathbf{t}$. In section 4.2.1 we have already shown how to compute the tangents of a fiber polyline, where we constructed halos with them.

Figure 4.14 shows some results a) without lighting, b) with a high diffuse lighting, and c) with lower diffuse lighting with specular lighting added. In both latter cases there is no ambient lighting.
4.4 ALTERNATE HINT LINE REPRESENTATIONS

Figure 4.14: Phong lighting

4.4.2 Line thickness

Artists sometimes vary the thickness of a line stroke based on perceptual significance of the line in question. We can imitate this by thickening polylines where fibers in the bundle change orientation and curvature is high, while reducing thickness of lines running in less active areas of the bundle where curvature is low. This emphasizes the interesting areas and de-emphasizes the more regular areas.

An approach that delivers a quick estimate of curvature is based on the use of an osculating circle. An osculating circle at point $p$ on a curve $C$ is the circle that has the same tangent as $C$ at point $p$ and the same curvature. The amount of curvature at point $p$ can be measured using the circle’s radius: the radius of curvature as depicted in figure 4.15.
Figure 4.15: The osculating circle of a point $p$ on curve $C$ and its radius of curvature $r$

Usually the osculating circle is computed using a limiting procedure, but we can create a geometric approximation that is suitable enough for our needs: we are only interested in a visual aid, which does not need exact numerical accuracy. We favor the additional speed. To find the curvature at point $P_1$ we construct a circle on $P_0$, $P_1$ and $P_2$ by using the simple construction method for the circle circumscribing a triangle. We find the circle’s center $O$ by finding the intersection of both bisectors and then we compute the radius of curvature $r = |P_1 - O|$. The procedure is illustrated in figure 4.16. Note that both the crude curve and the more detailed curve share the same osculating circle approximation and will thus correctly yield the same approximation of the curvature at point $P_1$.

Figure 4.16: Approximated osculating circle of $P_1$ on a crude and more detailed tessellation of a curve

If we take a reasonable upper bound on the values that $r$ can take and clamp everything that rises above it to that bound, we can map this measure of curvature to a given range $[0 \ldots \text{maxThickness}]$ of line thickness, which gives decent enough accuracy for the visual effect. Note that a large $r$ means there is less curvature, and a small $r$ that there is more curvature. Hence the range $[0 \ldots \text{maxThickness}]$ should be inversely mapped on line thickness.
Figure 4.17 shows a comparison between rendering a fiber bundle with and without a curvature dependent stroke width.

(a) Fixed line width

(b) Variable stroke width, where straight lines have a zero width, leading to a sketchy look. (Due to size reduction, the images experiences aliasing. The detail frame shows a close up view that is more representative.)

Figure 4.17: Curvature dependent stroke width
4.5 A GPU-based implementation

Modern consumer level graphics cards offer a lot of computational power. Graphics Processing Units (GPU) are specifically tuned to high speed, parallel processing of graphics in what is commonly referred to as the graphics processing pipeline. This pipeline can be roughly split into three stages we refer to as a) geometry processing, b) rasterization, and c) fragment processing.

The geometry processing stage takes the set of input vertices and applies some operations to them, which results in a set of primitives, such as lines or triangles. Rasterization takes these primitives and splits them into pixel-sized portions called fragments. The final fragment processing stage can again apply operations to these fragments. One of the more common operations in this final stage is called texturing, where the fragments are assigned elements of a texture that was previously bound or activated. The processed fragments are then finally rendered to the output, also referred to as the render target. The render target does not necessarily have to be linked to the view screen’s frame buffer. Modern OpenGL often supports a so-called frame buffer object (FBO), which allows the programmer to use a 2D bitmap texture as a render target.

On older cards the pipeline consisted of fixed functionality the programmer could exert little control over. This is now commonly referred to as the fixed function pipeline. Modern cards are more sophisticated and offer the programmer the capability to compile and run code directly on a GPU in the form of a shader program, composed of multiple components called shaders. Vertex shaders and the relatively new geometry shaders are executed in the context of the first geometry processing phase and can manipulate the data fed into the rasterization phase. Fragment shaders are executed in the context of the fragment processing stage and can change the values written to pixels in the output.

Because all the data pushed through the pipeline is local, this processing can be parallelized almost effortlessly. It is no wonder then, that modern graphics cards carry upwards of 256 GPUs, all ready to service. With a unified shader architecture each GPU can run in all required roles and a controller on the card handles load balancing transparently. This makes the GPU exceptionally qualified for algorithms of a parallel nature, provided they share as little state as possible.

For further reading on the history and structure of GPUs, we refer to van Pelt and vanden Boer [32, 34].

4.5.1 Global structure of the pipeline

Real-time operation of many modern rendering techniques relies on the specialized horse power offered by the graphics card and ours is no different. An efficient GPU-based implementation of the illustrative rendering technique described in the previous sections of this chapter was constructed using OpenGL, to prove our technique can operate in real-time.

Figure 4.18 shows a general overview of the algorithm’s implementation and how it fits into the GPU’s graphics processing pipeline.

Halo rendering is handled by a geometry shader, which creates the halos on the fly. The rendering output of this phase is redirected to a frame buffer object, which records both the color channel and depth channel into local GPU memory as textures. Both textures are subsequently offered to a fragment shader, which performs the dilation operation and outputs the result onto a screen-aligned quadrilateral.
A GPU-BASED IMPLEMENTATION

4.5 Rendering halos with a geometry shader

Because the orientation of our halos is view-dependent, they need to be rebuilt when the orientation of the viewpoint with respect to the fiber bundle changes. A one-time generation of the fins as static geometry would therefore be useless. Instead our implementation generates the fins on the fly, using geometry shaders: unlike vertex shaders, which can only manipulate existing geometry’s vertices, geometry shaders can create new geometry on the fly as data is processed by the graphics pipeline. This is ideal for turning a fiber’s polyline into a polygonal halo floating ‘around’ said polyline.

Besides plain vertices, geometry shaders accept other kinds of primitive for rendering. Lines or polygons, for instance. More importantly for us: geometry shaders accept input primitives which consist of a primitive including its adjacencies. Recall algorithm 4.2, which constructs halos with averaged, smoothed tangents. This algorithm requires knowledge of the previous and next line segment along with the current line segment. It requires a segment’s adjacent segments: its adjacencies. We use a ‘lines with adjacencies’ primitive here, such as OpenGL’s GL_LINES_ADJACENCY primitive.

Figure 4.19 shows a stream of successive vertices representing a polyline being sent to the GPU and being interpreted as lines with adjacencies by the geometry shader. Each pair of successive vertices is augmented with the previous and next vertex and processed as a quartet by the geometry shader. Two successive cases are displayed in figure 4.19. The window steps ahead one vertex at a time, sliding through the vertex buffer’s contents. The geometry

\[\text{color texture I}\]

\[\begin{array}{l}
\text{• line color } c_{\text{line}} \\
\text{• fill color } c_{\text{fill}} \\
\text{• halo radius } r_{\text{halo}} \\
\text{• ...}
\end{array}\]

\[\begin{array}{l}
\text{• line color } c_{\text{line}} \\
\text{• fill color } c_{\text{fill}} \\
\text{• silhouette radius } r_{\text{sil}} \\
\text{• contour radius } r_{\text{cont}} \\
\text{• threshold } t \\
\text{• ...}
\end{array}\]

\[\text{depth texture D}\]

\[\text{full screen quadrilateral}\]

\[\text{screen}\]

Figure 4.18: GPU implementation pipeline

The remainder of this section will walk us through this pipeline, detailing the procedure of fitting our rendering technique’s algorithm to the GPU where necessary.
shader can treat each case as three line segments in the form of the quartet of vertices it receives: \((P_0, P_1)\) forms the first adjacency. \((P_1, P_2)\) forms the segment itself and \((P_2, P_3)\) forms the second adjacency.

![Diagram](image)

Figure 4.19: Processing adjacent polyline segments with a geometry shader

Support for adjacencies means that a geometry shader can implement the bulk of algorithm 4.2 directly, a few idiosyncrasies in the shader language’s syntax withheld. We do need to take into account the rounded end caps on the halo though.

### Halo end caps

In algorithm 4.2 we rendered end caps on a halo by introducing a special case for the first and last segments of a fiber. We mentioned before that all the data being processed by the GPU has to be local to execute in parallel. The graphics card has no notion of ‘first item’ and ‘last item’. These are concepts of a globally visible list of data.

We introduce a local notion of ‘first item’ and ‘last item’ for the end caps in our geometry shader, by sending the first and last vertex of each fiber twice in a row. Recall that our geometry shader operates on line segments with their adjacencies: a quartet of vertices \((P_0, P_1, P_2, P_3)\).

A line segment \((P_1, P_2)\) is then the first segment of a fiber if and only if \(P_0 = P_1\). Synonymously, it is the last segment of a fiber if and only if \(P_2 = P_3\). This is the easiest way to mark the first and last vertex in a parallel batch without too much additional overhead. There are alternatives, but these require more bus bandwidth or involve multiple render passes.

Note that the remainder of the algorithm’s end-cap rendering can be implemented fairly straight forward again.

### Two types of primitives: two passes

While, as previously noted, a geometry shader allows for many different type of primitives to be used as input and output, it only allows a single input type and a single output type at a time.
We can use a geometry shader to render the halos belonging to a bundle of fibers. Thanks to the previous little trick, this includes the end caps in the same rendering pass. However, we cannot immediately render the fibers themselves in that same pass. This is because the halos require the output type to be set as a triangle strip to produce polygons, whereas the fibers themselves are simple polylines.

Therefore we split the halo rendering process into two passes: the first pass renders the halos for all the fibers, while the second renders the polylines representing the fibers themselves.

**The first pass** enables our geometry shader and renders our input stream of polylines. The geometry shader will transform the polyline geometry into the halo geometry on the fly. When sending our geometry to the graphics card, we have to take care not to override hardware imposed limits on the number of vertices the shader can produce, which requires us to chop our input into separate batches sent to the card one by one.

**The second pass** disables the geometry shader and renders our input stream directly into the scene as regular polylines. We use a ‘less than or equal’ test on the scene depth buffer. This dismisses those parts of the polylines that are positioned behind any halo geometry already rendered in the first pass, rendering only the remaining parts that should be visible.

### 4.5.3 Dilating a silhouette with a fragment shader

In algorithm 4.6 we defined a silhouette and contour generating operation based on a simple dilation operator. While it grew substantially more complex than a simple dilation operator, one thing it has in common is spacial locality. It has a spatial locality constrained to $B_p$, that is: to the structuring element $B$ translated to center over pixel $p$. It is therefore a good fit for execution on the GPU in the form of a fragment shader, where the outer loop of the algorithm (for all pixel $p \in \text{image } I$) is executed in parallel.

The algorithm needs both the color image and the depth buffer to work on. These are, if you will recall figure 4.18, supplied as textures that are retrieved from a frame buffer object the halo rendering step wrote its output to. By processing dilation on the GPU as well, we also prevent any additional costs associated with downloading these textures from the GPU’s memory back to the CPU’s main memory.

**Interpreting the depth texture**

While algorithm 4.6 can again be implemented fairly straightforward, there are two issues relating to the the depth buffer / depth texture $D$ we need to be aware of and handle correctly.

The first issue is the effective range of the depth buffer. Algorithm 4.6 blissfully assumes the depth buffer to range from 0 (at the viewer’s viewpoint) to infinity, where pixels where nothing is rendered have infinite depth. In practice, a depth buffer has limited accuracy: for most graphics cards a 16-bit or 32-bit integer is the norm, though some form of floating point is also supported on many cards.

A standardized view on the depth buffer is to use the range of $[0 \ldots 1]$. The value 0 is still closest to the viewer and defined as the *near plane* $d_{\text{near}}$. The value 1 is furthest
from the viewer and defined as the far plane \( d_{\text{far}} \). The physical depths in the world scene represented by these planes are configurable. This allows us to adjust the effective range our depth buffer needs to record to the objects in the scene, thus making better use of what accuracy is available in the buffer. (See figure 4.20)

\[
\begin{align*}
\text{near plane} & \quad \text{far plane} \\
\text{margin} & \quad \text{margin} \\
\end{align*}
\]

\[
\begin{align*}
d_{\text{near}} &= 0 & d_{\text{far}} &= 1 \\
\end{align*}
\]

Figure 4.20: Setting a near and far plane

Instead of ‘empty’ pixels being placed at infinite depth, our implementation puts them on the far plane, which we set to be slightly further away than any content which should be visibly rendered.

The second issue is the fact that the range of depth values being recorded in depth buffer \( D \) may not map linearly to the actual values stored. This depends on whether orthographic or perspective projection is being used. What OpenGL actually stores in the depth buffer is the result of \( z' \), where \( z' \) and \( w' \) come from the projection of homogeneous coordinates \( (x, y, z, w) \).

**When using orthographic projection** \( w' \) will equal 1 and the resulting value written to the depth buffer is just a \( z' \) that is linear in the range \([0, 1]\).

**When using perspective projection** \( z' \) is non-linear in the range \([0, 1]\), typically recording more fine grained differences in depth at the front than it does at the back. This coincides with general use for rendering real time graphics: geometry closer to the viewer needs to be rendered with finer detail and geometry further away sacrifices detail to accomplish this.

As actual order is still retained and only scale is affected, values that are non-linear are not an issue for many algorithms which do nothing more than comparing depths and deciding which is closer: the original use for the depth buffer. Recall however, that algorithm 4.6 at one point requires measuring if a difference between two depth values exceeds a certain threshold \( t \). There we require a proper comparison of depth values in a linearly mapped range.

In both cases it is possible to map the depth value \( d_p \) of a pixel back to its proper \( z \)-coordinate in camera space \( p_z \), which allows us to properly compare it to a given threshold \( t \).
In case of orthographic projection, we simply treat the depth value \( d_p \) as a linear interpolation factor between the near and far plane \( d_{near} \) and \( d_{far} \). In case of perspective projection, we have to undo the effect of OpenGL’s perspective projection matrix on the \( z \) coordinate, by applying the inverse of the matrix \([29]\).

\[
p_z = \begin{cases} 
  d_{near} + d_p(d_{far} - d_{near}), & \text{for orthographic projection} \\
  \frac{(d_{near} d_{far})}{d_{far} - d_p(d_{far} - d_{near})}, & \text{for perspective projection}
\end{cases}
\]

(4.11)

### Setting up a canvas for the fragment shader

As mentioned previously, the fragment shader receives two textures as its input. One contains the color image \( I \) and the other the depth image \( D \) of the haloed fibers. What remains is how to handle drawing the output of the fragment shader (the image where the silhouette and contours have been generated) to the screen.

To have the fragment shader map its results to the screen correctly, we use a simple screen aligned polygon as a canvas. We can fill in the method by which to obtain such a polygon in a few different ways.

1. We can (orthographically) project the bundle’s bounding box onto a screen aligned plane and use that projection as the canvas for the fragment shader.

2. We can compute a screen aligned bounding box for the bundle and use the front face as the canvas for the fragment shader.

3. We can compute a screen aligned bounding box for the bundle and use it as a clipping mask inside the fragment shader, having the shader discard any fragments that fall outside the mask. We use a screen aligned quad spanning the entire view port as the canvas for the fragment shader.

While both intuition and reasoning would say these are ordered most effective to least effective, the opposite seems to actually be true when the methods are put to the test in practice. The last method has to process far more fragments than the first. The last method has to execute more instances of the fragment shader than the second (even if many can be canceled out by early dismissal with the mask). But still the last method operates significantly faster. Hypothetically, this issue could originate from the graphics card utilizing an optimized special case when working with screen aligned quads sized to the entire view port.

### 4.6 Processing multiple fiber bundles

While the algorithms we have presented in the previous sections of this chapter only operate on a single bundle, our illustrative rendering method can easily be extended to work on multiple bundles. In fact: no extension is really necessary. The method will work for multiple bundles ‘out of the box’ and plays nice with any other objects being composed into the same scene.
AN ILLUSTRATIVE RENDERING METHOD FOR FIBER BUNDLES

Figure 4.21: Composing multiple bundles with our illustrative method

We already accomplished this when designing our algorithm back in section 4.3.4 by taking special care of what depth values ended up being written to the output scene in algorithm 4.6.

For each pixel the fragment shader implementing the silhouette generation process writes new values to the pixel’s depth value before it is written to the scene. These values override the regular depth values of the screen aligned quad the shader uses as a canvas. (See section 4.5.3) In effect, the surface depth of the screen aligned quad is ‘warped’ on a fragment-to-fragment basis and forms a kind of height field — or rather, depth field.

This warping process results in the depth buffer of the scene into which a bundle is composited, to be updated to take on depth values reflecting this dilated bundle. In other words: the proper functioning of the depth buffer is kept invariant.

This means no customized logic is needed to compose multiple objects into the scene. We do not have to alter the depth testing function. We do not have to disable depth checking. We do not need anything fancy. The underlying OpenGL renderer can simply use a default, hardware accelerated z-culling rendering method to compose the scene.
Chapter 5

A focus & context technique for illustratively rendered fiber bundles

When multiple illustratively rendered fiber bundles have been loaded into a scene as we described in section 4.6, a user should be able to focus his or her interest on a single fiber bundle and not be hindered by other bundles in the vicinity occluding the target of the user’s interest. We want to pull apart closely grouped fiber bundles into separate entities, which should allow us to observe bundles buried deeper inside the composition. Figure 5.1 illustrates the problem.

Figure 5.1: Example of the bundle representing the fornix (purple) being occluded and resolving the occlusion

To this end we want to accompany the illustrative rendering method from the previous
chapter with a focus & context technique that solves this occlusion problem.

The principle of focus & context is to clearly and cleanly present and suitably emphasize the object of interest in a visualization: the focus, while providing a more subtle representation of the remainder of the visualization as a context, serving to place the focus into perspective as part of the whole.

In medical illustrations, common methods to uncover hidden features the user wants to observe are the introduction of cuts or dissections in an image, the peeling of an outer layer, or ghosting or some kind a ‘magic lens’ which allows us to see through otherwise opaque tissue. These are often encountered in guides of surgical procedures.

In addition to these, exploded views [5] are a popular means to provide focus & context. An exploded view pulls a composed object apart into its separate components, i.e. the object explodes into its component parts. Two often encountered types of explosion are a generic spherical explosion, with all component parts spreading equally, or one or more linear explosions along a set number of vectors. In many technical illustrations, such vectors coincide with axels, shafts, etc. Another type of explosion folds the subject of interest open like a book.

These kind of exploded views in fact date all the way back all the way to early Renaissance sketches. Figure 5.2 shows two of such sketches, along with a modern example of exploded views in a medical visualization.

![Figure 5.2: Examples of exploded views](image)

(a) An early medical sketch by Da Vinci, using exploded views
(b) An early mechanical sketch by Da Vinci, using an explosion along a drive shaft
(c) A medical visualization of a turtle with an exploded view of the turtle’s shell, taken from [5]

**5.1 General overview**

The focus & context technique we have developed relies on the philosophy of exploded views, but borrows from the techniques of cuts and dissections for uncovering hidden features. It starts with a simple spherical explosion originating from the focused fiber bundle.

Note that it is important that the spherical explosion keeps the topology between the bundles roughly intact. The spherical explosion increases mutual distance between the context bundles and distance between context bundles and focused bundle, but does so uniformly. This preserves the user’s mental map. A mental map is structured cognitive information a user builds up in his memory and helps with reasoning, comprehending and navigating the higher level structure of a visualization [27]. Mental maps are generally associated with the
overview of the structure of graphs, but the concept is the same for other types of structured data.

This explosion decomposes tightly grouped fiber bundles into separately floating entities. The result is a cloud of separate bundles, centered around the focused bundle. As explained previously in this chapter, we want to use this explosion to enable the user to observe bundles buried deeper inside the composition. However, even after the explosion, several bundles may occlude the focused bundle, hiding the feature the user is interested in.

We could use spatial distortion and retract the offending fiber bundles, or wedge them apart [8]. This would closely mimic techniques that can be seen in surgical illustrations [9]. The downside of such a technique is that we end up warping (part of) the shape of the context. This is all well and fine if we have other visual cues left that can identify parts of an image, but a fiber bundle is a fairly abstract concept that is only recognizable by its general shape and relative position to other bundles of which an expert recognizes the general shape.

Therefore, we want a technique that keeps this shape intact. Instead of using volume manipulation techniques such as retraction or wedges, which introduce alienating distortions to the data, we simply slide the offending bundles out of the way in a plane parallel to the screen. Think of this as if we were looking at a collage of paper cutouts being laid out on a table and we slid a few cutouts out of the way to look at those cutouts they were covering.

The above is summarized in figure 5.3: first we explode the bundles relative to the focused bundle, then we slide bundles that continue to occlude the focused bundle out of the way.

This does not yet tell us how powerful an explosion should be, or how far bundles should be slid. Therefore we look at the problem from a different angle. Instead of causing an explosion, we resolve world-space intersections of the focused bundle with the context bundles. Instead of sliding occluding bundles out of the way, we resolve screen-space intersections of the focused bundle with the context bundles.

When we state that we ‘resolve an intersection’, it means the following. We find out if a context bundle intersects the focus bundle. If it intersects, we find out how much penetration there is. Then we translate the context bundle by this amount of penetration until it no longer intersects the focus bundle.

We can measure intersection and penetration in various ways: we can use simple bounding spheres, axis aligned bounding boxes, non-axis aligned bounding boxes, etc. Though it is possible to measure penetration on the objects themselves, we limit ourselves to bounding
volumes to retain interactive frame rates. We base our algorithm on oriented bounding boxes.

We will start by explaining our algorithm in section 5.2, using bounding spheres for simplicity. Then we illustrate the required changes to adapt it to working with oriented bounding boxes in section 5.3.

5.2 Resolving intersection of fiber bundles

Assume we have a focus bundle and a possibly intersecting context bundle, which we define by their bounding spheres $\mathcal{BS}_F$ and $\mathcal{BS}_C$ as depicted in figure 5.4a.

5.2.1 Resolving intersection in world space

To stop the context from partially overlapping or occluding the focus, the first step is to make sure the focus and context do not physically intersect in world space, which requires repelling the context away from the focus if both intersect.

The minimum distance required between the center points $c_F$ and $c_C$ of the spheres to have the spheres not intersect, simply coincides with the sum of their radii: $r_C + r_F$.

Therefore the penetration depth in world space $p_{ws}$ can be found by

$$p_{ws} = \max((r_C + r_F) - |\vec{d}|, 0),$$

where

$$\vec{d} = C_C - C_F$$

We can then translate $\mathcal{BS}_C$ by $p_{ws}$ along $\vec{d}$, which results in the situation depicted in figure 5.4b, where the bounding spheres $\mathcal{BS}_F$ and $\mathcal{BS}_C$ no longer intersect.

![Diagram](a) Initial situation has intersection (b) ‘Explode’ context away from focus to resolve intersection

Figure 5.4: Preventing intersection in world space by translating the context
5.2.2 Resolving intersection in view space

We have solved the intersection in world space, but the projections of $BS_F$ and $BS_C$ along $\vec{v}_{eye}$ onto the screen may still intersect. This is the case in figure 5.5a.

We repeat the process from our previous step to solve this issue, but adapt it to the screen aligned plane in which $c_F$ lies. This means that rather than working with vector $\vec{d}$, we work with its orthogonal projection onto this screen aligned plane, represented by $\vec{d}'$.

As before, the minimum distance required is $r_C + r_F$: the orthogonal projection of a sphere onto a plane is a circle with the same radius as the sphere after all. We can then obtain the view space penetration $p_{vs}$ in this screen aligned plane in similar fashion to $p_{ws}$. The situation changes slightly though, because we only want to manipulate view space intersections with contexts in front of the focus, not those behind. A simple case distinction based on a dot product solves this.

$$p_{vs} = \begin{cases} \max((r_C + r_F) - |\vec{d}'|, 0), & \text{if } \vec{v}_{eye} \cdot \vec{d} \leq 0 \\ 0, & \text{otherwise} \end{cases} \quad (5.3)$$

We can then translate $BS_C$ further by $p_{vs}$ along $\vec{d}'$. This finally results in the situation depicted in figure 5.5b, where the bounding spheres $BS_F$ and $BS_C$ no longer intersect.

![Figure 5.5: Preventing further intersection in view space by translating the context once more](image)

5.3 Adapting the method to oriented bounding boxes

We now make the transition from bounding spheres $BS_F$ and $BS_C$ to oriented bounding boxes $OBB_F$ and $OBB_C$. The simple algorithm previously described in section 5.2 has only a few
requirements to work on bounding objects other than spheres:

1. We need the 'centers of mass' \( C_F \) and \( C_C \).

2. We must be able to project the representation of the focus and context onto a plane.

3. We must be able to compute penetrations \( p_{ws} \) and \( p_{vs} \) on said representation and its projection.

Given a bounding box the first is easy to obtain by taking the center of the box. This can be done by computing the average of all corner vertices, taking the halfway point of the diagonal, etc.

The projection of a convex shape, such as a bounding box, is again convex. Therefore the projection of a bounding box onto a plane is equal to projecting its corner points onto said plane and computing the 2D convex hull of the projected points. Convex hulls \( \mathcal{CH}_F \) and \( \mathcal{CH}_C \) are easy to compute using Graham’s scan [18], or similar well known methods for computing convex hulls on a set of points. With this we meet the second prerequisite.

The third prerequisite can be met through an intersection test based on the method of separating axes [16, 17].

### 5.3.1 Measuring penetration through a separating axes test

A separating axes test is a well-known method of performing intersection tests for convex polyhedra. It relies on running a series of interval tests on projections of the polyhedra onto a set of potential separating axes. A separating axis is an axis on which the projected intervals are disjoint, proving the polyhedra do not intersect. In other words: the existence of such an axis separates the polyhedra.

For oriented bounding boxes \( \text{OBB}_F \) and \( \text{OBB}_C \), the set of potential separating axes \( A \) consists of the set of normals of the faces of the boxes combined with the set of vectors obtained by taking a cross product between each possible pairing of edge from \( \text{OBB}_F \) with edge from \( \text{OBB}_C \).

For 2D convex hulls \( \mathcal{CH}_F \) and \( \mathcal{CH}_C \), the set of potential separating axes \( A \) is less complex. It consists only of the outward facing normals of the edges of the hulls.

Performing the axis tests

Now for each axis \( \vec{a} \in A \) we perform a scalar projection of the vertices of \( \text{OBB}_F \) and \( \text{OBB}_C \) onto \( \vec{a} \), obtaining two intervals \( I_F \) and \( I_C \) representing the projections of the bounding boxes. These are defined as:

\[
I_F = \left[ \min\{\vec{a} \cdot \vec{v} : \vec{v} \in \text{BB}_F \}, \max\{\vec{a} \cdot \vec{v} : \vec{v} \in \text{BB}_F \} \right] \\
I_C = \left[ \min\{\vec{a} \cdot \vec{v} : \vec{v} \in \text{BB}_C \}, \max\{\vec{a} \cdot \vec{v} : \vec{v} \in \text{BB}_C \} \right]
\]
5.3 ADAPTING THE METHOD TO ORIENTED BOUNDING BOXES

When the intervals are disjoint, we have found a separating axis. This means $BB_F$ and $BB_C$ do not intersect, meaning penetration $p$ is zero.

When the intervals overlap, we measure the amount of overlap as the penetration $p_a$ along axis $\vec{a}$. For this, we need to know in which direction we need to measure: either we measure from the start of $I_C$ to the end of $I_F$ or we measure in reverse. This is necessary to obtain the value corresponding to the direction of $\vec{d}$.

To this end we perform a scalar projection of $C_F$ and $C_C$ onto $\vec{a}$, which gives us $C'_F$ and $C'_C$. We then check the order in which these occur on axis $\vec{a}$ and decide which direction of penetration we should report as $p_a$. Note that by construction, $p_a$ is prevented from receiving a negative value.

\[
p_a = \begin{cases} 
\max\{x : x \in I_F\} - \min\{x : x \in I_C\}, & \text{if } (C'_C - C'_F) > 0 \\
\max\{x : x \in I_C\} - \min\{x : x \in I_F\}, & \text{otherwise}
\end{cases}
\]  

(5.7)
Figure 5.7: Finding the level of penetration on an axis

After having processed all axes $\vec{a} \in A$ without finding one with disjoint intervals, we obtain the final penetration depth by taking the minimum of all $p_a$ (which, as we have stated, is never negative). As each $p_a$ represents how far to move along said axis $\vec{a}$ to undo any intersection, we can take the ‘most conservative’ option. This is the axis $\vec{a}$ with the smallest penetration depth $p_a$.

**For oriented bounding boxes** we unproject this penetration $p_a$ from its axis $\vec{a}$ back onto $\vec{d}$ to obtain $p_{ws}$.

**For 2D convex hulls** we unproject this penetration $p_a$ from its axis $\vec{a}$ back onto $\vec{d}_{pr}$ to obtain $p_{vs}$.

Note that in the degenerate case where $\vec{a}$ and $\vec{d}$ or $\vec{d}_{pr}$ are perpendicular, we can simply discard the chosen $p_a$ and $\vec{a} \in A$ and select the next smallest $p_a$. We are guaranteed to have at least one $\vec{a}$ not perpendicular to $\vec{d}$ because at least all sides of a (non-degenerate) bounding box or convex hull will be represented in $A$.

### 5.4 Handling multiple context bundles simultaneously

If we want to apply this algorithm for $N$ context bundles $C_N$, then we proceed as follows:

1. For all context bundles, compute their world-space penetration.

2. Use the maximum of these computed world-space penetrations as the measure by which *all* context bundles $C_N$ are translated in world-space.

3. For all context bundles, compute and apply their view space translation, if it applies.
We explicitly translate all the bundles in world space by the maximum value of $p_{ws}$ that was found. This assures us that we will not be pushing context bundles into one another, but will move them uniformly outward without disrupting relative position too much as per figure 5.8.

![Diagram](image)

(a) Penetration of $F$ by $C_1$
(b) Translating $C_1$ and $C_2$ by their individual penetration leads to collision
(c) Translating $C_1$ and $C_2$ by the maximum penetration preserves the structure of the context

Figure 5.8: Handling multiple world space translations

The same constraint does not hold for the screen-aligned slide we apply afterwards. Since this second translation interferes with the relative position of the context bundles on a much greater scale, we want to keep its effect to a minimum. That means each individual context bundle is only to be translated as far as strictly necessary for that bundle to clear any occlusion with the focused bundle, and no more.

Some users might find the default level of displacement still too invasive for particular data sets they work with. Therefore, we give the user a finer control over the displacements by employing a modifier which scales the penetration values before they are used for displacement. This gives users a way to fine tune the level of displacement to taste.

### 5.5 Animating transitions of context bundles

A problem with the focus and context technique as we have presented it so far, is that it does not preserve a user’s mental map very well. When we suddenly shift around the structure of the data, such as is the case with focus & context techniques based on instant displacement, this adversely affects a user’s constructed mental map and comprehension of the visualization [27]. What we want to do is visualize unexpected changes in the visualization’s layout and thus allow the mental map of a user to be updated accordingly [3]. In short: we want to animate any important transitions not directly caused by user interaction.

Important transitions in our focus & context technique result from a) the user enabling and disabling the focus & context technique, b) the user picking a new focus bundle, and c) context displacements taking place in the second ‘sliding’ step.
The first two types of transition speak mostly for themselves. You want a user to be able to discern where bundles came from before the focus & context technique displaced them and you want a user to be able to re-balance the mental map around the new focus bundle without having the relearn the entire layout of the data.

We will pay attention to the last type in particular, which may seem odd at first. However, it is an important type of transition to animate due to a nasty issue with a ‘dead point’ located exactly in the path of the eye vector $\vec{v}_{\text{eye}}$ passing through the center of the screen. The direction an occluding context bundle is slid flips by 180 degrees in that point. This is particularly noticeable and disrupting when rotating the viewpoint in such a way that a context’s center point $C_C$ passes through this dead point. Chances of this happening increase with more context bundles present as well. Without animation to ease the transition, such a bundle would instantaneously ‘snap’ to the other side of the visualization’s view port. With animation present, the context bundle gently floats to the other side as is depicted in figure 5.9.

5.5.1 Using animation tracks

Our focus and context technique is stateless, meaning that every time the viewpoint changes, computation of the focus & context layout starts from the neutral situation where the algorithm has not yet been applied.

This makes it easy to introduce animation tracks. Instead of immediately translating the bundles in the rendered scene, we use our displacement algorithms to compute a set of end positions in memory. These end positions are treated as movement targets for each fiber bundle. An animation track for a transition can then simply consist of moving a fiber bundle’s scene representation from its current position to the (current) target position stored in memory.

This can be done with a fixed velocity or with a more advanced model with acceleration and friction incorporated for a smoother transition with ease-in and/or ease-out properties. In our experience, a simple constant velocity already works quite well at smoothing out all transitions, including where a context bundle affected by the second screen-aligned slide suddenly snaps to the opposite side.
5.5 ANIMATING TRANSITIONS OF CONTEXT BUNDLES

Figure 5.9: Demonstrating animation to ease movement of context bundles
Chapter 6

Results

In the previous chapters, we have presented an illustrative rendering technique for fiber bundles and a related focus & context system. In section 4.5 we have shown a GPU-based implementation of the rendering technique.

In this chapter we give an overview of this implementation’s performance followed by the functional results of both the rendering technique and the focus & context system, as gauged against the objectives we presented in section 1.3. In closing are example images produced by our rendering technique and focus & context system.

6.1 Performance results

Our illustrative rendering technique consists of two steps. First we cull unnecessary fibers with our halo rendering technique. This step has a complexity of $O(v)$, where $v$ is the total number of vertices in the fiber polylines of the input data. Secondly, we apply an image space algorithm to the rendered result. This algorithm analyzes and compares information present in the rendered image’s depth buffer and appropriately applies dilation to grow a silhouette with contours. This step has a complexity of $O(bm)$, where $m$ is the total number of pixels in the view port and $b$ is the number of bundles.

For a well balanced picture of our technique’s performance we have recorded performance data where we vary a) the screen resolution, b) the combined size of the bundles, and c) the number of bundles, which reflects the dependency on $m$, $v$ and $b$ respectively. Furthermore, we have varied the thickness of the silhouette and contour lines, which affects the radius of our image space algorithm’s structuring element. This change significantly affects the results, as it has a large impact on the number of required texture samplings. Lastly, we have recorded two variations: one where our approximated curvature analysis (used to render artistic strokes with varying width) is enabled and one where it is disabled.

Measuring performance is a non-trivial task. How long a general task truly takes to execute is practically impossible to compute on operating systems where a preemptive scheduler is in use, such as is the case for most modern systems. The OS scheduler can skew results by suspending our running process to give CPU time to tasks with higher priority.
6.1 PERFORMANCE RESULTS

We could use a high performance timer that measures exact CPU cycles. However, as
our implementation does most of its work on the GPU rather than the CPU, this would
give inaccurate results as well: the CPU would mostly be idle. Instead we have settled
on measuring execution times using the CPU clock. Typically the clock only has millisecond
accuracy. To combat the loss of accuracy and anomalies due to interference of the OS scheduler
we have run the same test case multiple times in sequence.

Each test case has been set up to run 1000 iterations of the rendering technique and to
report back the average time taken and derived frame rate. (Note that the frame rate is
derived from the time taken before rounded down for display in our results.) As a reference,
we have included a case where we render an empty scene. All measurements were taken using
a NVidia Geforce 9800 GX2 graphics card.

We note here that our implementation has an optimization built in that aims to reduce
the number of processed pixels \( m \) to a particular subset of pixels on which the image space
algorithm could have an actual effect. This optimization makes the algorithm dependent on
the shape, orientation and scale of the displayed fiber bundles. Therefore, figure 6.1 shows
the bundles used in our test cases in their exact orientation and scale, relative to a view port
of 1100 \times 1100 pixels.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{61.png}
\caption{Test cases for performance measurements}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Resolution (px) & Silhouette (px) & Curvature & Time taken (ms) & Frame rate (fps) \\
\hline
0.35 & 2938.38 & & & \\
\hline
\end{tabular}
\caption{Performance for the empty scene}
\end{table}
Table 6.2: Performance for a single bundle of 1000 fibers

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Silhouette (px)</th>
<th>Curvature</th>
<th>Time taken (ms)</th>
<th>Frame rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>no</td>
<td>34.68</td>
<td>28.84</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>yes</td>
<td>62.26</td>
<td>16.06</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>no</td>
<td>17.05</td>
<td>58.64</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>yes</td>
<td>22.49</td>
<td>44.46</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>no</td>
<td>18.60</td>
<td>53.76</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>yes</td>
<td>24.14</td>
<td>41.43</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>no</td>
<td>17.76</td>
<td>56.30</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>yes</td>
<td>23.12</td>
<td>43.26</td>
</tr>
</tbody>
</table>

Table 6.3: Performance for single bundle of 4000 fibers

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Silhouette (px)</th>
<th>Curvature</th>
<th>Time taken (ms)</th>
<th>Frame rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>no</td>
<td>62.26</td>
<td>16.06</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>yes</td>
<td>83.39</td>
<td>12.14</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>no</td>
<td>46.72</td>
<td>21.40</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>yes</td>
<td>65.79</td>
<td>15.20</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>no</td>
<td>49.87</td>
<td>20.05</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>yes</td>
<td>68.78</td>
<td>14.54</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>no</td>
<td>44.47</td>
<td>22.49</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>yes</td>
<td>63.43</td>
<td>15.76</td>
</tr>
</tbody>
</table>

Table 6.4: Performance for 10 bundles of 3000 fibers total

<table>
<thead>
<tr>
<th>Resolution (px)</th>
<th>Silhouette (px)</th>
<th>Curvature</th>
<th>Time taken (ms)</th>
<th>Frame rate (fps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>no</td>
<td>71.96</td>
<td>13.90</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>5</td>
<td>yes</td>
<td>90.30</td>
<td>11.07</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>no</td>
<td>48.14</td>
<td>20.77</td>
</tr>
<tr>
<td>1100 × 1100</td>
<td>3</td>
<td>yes</td>
<td>62.32</td>
<td>16.05</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>no</td>
<td>57.67</td>
<td>17.34</td>
</tr>
<tr>
<td>730 × 730</td>
<td>5</td>
<td>yes</td>
<td>64.26</td>
<td>15.56</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>no</td>
<td>39.53</td>
<td>25.29</td>
</tr>
<tr>
<td>730 × 730</td>
<td>3</td>
<td>yes</td>
<td>52.47</td>
<td>19.06</td>
</tr>
</tbody>
</table>
As the results show, the illustrative rendering technique continues to deliver results at rates suitable for real-time interaction. Even in the cases where we have a large $1100 \times 1100$ pixel view port, a large silhouette radius and compute curvature, we retain frame rates in excess of 10 frames per second.

We note that the performance is also still significantly affected by the curvature analysis and subsequent line stroke rendering. There should be some room for improvement there.

### 6.2 Functional results

On the whole, reception of our rendering technique is positive. Informally the cleanness of the rendered results is praised above all, commenting that the images as shown in figure 6.2 seem to have leapt straight from a medical textbook. As figure 6.3 shows, we manage to mimic the style of the hand drawn example we gave in section 3.1 quite well, if somewhat more detailed.

If we compare our illustration with a regular rendering procedure for clustered fiber bundles, as in figure 6.4, we note that thanks to the silhouette and contours it has become easier to identify the boundaries of clusters. The silhouette also gives the bundles a more solid look which somewhat helps with perceiving the depth order. Reduced crowdedness of lines makes it possible to clearly follow structure of the bundles.

Several voices within the BMIA research group at the Eindhoven University of Technology have shown interest in the rendering technique for other purposes. These include the visualizing of confidence domains on fibers or application to different tissue such as muscle.

The accompanying focus and context displacement system as shown in figure 6.6 works for navigating simple clusterings, with 6 to 8 bundles. Feedback included that the focus & context system looks as if we pick apart and shift around a set of building bricks, like the popular LEGOs. We can make an analogy to anatomical dummies here, whose organs can often be piecewise removed for closer inspection. However, when working with more complex data sets the technique currently requires too much manual tuning of the displacement modifier to truly be of practical use. Figure 6.5 is indicative of this.

The capability to define separate rendering styles for the focus and context was generally found to improve perception. Taking cues from hand drawn work, reducing the context’s colors to softer tones with a low contrast with the background works well, as does putting the entire context in a single sepia or gray scale tone. Reduced line detail works as well. This finding coincides with what we noted in section 4.4.2, when we discussed varying the thickness of hint lines in a fiber bundle. Thicker lines are used by the focus to draw attention, whereas thinner lines are used on the context. Figure 6.7 shows some of the possibilities.

Regrettably time no longer permitted integration of the rendering technique into the DTITool, but every effort was taken to make the integration path as straightforward as possible.
6.3 Examples

(a) Arbitrary viewpoint in space

(b) Axial plane
(c) Sagittal plane
(d) Coronal plane

Figure 6.2: Visualizing a set of 20 bundles clustered from fiber tracts tracked through a DTI volume
Figure 6.3: A side-by-side comparison of a hand-drawn legend of fiber bundles and a matching illustratively rendered clustering.

Figure 6.4: A side-by-side comparison of a regular non-illustratively rendered clustering and an illustratively rendered clustering, using our method.
Figure 6.5: Current focus and context technique does not work well when too many clustered bundles are loaded.
Figure 6.6: Focus & context active on a set of 3 bundles, where context bundles have been given lower hint line detail, thinner contours and colors that have lower contrast with the background.
(a) Softer colors and more thin, more sparse lines

(b) In grayscale

(c) Contours only

Figure 6.7: Some possible methods of styling the context bundles
Chapter 7

Conclusion

In this thesis we have described the need for a more abstract way of representing bundles of fiber tracts. We have fulfilled this need with an illustrative rendering technique supplemented by a focus & context system, that rely on basic principles of information removal and focus & context from traditional hand drawn illustration.

7.1 Contribution

Functionally, our illustrative rendering technique works well. Whole-brain white matter tract visualizations are clean and comprehensible. The clear boundaries introduced by the use of silhouettes and contours make for a clear visual partitioning of clustered bundles. Comments relating to the textbook appearance show that as an illustrative technique mimicking an illustrator’s approach the technique does well. We can conclude that the development of the rendering technique is a definite success with respect to the objective we started with.

We have shown how our rendering technique can be fit onto the GPU and have given performance data that indicates such an implementation is capable of performing in real time on current day hardware. We can expect this technique to scale up its performance very well as hardware improves. It is completely of linear complexity.

While the focus & context system works well for smaller amounts of bundles, it currently breaks down with larger sets, displacing many context bundles by a too great amount of distance. The basic algorithm is sound though and can be made to work with other bounding volumes if we adhere to the simple set of three guidelines specified in section 5.3.

The lack of integration with DTITool was a let down, but it is still a definite possibility. The rendering technique is self-contained. It takes a set of polyline fibers and outputs them as an illustrative rendering to the screen and does so without external dependencies. This makes it fairly easy to plug into already existing projects such as DTITool. Integrating the focus & context system, and particularly the animation tracks, will be harder. The former depends more on the application architecture and the latter requires a timer interface to be present which can fire animation events on an interval.

7.2 Limitations

Due to architectural decisions that would allow speedy integration of the rendering technique into frameworks such as DTITool, the algorithm is currently locked to a ‘one instance per
bundle’ approach. This is somewhat problematic with respect to the performance. From the
performance data we presented in section 6.1 it becomes clear that our rendering technique’s
bottleneck lies in the amount of bundles being rendered — or rather, it lies in the amount of
texture sampling taking place. For each bundle, for each pixel in the screen, for each pixel in
the structuring element a texture has to be sampled. That amounts to a very large amount
of texture samples being taken. In the most pessimistic case we have recorded, the number
of texture samples required leads to $1100^2 \times 5^2 \times 10 = 302,500,000$ samples. Performance is
currently limited by the dependence of the number of texture samples taken on the number
of bundles in the scene (the $\times 10$ in the above pessimistic estimation). The impact should be
profoundly lower on more modern graphics cards with better texture sampling capabilities
and especially on machines equipped with such technologies as Quad SLI, but it is still a
limitation that should be dealt with.

The focus & context system is currently limited by the type of bounding volume that is
being used. The current oriented bounding boxes used as the bounding shape give too much
overshoot with respect to the actual shape of the bundles, which causes too much shifting
around of fiber bundles. In general oriented bounding boxes are also not a good fit for arched
bundles such as the corpus callosum. To properly facilitate those kind of bundles, we would
require a finer collision structure and a finer method of determining the direction in which
and magnitude with which a clustered bundle is to be translated.

7.3 Recommendations & future work

There are a few points where the presented work can be improved. Most notable are the
noticeable dependency of the rendering speed of our illustrative technique on the number
of bundles and the inaccurate bounding volumes used for penetration tests in the focus &
context system.

The dependence of the rendering speed of our illustrative technique on the number
of bundles could be resolved by combining the output of all the instances of the halo rendering
process for all fiber bundles in the scene and sending that single combined image through
one instance of the image space algorithm to produce the silhouettes and contours. It would
require an additional lookup or index texture, which records to which bundle a rendered pixel
in the image belongs. Algorithm 4.6 should then be adapted to pick a correct color based
on this texture. (In fact, it may then require two additional textures. One with the bundle
index and one containing the line and fill colors for all bundles.)

On an application or implementation level, this requires changing the characteristics of
the rendering technique’s architecture from a ‘one instance per bundle’-approach to a ‘one
instance for all bundles’-approach. We should note that this requires significant changes to
the current implementation of the technique and how it would integrate with a framework
such as DTITool. Furthermore, the current method by which the focus & context system
operates requires separate entities that can be displaced. When only one entity processes all
of the data streams belonging to all the separate bundles, the focus & context system can
no longer displace the separate bundles. It should be merged into the class responsible for
implementing the rendering technique, or in some way be applied as a pre-filter to the data
entering that class. The last option is preferable, as it would leave the rendering technique
itself as a separate reusable entity.

A combined technique like we suggest does have another merit. It gives algorithm 4.6
access to multiple bundles simultaneously, which could be used to add nice contours where
different bundles intersect. Intersecting bundles currently clip into one another, which can
be ugly when viewed at certain odd angles. A possible way to tackle the problem is to
adapt algorithm 4.6 to not only compare relevant depth values to decide whether a pixel is
background, silhouette or contour, but also take into account the bundle index.

Lastly, the collision shapes of the focus & context system are currently based on simple
oriented bounding boxes, but can be applied to any bounding volume that adheres to the few
simple rules we described in section 5.3. Further investigation is required to see if hierarchical
structures like bounding box trees could improve the matter. Perhaps a radically different
method of measuring penetration in image space could be used instead. Such a change may
also more easily permit the rendering technique to migrate to a ‘one instance for all bundles’-
approach as mentioned earlier.
Bibliography


Acknowledgements

I would like to express sincere gratitude to both my supervisors, dr. A. Vilanova and dr. ir. H.M.M. van de Wetering, for their continual support. This thesis was made possible, thanks to their guidance and effort.

Further thanks go to ir. R. van Pelt for support and advice on technical matters regarding GPU programming and ir. T. Peeters for his advice on the inner workings of the BMIA group’s DTITool application. The insight gained helped greatly in accomplishing my goals.

A special thanks goes to my colleagues from the BMIA research group, for showing a continued interest in the work I’ve done.

Finally I would like to thank my parents for the opportunity I was given to undertake the educational career that led to this thesis, for standing by me through the trials of my education and for their unrelenting support during these last stressful months.
Appendices
Appendix A

Software architecture

In this appendix we show the architecture of the implementation of our illustrative rendering technique. We look at how our implementation fits into the libraries of the Visualization Tool Kit (VTK) and can be made to integrate with the existing DTITool under development with the BMIA group at the Eindhoven University of Technology.

Before that we also pay some attention to the library of reusable classes that was created to facilitate common tasks in OpenGL, such as setting up and binding vertex buffers or frame buffers, compiling shader programs or loading textures.

A.1 OpenGL abstraction classes

The implementation of our rendering technique relies on OpenGL, which has an API defined in flat procedural C. This does not fit well with modern object oriented architecture and program design. OpenGL has little in the way of utility functions and the like that can be used to handle common tasks, instead relying on other external C libraries to give this. It is also incredibly vague and non-intuitive when it chooses to handle many things through a plethora of not-quite-enumerated types. (A long list of integer pre-compiler definitions actually.)

For the rendering technique’s implementation, a set of reusable C++ classes was developed. These make it possible to work with OpenGL on an object oriented level and wrap a more intuitive API around common tasks. These classes were set up to be reusable outside the masters project. In particular: they can serve as the basis for a complete set of OpenGL wrapper classes for the BMIA group’s DTITool.

The set of OpenGL classes we present here contains more features than required for our rendering technique. Some of these classes originate from early investigatory work on DTITool. This was part of the work done for this thesis, to develop an understanding of the tool and how our rendering technique could be shaped to be integrated. The investigatory work for DTITool’s architecture consisted of upgrading earlier work on GPU-based streamline tracing into a GPU-based fiber tracking module for the DTITool.

The set of created classes can be split roughly into three categories: shader programs, textures and buffers.
A.1.1 Shader programs

There are two classes related to shader programs: `GpuProgram` and `GpuShader`, representing shader programs and their component vertex, geometry and fragment shaders.

![UML diagram showing the classes related to managing shader programs](image)

**GpuProgram** defines a factory method for adding new shaders to a program and instances manage the life cycle of their component `GpuShader` instances directly. The only way to remove a `GpuShader` is to ‘take it back to the factory’ by calling the `destroyShader()` method on the `GpuProgram` instance the shader belongs to. This is a common pattern implemented on all the OpenGL abstraction classes where appropriate.

Activating and deactivating a shader program is handled through the `bind()` and `unbind()` methods, which transparently handle cases such as trying to bind a program when it has not been compiled yet or trying to unbind it when it was not active to begin with.

Furthermore `GpuProgram` contains methods to (re)compile or (re)link a shader program after adjustments have been made. E.g. swapping out a vertex shader for another vertex.
shaders, adding an additional fragment shader, etc. It also contains methods that allow the setting of shader variables of the **uniform** and **varying** type.

If during any time the OpenGL library requires a program to be activated for a configuration change, the currently active program (if any) is saved and restored after the operation completes. The **GpuProgram** and **GpuShader** classes are set up in a way that minimizes side effects on the OpenGL state machine.

### A.1.2 Textures

There are three classes related to textures: **Texture2D**, **Texture3D** and **TextureDeclaration**.

![UML Diagram](image)

**Figure A.2:** A UML diagram showing the classes related to managing textures

The former two represent 2D and 3D OpenGL textures. The latter is a configuration or **declaration** object, in which various settings for a texture are pre-configured before it is used to create the texture. This is analogous to how buffers are declared, as we will see.

Activating and deactivating a texture is handled through the **bind()** and **unbind()** methods, which transparently handle cases such as trying to bind a texture when it has not been declared yet or trying to unbind it when it was not active to begin with.

Furthermore **Texture2D** and **Texture3D** contain a **fill()** method, which can be used to
load data into the texture. This is one of the few places in the class library where type safety could not be introduced.

A.1.3 Buffers

Two types of buffers are supported in the class library: frame buffers and vertex buffers.

Vertex buffers are supported by the following classes: *VertexBuffer*, *VertexBufferBinding* and *VertexBufferDeclaration*.

Analogous to this, frame buffers are supported by the following classes: *FrameBuffer*, *FrameBufferBinding* and *FrameBufferDeclaration*.

Vertex buffers and frame buffers can bind specific channels, such as foreground color, background color, depth, etc. These are expressed with the *BindingType* enumeration of a *VertexBufferBinding* or *FrameBufferBinding*. A suitable declaration object has to be set up beforehand to which such bindings are added. Both declaration objects have factory methods for their relevant binding class and manage their life cycle. The only way to remove a binding is by calling the relevant *destroyBinding()* method on the parent declaration.

Before use a buffer has to be declared with the help of a declaration object. This can either by passed into the constructor, or can be passed using the *declare()* method. The combination of a default parameterless constructor combined with the *declare()* method allows a *VertexBuffer* or *FrameBuffer* object to be instantiated on the stack rather than the heap, allowing use of the RAII pattern.

---

**FrameBufferDeclaration**

```
0.1
```

<table>
<thead>
<tr>
<th>member</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mWidth</td>
<td>int</td>
</tr>
<tr>
<td>mHeight</td>
<td>int</td>
</tr>
<tr>
<td>mBindings</td>
<td>FrameBufferBinding[0..*]</td>
</tr>
</tbody>
</table>

- containsBinding(bindingType: BindingType): bool
- getBinding(bindingType: BindingType): FrameBufferBinding
- createBinding(bindingType: BindingType, dataType: Data Type, textureUnit: int): FrameBufferBinding
- destroyBinding(bindingType: BindingType): void
- destroyBinding(binding: FrameBufferBinding): void
- getWidth(): int
- setWidth(width: int): void
- getHeight(): int
- setHeight(height: int): void

**FrameBuffer**

```
0.1
```

<table>
<thead>
<tr>
<th>member</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mBufferHandle</td>
<td>int</td>
</tr>
<tr>
<td>mBufferDeclaration</td>
<td>FrameBufferDeclaration</td>
</tr>
<tr>
<td>mBoundTextures</td>
<td>Texture2D[0..*]</td>
</tr>
</tbody>
</table>

- declare(declaration: FrameBufferDeclaration): void
- bind(): void
- unbind(): void
- getWidth(): int
- getHeight(): int
- getBoundTexture(): Texture2D

**FrameBufferBinding**

```
0.1
```

<table>
<thead>
<tr>
<th>member</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>mTextureUnit</td>
<td>int</td>
</tr>
<tr>
<td>mBindingType</td>
<td>BindingType</td>
</tr>
<tr>
<td>mDataType</td>
<td>Data Type</td>
</tr>
</tbody>
</table>

- getDataTextureUnit(): BindingType
- getBindingType(): BindingType
- getDataType(): Data Type

**Data Type**

```
<<enum>>
```

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELM_FLOAT</td>
<td></td>
</tr>
<tr>
<td>ELM_UNSIGNED_BYTE</td>
<td></td>
</tr>
</tbody>
</table>

**BindingType**

```
<<enum>>
```

<table>
<thead>
<tr>
<th>value</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BND_COLOR_ATTACHMENT</td>
<td></td>
</tr>
<tr>
<td>BND_DEPTH_ATTACHMENT</td>
<td></td>
</tr>
</tbody>
</table>

---

Figure A.3: A UML diagram showing the classes related to managing frame buffers
Activating and deactivating a buffer is handled through the `bind()` and `unbind()` methods, which transparently handle cases such as trying to bind a buffer when it has not been declared yet or trying to unbind it when it was not active to begin with.

`FrameBuffer` instances contain a `getBoundTexture()` function, which allow the programmer to retrieve a bound channel of the frame buffer (such as foreground color or depth) as an instance of the `Texture2D` class, which we discussed before. Note that in this case, the life cycle of the `Texture2D` class instances is directly managed by the `FrameBuffer` instance for which the textures were set up. (They match the `FrameBufferBinding` instances on the buffer's `FrameBufferDeclaration`.)

Figure A.4: A UML diagram showing the classes related to managing vertex buffers and transform feedback

`VertexBuffer` instances contain `fill()` and `fillRange()` methods, which can be used to load vertex data into the buffer. These are some of the few places in the class library where type safety could not be introduced.

`VertexBuffer` instances can be used with the transform feedback option introduced
through new OpenGL extensions. Transform feedback allows the programmer to redirect the output of a geometry shader back to a memory buffer, instead of sending it into the rasterizer part of the graphics pipeline. This mechanism comes in handy for GPGPU computations and several algorithms which require multiple passes before a final pass renders data to the screen. *VertexBuffer* contains a pair of binding methods *feedbackBind()* and *feedbackUnbind()* which explicitly bind and unbind the buffer for and from use with this transform feedback mechanism.

To further facilitate the transform feedback mechanism, two additional classes are present: *TransformFeedback* and *TransformFeedbackQuery*.

The first manages the activation and deactivation of the transform feedback functionality. It again contains factory methods for instances of *TransformFeedbackQuery* and again manages their life cycle internally.

The second can be used to set up a transform feedback query. Currently the only type of query supported is to ask OpenGL for the number of primitives it has output back into the currently bound transform feedback *VertexBuffer*.

### A.2 Integration with VTK & DTITool

DTITool is the BMIA research group’s tool kit for working with medical DTI volumes. It is largely built on top of the Visualization Tool Kit (VTK) library, which handles the data pipeline from reading from a file all the way to rendering on the screen. It has an extensive support library with all kinds of data filters and mappers, some of which DTITool relies on, some of which it augments with its own. DTITool uses the Qt (pronounced “cute”) library to manage and render its user interface, which links to the rendering view port VTK outputs to through a specialized integration widget. Both libraries are written in C++ and are platform independent, capable of compiling for the Windows and Linux platform, among others.

Our illustrative rendering technique was implemented in the class *SketchedFiberMapper*, a descendant class of VTK's general *vtkPolyDataMapper*, used to render polygonal data. Each individual bundle requires its own instance of the mapper, which leaves the mapper as purely a visualization agent of the data: no filtering or decision making on clustering takes place. Figure A.5 shows how this would fit into the general VTK pipeline. Implementing the rendering technique as a proper VTK mapper makes it an unobtrusive drop in replacement for the current *vtkPolyDataMapper* instance used by DTITool to render streamline based fiber tracts. Note however that data sets combining several bundles into one multiplexed stream of data will need to be hooked up to a customized data-splitting filter. It should be up to such a filter class to split the bundles into separate data streams for further processing.
INTEGRATION WITH VTK & DTITool

To integrate the rendering method into the DTITool a custom user interface control panel has to be written with Qt and linked together with an instance of the `SketchedFiberMapper`. This can be done similarly to the existing fiber rendering module in DTITool, which stitches the UI and mapper together through a `controller` class.

The `SketchedFiberMapper` class exposes all possible configuration options to the programmer: line color, fill color, silhouette thickness, contour thickness, width of halos, the family of Phong lighting constants, etc. All the controller needs to do is instantiate the mapper, attach it to the data pipeline and manage this list of settings.

A.2.1 A friendly warning

Be warned though, that the above makes it sound rather simple to build custom applications on top of VTK and build extensions to VTK. Much more simple than it is in reality!

VTK is a library that has been grown over a period of many years, rather than having been architected well from the ground up. This shows in various rather insidious ways when you start trying to extend it (not the least of which are problems with destructors in base classes not being called properly). A chronic refusal on the side of the VTK development team to adopt the C++ Standard Template Library (STL) available in the `std::` namespace does not help much either. It does explain why the bastardized runtime typing system (which is so needlessly complex it has to be accessed through a set of pre-compiler macros) exists: the standardized `std::type_info` construct and related design patterns would not be available...

Over the course of implementing the rendering technique and especially the focus & context system described in this thesis on more than one occasion was it the better idea to just 'build it yourself', unless you wanted it done wrong. VTK contains a wealth of domain specific knowledge though and that makes it very tough to pass up.