MASTER

Illustrative volume rendering on consumer graphics hardware

van Pelt, R.F.P.

Award date:
2008

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

Master's Thesis

Illustrative Volume Rendering on Consumer Graphics Hardware

by

Roy van Pelt

Supervisors:

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

Eindhoven, November 2007

Abstract

English

In the field of computer graphics, illustrative techniques are generally applied to produce stylised renderings. Typically, these simplified depictions produce much clearer images; comprehensibly presenting the content of the data.

We investigate a hardware-rendered implementation of an illustrative framework for rendering medical volume data, based on particle systems. This implementation allows interactive inspection of the volume data, exploiting parallelism in both consumer graphics hardware and particle systems. The particle systems provide a general and scalable rendering approach, founded just on the local data surrounding each particle.

The implementation presented in this thesis comprises a renewed version of the existing illustrative framework, VolumeFlies, proposed by S. Busking [6]. The proposed hardware-rendered particle system, as well as the real-time curvature estimation, rely on the recently introduced geometry shaders, and can be applied as general building blocks in future applications.

Dutch

Op het gebied van graphics, worden illustratieve technieken toegepast om gestileerde representaties te creëren. Deze vereenvoudigde afbeeldingen zijn doorgaans veel duidelijker en bieden een inzichtelijke weergave van de data.

Wij onderzoeken een hardware gebaseerde implementatie van een illustratief raamwerk voor het weergeven van medische volume-data, gebaseerd op particle systemen. Deze implementatie maakt interactieve inspectie van de volume-gegevens mogelijk en benut het parallellisme in zowel de grafische hardware als de particle systemen. De particle systemen bieden een generieke en schaalbare visualisatie-methode, gebaseerd op enkel de directe gegevens in de omgeving van ieder particle.

De implementatie die in deze thesis gepresenteerd wordt, behelst een vernieuwde versie van het bestaande illustratieve raamwerk VolumeFlies, van S. Busking [6]. Het hardware gebaseerde particle systeem, evenals de methode tot het schatten van de kromming op een oppervlak, zijn gebaseerd op de onlangs geïntroduceerde geometry shaders en kunnen worden toegepast als generieke modules in toekomstige applicaties.

Keywords

Illustrative rendering, non-photorealistic rendering, volume rendering, particle systems, consumer graphics hardware, GPU, GPGPU, geometry shader, transform feedback

Contents

List of Acronyms & Abbreviations

API	Application Programming Interface
CG	C for Graphics
CT	Computed Tomography
DVR	Direct Volume Rendering
FBO	Frame Buffer Object
GLSL	OpenGL Shading Language
GPGPU	General Purpose Computations on GPUs
GPU	Graphics Processing Unit
GUI	Graphics User Interface
HLSL	High Level Shading language
IVR	Illustrative Volume Rendering
MIP	Maximum Intensity Projection
MRI	Magnetic Resonance Imaging
NPR	Non-Photorealistic Rendering
Pixel	Picture element
QT	Quasar Toolkit
Texel	Texture element
TBO	Texture Buffer Object
VBO	Vertex Buffer Object
Voxel	Volume element

Chapter 1

Introduction

"What is real is not the external form, but the essence of things."
- Constantin Brancusi

Since the history of mankind, images have been used to convey all kinds of information. It is in the nature of human beings to be able to easily extract this information through visual interpretation. Therefore images have always been of great significance in the way people communicate. A captivating challenge in this respect is to generate images, such that they contain the necessary information, without loss of intelligibility.

With the introduction of the computer, a new science discipline known as 'computer graphics' originated. The computer offered innumerable new possibilities to create, edit and show images; and therefore new ways to capture essential information into visual representations. Today still 'computer graphics' is a prominent field of research and is continuously evolving. The field addresses a great variety of topics, which can be considered as independent research disciplines. For this project results from several disciplines have been combined into a concept called 'Illustrative Volume Rendering'.

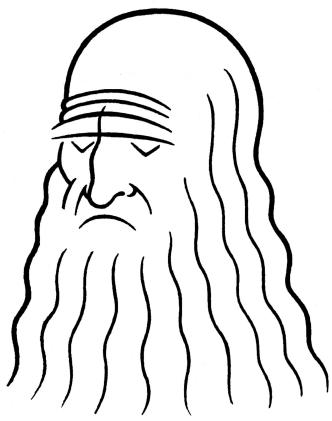


Figure 1.1: Leonardo da Vinci

Although initially the main focus within the 'computer graphics' field was directed to creating images that resemble reality as seen in the world of photography, nowadays also the effect of illustratively visualising information is being studied. Illustrations allow us to emphasise important features within an image, such that they can effectively carry out the underlying message. In consequence the informative value of an image can increase substantially if it is cleverly distorted. This fundamental principle has been applied to various kinds of artwork. Exceptionally empathic examples can be found in comics and caricatures. However also paintings and drawings attract the attention to important features of the image.

For this thesis, several artistic techniques were studied and applied to medical data. Already in the late 15th century Leonardo da Vinci used sketches for his studies into the human anatomy. Even though his sketches weren't always anatomically correct, he was able to show parts of the human body emphasising his findings. Today still, literature describing human anatomy is mostly accompanied by illustrations instead of photographs. Illustrations are often much clearer, since the visual exuberance often seen in photographs is omitted.

The goal for this thesis is to study and implement several illustrative techniques that can be automatically applied to medical volume data. This allows us to generate stylized images of parts of the body, acquired through any form of radiology. The envisioned program should be capable of visualising these stylized three-dimensional models at an interactive speed. Furthermore the illustrative representation is supposed to be combined with a more realistic view on the dataset, where the illustrative depiction is applied as a context.

1.1 Terminology

The work described in this thesis addresses multiple research areas, each with it own jargon. This section introduces several common terms that are important in the context of this project.

In the area of 'computer graphics' the term *rendering* is used for the process of generating the picture elements (*pixels*) that construct an image. The rendering process in itself however, does not deal with the main issue at hand: 'What message should the (generated) image express?'. For some images the aim is to come near reality, as seen in the world of photography. The process of generating these images is called *photorealistic rendering*. However in the context of this project we strive for stylized images, as seen in the world of art. Rendering these image is somewhat awkwardly called *non-photorealistic rendering (NPR)*.

The term NPR was firstly introduced by Winkenbach and Salesin [33]. At the time this naming convention seemed to be a logical answer to the prevailing mind-set of persuing photorealism. However the term oddly demarcates an area of research, by describing what topics are excluded. Throughout the years several other terms were proposed, such as *expressive, artistic, interpretive* and *illustrative rendering*. These terms provide a much better indication of the essence of the research field, which is to automatically create illustrative images. Even though from a historical perspective the term *NPR* is used most often, we will adopt the term *illustrative rendering* for the remainder of this thesis. In order to oppose confusion we determine that for this thesis the terms image, drawing and illustration refer to illustrative depictions, unless another interpretation is obvious from the context.

Another research challenge in the area of 'computer graphics' is found in visualising three-dimensional data sets, often referred to as *volumes*. One can consider a volume as a pile of images, together defining one or more features in three-dimensional space. The process for generating images that visualise these features is called *volume rendering*. Volume rendering aspires to visualise volumes from an arbitrary angle, without using an explicit definition of geometry. When applied in a medical context, a typical data set is acquired through a CT or MRI scanner.

The introduction raised the objective of visualising medical data in an illustrative way. Such an approach requires a combination of techniques used in *volume* and *illustrative rendering*. Obviously naming this multi-disciplinary concept boils down to the term *illustrative volume rendering* (IVR). This concept will be incorporated in a software *framework*, which

comprises a modular approach to stylise volumetric data. The architecture of the framework will be elaborated in chapter 3.

There are numerous approaches that lead to a decent realisation of such a framework. One could think of well known volume rendering approaches, such as *raycasting* and *texture slicing*. Such approaches typically construct an *implicit surface* using the data set, whereas from an illustrative point of view a discrete representation of the data is preferred. Therefore a point representation of the volume seems to be an appropriate starting point.

Generating a point representation from a volume involves some intelligence in order to initiate and maintain the points. A similar concept can be found in the notion of *particle systems*. A particle is a small element, that consists of a set of properties describing the particle. In a particle system, particles originate, displace and vanish based on the set of given properties.

1.2 Motivation

Illustrative rendering is a rather new field of research of which the value has been doubted by a vast crowd of researchers. One reason for their doubt was already mentioned earlier, namely the vague boundaries outlining the field. Furthermore critics question the feasibility of generating art by any kind of computer system, since the deterministic behaviour of a system contradicts with the intuitive and emotional approach of an artist.

Despite the adversary, there is also a steady group of researchers that supports illustrative rendering. Different motives are proposed to justify the research. An obvious cause is found in the *artistic* results produced by different illustrative rendering systems. A lot of people are intrigued by the question if it is possible to automate the creation of art. Furthermore illustrative approaches can be applied in order to beautify images. This *aesthetic* motive has a more commercial base and occurs regularly in the marketing of expensive products. Advertisements of houses, kitchens and cars are often stylized to pretend a certain perfection. Finally illustrative techniques can be applied in order to clarify the content of an image. Simplifying the representation of an image, and hence the removal of visual clutter, often results in more *comprehensible* images.

For the framework presented in this thesis, comprehensibility of the images is the main argument to study and apply illustrative rendering techniques. However we need to question the soundness of the resulting images, especially since all adopted techniques are applied to medical data sets. Medical volumes conceal an immense quantity of information, for which a large variety of visualisation methods have been proposed. Yet no technique is capable of showing all latent information of a particular volume. More often a technique is used to visualise a specific region or property of interest. The illustrative approaches presented in this thesis also visualise just a fraction of the information contained in the data. However the sparse representations also greatly simplifies the resulting images, emphasising properties like tone and curvature. Furthermore such simplified depictions offer various possibilities to combine with other visualisations.

The majority of the literature describing the human anatomy and physiology uses illustrations, instead of photographs. Authors also observe the gain in clarity when using simplified representations. Special studies are established in order to investigate medical illustration techniques in general. Several methods have been proposed, automating the process of creating such illustrations. In particular we are interested in systems that apply illustrative

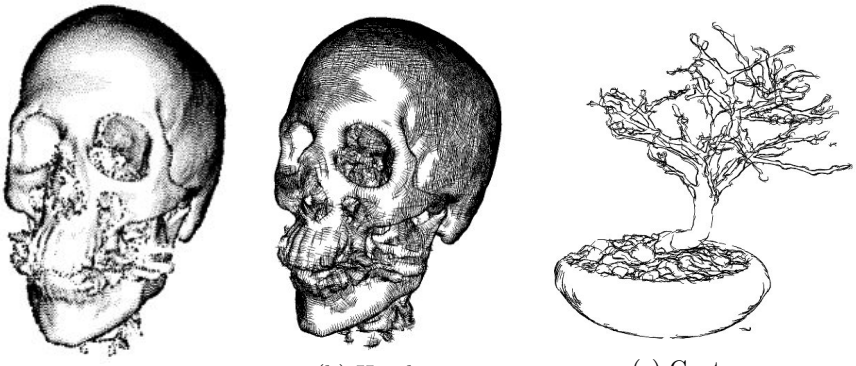
techniques to volumetric medical data.

The usefulness of such a system heavily depends on the way the user can interact with the system. Typically illustrative rendering approaches are based on various parameters. Interactive adjustment of these parameters and real-time visualisation of the data provides a valuable contribution to the user experience.

1.3 Previous work

In the past few years a substantial amount of research addressed illustrative rendering by imitating artistic styles. A large share of the proposed methods is based on explicitly defined geometry, whereas other approaches rely on the concept of volume rendering. An appealing example, based on *direct volume rendering* (DVR), is known as 'VolumeShop' [3]. Direct volume rendering approaches project the entire data set onto the image plane, whereas *indirect volume rendering* approaches project a geometric representation of the features extracted from the volume to the image plane. Chapter 2 will elaborate on several volume rendering techniques.

The illustrative volume rendering framework, underlying this project, was introduced by S. Busking in his master's thesis [5]: *'VolumeFlies - a smart-particle-inspired framework for illustrative volume rendering'*. The framework, called *'VolumeFlies'*, includes a diversity of illustrative styles, based on an indirect volume rendering approach. An article describing VolumeFlies will soon appear in 'The Visual Computer Journal' [6]. The illustrative styles contained in VolumeFlies are listed below, together with the main related publications found in literature.



(a) Stippling (b) Hatching (c) Contours

Figure 1.2: VolumeFlies illustrative styles

The first illustrative style is called *stippling* and is depicted in figure 1.2a. Such a point-based approach has been applied directly on volume data by Lu et al [24]. They provide a concept to control shade by adjusting the point density, emphasising contours and curvature. Their system however often produces noisy images, because of the absence of *visible surface calculation* (VSC). Methods for visible surface calculation often rely on geometry that veils unwanted surfaces. In our application this geometry should consist of a surface which removes the backside surfaces of the feature under consideration. A well known surface creation method, based on sparse data, is called *surface splatting*. An important contribution to splatting theory was delivered by Zwicker et al. [34].

Another illustrative style is called *hatching*. This style also resembles a pen-and-ink drawing and is depicted in figure 1.2b. Several volume based techniques have been proposed. For instance Nagy et al. introduced a method [27] that directly depends on volume characteristics and results in images that seem to be sketched quickly. A similar method was presented by Dong et al. [12]. However they show more densely hatched images, together with contours. Both systems are computational expensive and do not deliver real-time results. Also the system by Hertzmann and Zorin [16] does not obtain interactive speed. However they propose a method, based on piecewise-smooth surfaces, which allows smoothing the directions of the hatches. Moreover they have addressed the influence of lighting on the hatches.

Finally VolumeFlies includes volumetric *contours*, depicted in figure 1.2c. One of main concepts was presented by Csébfalvi [10] et al.. Their method obtains a contour representation from volume data, though without a reliable parametrisation controlling the line thickness. The approach proposed by Kindlmann et al. [17] on the other hand, does put contour thickness under user control, although it comes with extra computational expenses.

An important measure that occurs in most of the above mentioned rendering styles, is the *curvature* of an implicit surface of a volumetric object. A step-by-step method for calculating this curvature is presented in [17]. Moreover we require such curvature computations to be calculated in real-time. A method, presented by Hadwiger and Sigg [15], exploits the hardware capabilities to obtain higher order characteristics from the volume in real-time. Subsequently curvature information can be calculated based on the reconstructed partial derivative information.

The graphics hardware capabilities are largely dependent on the *graphics processing units* (GPU), providing exceptional computational power for graphics processing. The availability of this high processing speed has urged researchers to endeavour more general computations on the GPU. This kind of research evolved into a separate discipline named *'General Purpose Computations on GPU'*, which is abbreviated to GPGPU. A specific topic of interest is the implementation of GPU-driven particle systems.

Several approaches have been proposed throughout the years. In the year 2004 Kipfer et al. presented the 'Uberflow' method [18], which allows visualisation of a large amount of particles through proper use of the GPU. Furthermore their system prevents particle collisions and provides depth ordering of the particles. A similar system was presented by Krüger et al. [20]; including several ways to visualise the particle flow.

A different approach to particle systems is offered by Meyer et al. They propose a method [25] that optimally distributes particles on a surface. Eventhough most methods employ particle systems for visualisation of flow fields; methods similar to the one presented by Meyer show that particle systems can very well serve to visualise static characteristics of a geometrical object. For instance Cornish et al. [8] already observed the value of particle systems for illustrative rendering. They present a complete framework for illustrative rendering, including automated frame-to-frame coherent line drawings and paintings. Similarly S. Busking adopted a particle system as the foundation of the VolumeFlies environment.

Furthermore there are a couple of books that are of particular interest in the context of this project. The book 'Non-photorealistic rendering' [14] provides a profound overview of the illustrative rendering techniques, though there have been new developments since the book was published. An overview of the developments in the GPU field in general was found in the 'GPU Gems' series. Specifically 'GPU Gems 2' [29] turned out to be of great value. An in-depth book on real-time volume rendering was found in 'Real-time volume graphics' [13], which includes an extensive part on illustrative volume rendering.

1.4 Objectives

The main objective for this thesis is to speed up the VolumeFlies framework. A new hardware-accelerated approach should deliver real-time interaction with the user. Consumer graphics cards nowadays provide exceptional computational power, using several *graphical processing units (GPU)*. The intention of the project is to implement the complete VolumeFlies functionality, exploiting the computational advantages of the graphics hardware.

First of all it is important to identify the performance bottlenecks in the current software-rendered framework. Subsequently new hardware-rendered approaches should be introduced, increasing the system performance. This involves studying current GPU-based particle system methods, as well as other GPGPU techniques. Using this knowledge, current algorithms must be altered to fit the GPU, retaining the VolumeFlies functionality. The newly found methods are allowed to be implemented for specific graphics cards, but the algorithms and software decomposition are expected to be as generic as possible.

A secondary object is to integrate this GPU based illustrative approach with a GPU based direct volume rendering system. The line of though is to use the illustrative visualisation as a context for the element of interest, which can be visualised in a more realistic way. For example the brain cortex could be sketched, helping a physician to situate a realistically rendered tumor.

Ultimately the performance of both the software and hardware rendered framework should be measured and compared in order to get insight in the performance gain of a GPU-based implementation.

1.5 Organisation

This document consists of eight chapters, of which the first one is the introduction. The reader is strongly encouraged to read through the chapters chronologically. The second chapter provides background information to several topics that are combined for the project. The chapter treats volume and illustrative rendering, as well as the concept of particle systems. Important aspects of the GPU that were engaged in the system are also discussed in chapter two.

Starting from chapter three the reading matter becomes more specific. The concept of illustrative volume rendering is described more elaborate by means of the system requirements. Furthermore the chapter describes the architecture of newly implemented VolumeFlies environment.

From the system description we move on to the actual methods that build up the framework. In chapter four the preceding software-rendered framework is addressed. An overview is given of the architecture of the framework, as well as the applied methods in order to realise the illustrative renderings.

The core of this thesis however, is discussed in chapter five. This chapter gives an in-depth view on the new framework, which is completely hardware-rendered. Apart from the introduction, nearly the same structure is preserved from chapter four, which allows easy comparison between the systems. Chapter five will emphasise the renewed methods that allows the software-rendered framework to be ported to the GPU.

Subsequently, chapter six provides an overview of the obtained results. This involves a performance analysis of both frameworks. Furthermore, the performance measures will be

compared to evaluate the benefit of the hardware implementation.

Chapter seven gives a concise excerpt of the thesis, which serves as a basis for the conclusion. This conclusion is made explicit in the final chapter, namely chapter eight. After the conclusion, the reader will find a list of figures, the bibliography and the acknowledgements. Finally the appendices describe the technical peculiarities of the project, such as a system architecture diagram.

Chapter 2

Background

Recall from the first chapter that VolumeFlies combines various topics from different research areas; each with their own terminology. This chapter provides general background information to these areas, establishing the essential knowledge to interpret the newly proposed methods, further on in this thesis.

2.1 Volume rendering

The field of 'computer graphics' has a strong interest in visualising objects from an arbitrary angle in three-dimensional space. Traditionally objects were modelled with surface respresentations. Later on, a considerable amount of researches started to work with volumes containing the objects of interest.

In the first chapter the notion of a volume was introduced. Theoretically such a volume is assumed to be a continuous three-dimensional scalar field. In practice, however, a volume consists of a three-dimensional matrix of real-numbered scalar values, positioned at nodes of a regularly spaced lattice. The values of these nodes are either the result of a simulation or a measurement.

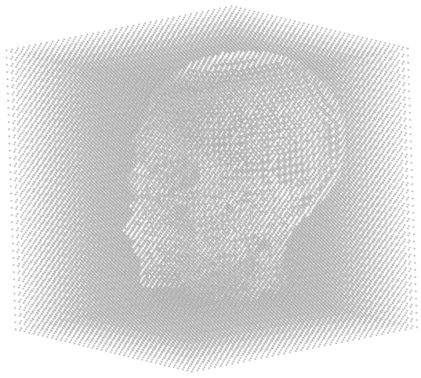


Figure 2.1: 3D scalar volume

Each node represents a volume element, generally known as a *voxel*. The main reasoning is similar to the two-dimensional case, where each picture element, or *pixel*, represents a measurement at a certain position in an image. Even though pixels are commonly interpreted as small square elements of an image, one should be aware that each pixel includes only a single measurement, and thus a single scalar value. These scalar values represent the intensity value for its square neighbourhood. Similarly in three-dimensional space we observe that each measurement represents a small cubic subspace in the volume. Figure 2.1 depicts a volume as a three-dimensional uniform scalar field.

Volume rendering simulates the propagation of light through a participating medium, represented by the volume. In this optical model, light flows through the volume, where the light can be absorbed, scattered or emitted, dependent on the medium characteristics. The

evaluation of this model can be divided into multiple stages, known as the volume-rendering pipeline. This pipeline is accurately explained in section 1.6.1 of the 'Real-time volume graphics' [13] book. There are various methods implementing this pipeline.

The most popular approach is *ray-casting*, for which a number of rays traverse through the volume. Popularity of the ray-casting method is mainly due to the fact that the concept is particularly intuitive. One can think of the rays as a narrow bundle of light, propagating through the volume and evaluating the volume rendering equation.

Other ways to approximate the volume rendering equation exist. For instance there is the method known as *splatting*. One can think of this method as virtually throwing the voxel colour to the viewing plane, as if it came from a small bucket of paint. The resulting spatter or splat on the viewing plane is called a *footprint*. These footprints are accumulated in image space values for all voxels.

A different approach to splatting, comprises a surface rendering method called *surface splatting*. Instead of projecting the voxel intensity to the image plane, a set of geometric objects is rendered, which just projects the voxels that are part of an implicit surface. This opaque set of geometric objects consists of a large number of flat disks, which are scaled and oriented to fit the implicit surface, contained in the volume. In image space these flat disks resemble the footprints, known from the volume splatting approach.

Volume rendering has gained popularity over time. Initially volume rendering was applied to realistically render effects such as water, clouds, smoke and fire. Also the game-industry realised the potential of volume rendering for the visualisation of these effects. Furthermore, volume rendering was practised for scientific reasons, offering a flexible visualisation framework mainly for physical simulations and medical data.

Both the game-industry and scientists realised, that using the capabilities of a graphics card could lead to a significant performance gain. The first GPU-based volume rendering approaches where based on a set of two-dimensional textures. These textures consisted of fixed set of volume slices, taken for each direction along the main axes. Depending on the view direction, the correct set of slices was projected onto the viewing plane. Later on, graphics card manufacturers introduced three-dimensional textures. This development allowed a new method, called 3D texture mapping, which divides the three-dimensional texture in slices perpendicular to the view direction. With the introduction of shader programs, the GPU become programmable. As a consequence a GPU-based raycasting approach became available. Nowadays most volume rendering approaches can be developed using the GPU, thanks to the considerable increase of flexibility and computational power of the graphics cards.

2.2 Illustrative rendering

Research in illustrative rendering comprises two major divisions. On the one hand, there is a notable group of researchers trying to *simulate* artistic media. Such systems are mainly based on volumetric models, simulating physical properties of the substrate, painting tools and products. For example one can think of simulating water paintings [21] or ink drawings [7], by modeling characteristics of the substrate, water or ink and brushes.

On the other hand researchers have shown great interest in *imitating* artistic work. A style of art is being modeled, without explicitly defining real-world characteristics. For the system presented in this thesis we have chosen to imitate several artistic styles. This paragraph lists the selected illustrative styles and provides intuition to the concepts. Please note that chapters four and five elaborate on these styles in the context of a volume rendering setting. This chapter will furthermore provide a list of alternative illustrative styles found in literature.

The first style which is part of the system is called *stippling*. This technique occurs often in pen drawings, where dots characterise the surface structure of an object. Stippling techniques are known to emphasise curved areas of the object surface, either by changing the *density*, or by changing the *size* of the dots.

Manually stippled drawings consist of a large amount of *uniformly scaled* dots. Varying the density of the dots allows a change of *tone* in different areas of the drawing, as depicted in figure 2.2. Although careful placement of dots results in clear images, the method is often dismissed because of its expensive creation. However, nowadays computer systems are capable to take over the handwork, through an automated stippling process. Several stipple placement approaches have been proposed, but only few apply to volume rendering [24].

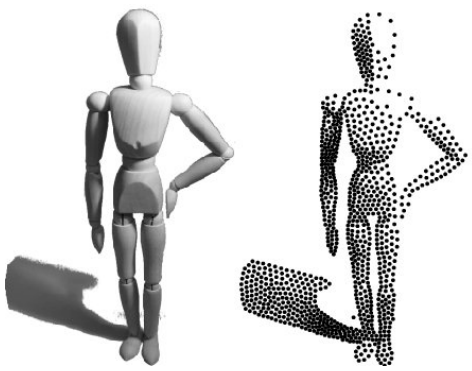


Figure 2.2: Stippling: tone variation [30]

Similar effects can be achieved by scaling the dots. Areas with a great amount of relatively large dots appear darker than areas with small sized dots. Again this results in a stronger representation of the shape of the object. The idea of scale based stippling is inspired by the concept of halftoning. Halftoning reproduces original images by means of equally spaced dots of varying size. These images often occur in printed matter, since printing expenses decrease, compared to the original image. Furthermore the picture quality remains acceptable.

The second style that is part of the system is called *hatching*. Also hatching is a pen drawing technique, however strokes are applied instead of dots. Hatching, as well as stippling, are illustrative techniques that occur quite often in technical, architectural and medical illustrations. The efficient and concise communication of information brought forward by these techniques, was already known in the 15th century by Leonardo da Vinci; whose caricature is depicted in figure 1.1. Amongst many other scientific interests, he was fascinated by the human anatomy. In order to communicate his learnings he used sketches, similar to the hatching approach we devise. Other intriguing anatomical drawings where made by the well-known Flemish anatomist Andreas Vesalius in the 16th century. Some examples are depicted in figure 2.3.

In the first chapter we've observed that these illustrative techniques are still used in modern

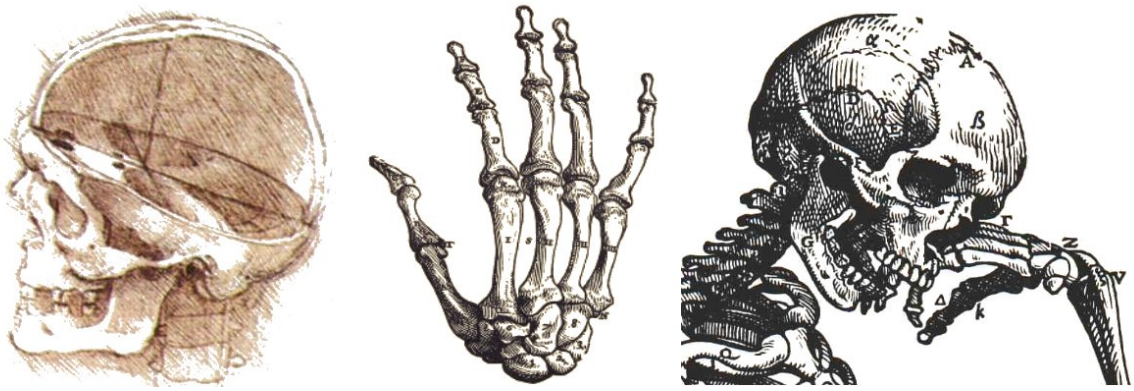


Figure 2.3: Sketches by Da Vinci (left) and Vesalius (middle and right)

anatomy and physiology literature. Yet the creation of handmade hatched drawings is rather expensive. An automated approach allows easy generation of images, which are above all anatomically correct, when based on real medical volume data. Furthermore an automated approach allows to dynamically interact with the three-dimensional illustrative depiction of the features in the volume.

The third style in the systems applies *contours* to the object. In literature the use of terminology regarding this subject is often inconsistent. It is therefore important to specify what exactly we mean by contours. We mainly adopt the terminology as presented by DeCarlo et al. [11]. Firstly we should make a clear distinction between contours and silhouettes. The contours of an object is the set of lines that demarcates areas where the objects surface turns away from the viewer. These lines therefore define both the outer boundary of the object, as well as curved features on the object. This notion of contours can be expanded to the set of *suggestive contours*. Suggestive contours demarcate the default contours as well as the areas where the surface presents a contour, when slightly changing the viewpoint. In contrast to contours, a silhouette line only represents the outer boundaries of the object, indicating the image-space transition to the background.

In order to reach a full-grown illustrative volume rendering framework, other styles could be included. An often used technique is to shade the object similar to cartoon drawing. This *toon shading* style was initially proposed by Gooch and Gooch, and has been extended by Barla et al [2]. Furthermore one could think of styles like water- and oilpaintings, for which the additional value should be considered.

Apart from the illustrative styles, other valuable additions could be included to extend VolumeFlies. For instance *cut-aways* would be a proper extension, which requires the data to be segmented. Using cut-aways in combination with *ghosting* has proven to be a powerful way of visualising data and emphasising the main area of interest [3]. A cut-away shows a segment of the volume, as if it was cut from the volume; while the ghosting indicated the original location and size of the highlighted segment.

2.3 Particle systems

Particle systems cover the third main research topic that is included in our framework. Again some mingled terminology should be clarified. In literature both particle-based and point-based systems occur and definitions are occasionally interchanged. In a particle system each point is given a set of properties, which influences the particles behaviour in space and time. In point-based systems however the points are just a representation of characteristics of the data, without 'carrying' any specific information.

Particle systems iteratively execute two stages. First it executes the *simulation* stage, updating the particles properties. The second stage is the *rendering* stage, which allows different types of visualisations of the particles. This generic two-stage approach has proven to be valuable in various applications.

The main application for particle systems is found in various simulations. These simulations include physical dynamics affecting the behaviour of the particles, mostly implicating a particle flow. Research focuses mainly on gas and fluid flow models, where uncoupled particles displace over time. It is the responsibility of the model to detect and resolve possible collisions with objects or neighbouring particles. Krüger et al. presented a general particle system approach [20], which is hardware rendered and clearly distincts between the simulation and rendering stage.

Furthermore particle systems are often applied to visual effects, for which geometry is hard to capture. Typical examples are smoke, fire and water. Particle systems are able to realistically visualise these effect, including their dynamic behaviour. Another example can be found in the simulation of cloths, for which the particles are coupled. Modeling the physical properties of the cloth results in realistic draping of the fabric.

The generic approach of particle systems equips us with the means to employ the concept in a different context. In VolumeFlies particles are engaged to define an implicit surface in the volume, creating a starting point for the illustrative styles. The particles are initiated near the surface, after which they are redistributed to a smooth and equally spaced set on the surface. The particle repulsion scheme proposed by Meyer et al. [25] is adopted in order to realise the particle redistribution. Unfortunately the latter approach is computational expensive, resulting in a dilatory redistribution of the particles. In chapter five we present a GPU-based solution, which considerably increases the redistribution speed.

2.4 Consumer graphics hardware

Rendering illustrations from volume data at interactive speed requires a considerable amount of computational power. Consumer graphics hardware nowadays delivers this power through modern graphics processing units. An important aspect of this thesis includes exploiting the strengths of the GPU to accelerate the illustrative framework. This paragraph introduces the global characteristics and operation of the GPU.

2.4.1 Computational power

The main motive for using a GPU is the remarkable amount of computational power it delivers. Although not a first-rate measure, often the number of transistors on a chip is used to give some intuition of the computational power. In 1965 Gordon Moore stated that the number of transistors on a chip would approximately double every 18 months. For CPUs this still holds today and is known as 'Moore's Law'. For GPUs however we see that the number of transistors increases even faster than Moore's Law. This remarkable trend creates possibilities to speed up computational expensive applications. However, in order to exploit the potential performance gain, the algorithms need to be redesigned to make sensible use of the hardware. Figure 2.4 depicts the increase of transistors over time for both CPUs and GPUs.

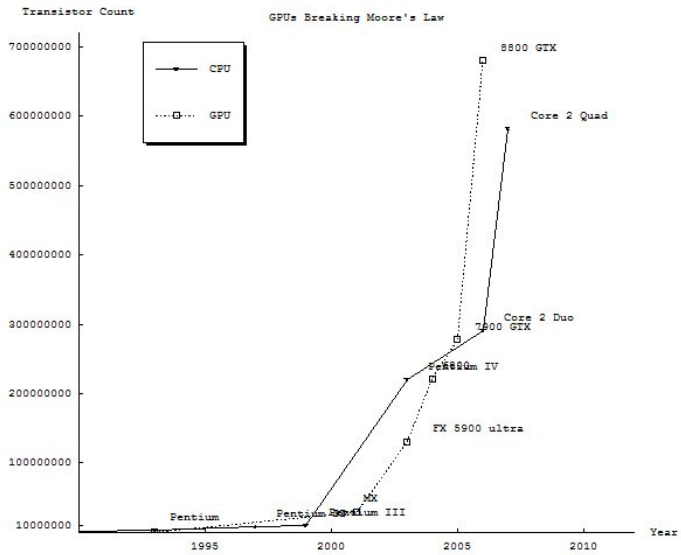


Figure 2.4: CPU/GPU transistor count (en.wikipedia.org/wiki/transistor_count, nvidia.com)

In a way, this comparison is not entirely legitimate. The transistor count gives a fair estimation of the technical progression of both CPU and GPU. However the architecture of the processing units differs considerably. As a result, the algorithms should be constructed to fit the GPU processing pipeline. Only then a GPU program optimally employs the hardware capabilities.

From the given reasoning, we can conclude that the GPU potentially delivers a performance gain, but only if the algorithms of the framework fit the GPU architecture. The latest NVidia graphics hardware was used to accelerate the framework. Performance characteristics of the card are given in table 2.1. The remainder of this paragraph will elaborate on the architecture of this particular card.

NVidia GeForce 8800 GTX	
Stream Processors	128
Core Clock (MHz)	575
Shader Clock (MHz)	1350
Memory Clock (MHz)	900
Memory Amount (MB)	768
Memory Interface (bit)	384
Memory Bandwidth (GB/sec)	86.4
Texture Fill Rate (billion/sec)	36.8

Table 2.1: Performance characteristics

2.4.2 Programmability

A GPU is constructed specifically for graphics operations and relies on the general graphics processing pipeline. This pipeline transforms the initial set of vertices into pixels on the screen. This process can roughly be divided into three stages. The first stage applies operations to the set of vertices, resulting in a set of *primitives*. The primitives can for instance be lines or triangles. The next stage decomposes this set of primitives into smaller units corresponding to the pixels in the destination frame buffer. These units are called *fragments* and resemble a potential future pixels. The fragments however can still be altered by several operations. These operations together form the final stage of the pipeline. One of the most important operations in this stage is *texturing*, where texture elements are assigned to the fragments. The general graphics pipeline is depicted in figure 2.5.

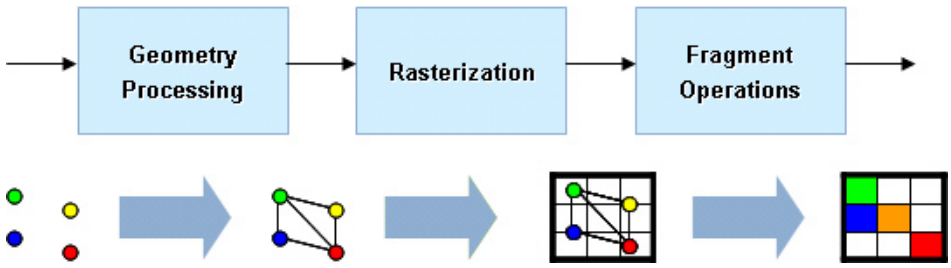


Figure 2.5: Graphics pipeline

In consumer graphics hardware, the operations of the graphics processing pipeline are implemented, resulting in a remarkable acceleration of the process. Initially the hardware only offered rasterisation of pre-transformed triangles and multi-texturing. However when the GPUs evolved, more and more operations were delegated to the hardware. Second generation GPUs allowed three-dimensional vertex transformations and lighting. However it was only in the third generation that vertex and fragment processing became programmable. The programs substituting the vertex and fragment processing are called *shaders* and initially needed to be implemented in a specific GPU assembly language. Nowadays the greater part of the graphics pipeline is programmable through a high level *shading language*. Most frequently used languages are 'C for graphics' (Cg) by NVidia, 'High Level Shading Language' (HLSL) by Microsoft and NVidia and finally the OpenGL Shading Language (GLSL), which is part of the OpenGL specification. For the framework we have chosen for the GLSL language, since it is cross platform and well supported by different hardware vendors. The hardware graphics pipeline is depicted in figure 2.6. A more extensive overview of the GPU history and the shading languages can be found in [32].

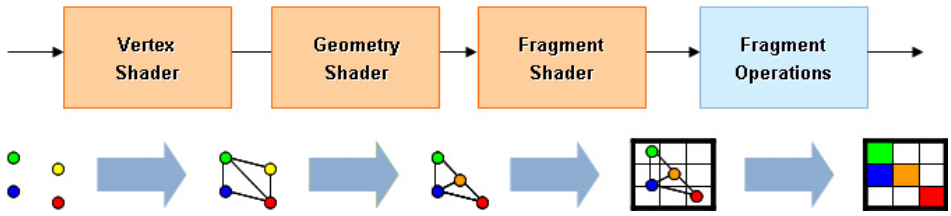


Figure 2.6: Programmable hardware graphics pipeline

In figure 2.6 there is one stage that differs from the general graphics pipeline. The geometry shader is a recent development, which further expands the programmability of the GPU. With the geometry shader one can create, transform and destroy geometry, as depicted in figure 2.6. This new stage is of great importance to this project and will be elaborated in the 'extensions' paragraph.
With the insight that the GPU executes its own programmable graphics pipeline it is meaningful to concentrate on the global execution model. A developer can build software in any high-level language. The main program should include the shader programs that need to be executed by the hardware. Next the program asks the OpenGL driver to compile and link the hardware specific shader program, using the OpenGL API. Finally the driver will upload the program to the GPU. The execution model is depicted in figure 2.7.

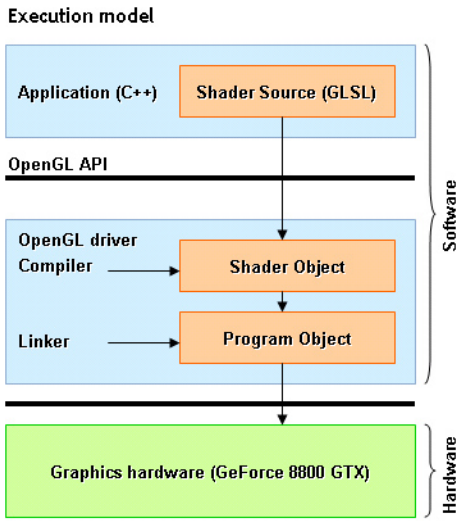


Figure 2.7: Execution model

2.4.3 Parallelism

The main strength and simultaneously the biggest constraint of the GPU is parallelism. From table 2.1 we observe that the chosen GPU contains 128 stream processors, which means it is capable to execute 128 processes concurrently. Former generation GPUs had a fixed number of processors allocated for vertex processing and equally for fragment processing. However when the workload of a particular application heavily depends on either vertex processing or fragment processing this resulted in idle processing units.

The new architecture, presented in [9], solves this inefficiency. The new architecture, called the *unified shader architecture*, is schematically depicted in figure 2.8. It involves a drastic change where each processing unit is able to process any shading thread. Therefore the thread processor of the graphics hardware (figure 2.8) automatically employs load balancing of vertex, geometry and fragment processing.

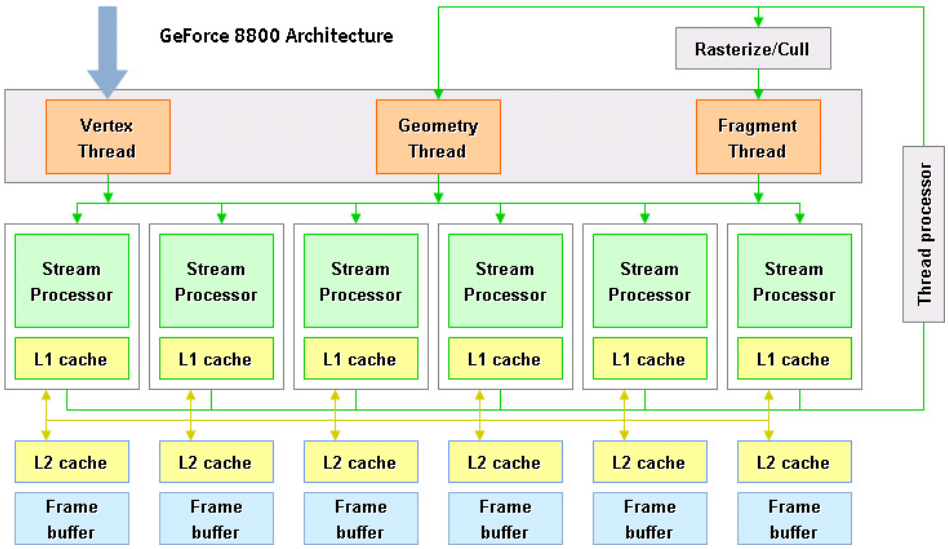


Figure 2.8: Unified shader architecture

Note that each processing unit is identical and is able to process either a vertex, geometry or fragment thread. Programming these processing units is different from the common serial approach seen in CPUs. The input data is considered to be a stream of elements, processed by instructions in one of the shading threads. Subsequently, data is returned in a separate output stream. The GPU can now function completely data parallel, because of the structured memory accesses in the stream model.

Other vendors of GPUs, like ATI, are now also changing to architectures implementing generic processing units. The framework in this thesis heavily depends on this new architecture, where the main workload is delegated to the vertex and geometry processing stages.

2.4.4 Extensions

Computer-game industry is one of the fastest growing segments in the entertainment sector, converting up to 60 billion dollar worldwide [22]. This industry requires faster GPUs for the ever growing realism and complexity of games. As a result the GPU market also evolves rapidly in order to keep up with the game-industry demands.

Graphics card manufacturers present the hardware functionality, available to the software developer, by means of a *shader model*. This model specifies the operation of the hardware incorporated techniques and how to apply them in a software application. Fast developments make it hard to grasp all new techniques and their durability. Therefore the new techniques are presented in the form of extensions to the model, which eventually might, or might not, be adopted in a new shader model specification. These extensions can be subject to lots of changes. Nevertheless they form a hotbed for future techniques.

For this project we have adopted several extensions that are very likely to be part of the upcoming specification. In this way, we can present new approaches that were not feasible with previous graphics cards. The remainder of this paragraph summarises the most important extensions which are used in our framework.

Vertex Texel Fetch (June 2004)
Formerly processing of texture elements was reserved for the fragment shader. Texturing is part of the last stage of the graphics pipeline and therefore accessing texture elements was only allowed in the fragment shader. Nowadays however there are good reasons to also allow the vertex shader to access textures. This is called a *Vertex Texture Fetch*.

Buffer Objects
A buffer object is conceptually a coherent memory block in the GPU memory. Initially these memory blocks were accessible only to the GPU. With the trend of increasing programmability on the GPU, these memory locations have become available to the developer as well. Several types of buffer objects are available serving different objectives.

For instance a buffer object can be used to store an amount of vertices. Such a memory block is called a *Vertex Buffer Object* (VBO). In the OpenGL 'immediate' mode we supply the driver with one vertex at a time, resulting in a large amount of CPU to GPU transfers. Using a VBO however allows a precomputed set of vertices to be stored in GPU memory, which results in faster rendering times. In a way one can think of the VBO as the successor of the display list. Both VBOs and display lists reside on the GPU memory, but the VBO provides a more profound way to control the memory.

Another possibility is to make the GPU access a buffer object as a texture. Such a memory block is called a *Texture Buffer Object* (TBO). It was possible to create and modify textures already before the existence of texture buffer objects. However considering the texture as being a buffer can be convenient. For instance when a 'render-to-texture' approach renders a colour buffer to a memory block, it can be considered a texture using the TBO extension. This texture can directly be used for the next rendering pass.

The 'render-to-texture' approach is a form of offscreen rendering, which can be implemented through the *Frame Buffer Object* (FBO). This object replaces the default GL framebuffer, keeping track of the colour and depth buffer for each frame.

A buffer object extension which is not used in our framework is the *Pixel Buffer Object* (PBO). The PBO permits the use of buffer object for pixel data. In a way these objects are

similar to the way a VBO treats vertices. Again the goal is to accelerate pixel operation, by directly accessing the pixel data GPU-side.

Object	Extension	Date
Vertex Buffer Object	GL_ARB_vertex_buffer_object	January 2003
Texture Buffer Object	GL_EXT_texture_buffer_object	November 2006
Frame Buffer Object	GL_EXT_framebuffer_object	April 2006
Pixel Buffer Obect	GL_EXT_pixel_buffer_object	December 2004

Table 2.2: Buffer object extensions

Transform Feedback (October 2006)
One of the most important extensions applied in our framework is the *Transform Feedback* extension, also called *Stream-Out* in DirectX terminology. Transform feedback is a new method that records vertex attributes into a buffer object, before actually rendering the current frame. As a result the rasterisation stage of the graphics pipeline can be discarded and the obtained buffer object can serve as an input for the next rendering pass. The transform feedback can be employed in either the vertex or geometry shading stage and can record the vertices of processed primitives either interleaved in one buffer object or into two separate buffer objects.

Geometry Shader (January 2007)
Another major new extension is the *Geometry Shader*. The geometry shader does not actually shade anything, instead it provides programmability of the primitive assembly stage of the graphics pipelines. Figure 2.6 shows that the geometry shading stage is executed after the vertices are transformed. The geometry shader accepts the vertices of primitives such as points, lines or triangles. Subsequently these vertices can be transformed, changing the geometric object. For example a geometry shader can accept a point primitive, which is returned as a line primitive by adding one or more vertices. Not only can the geometry shader create new geometry, it is also capable of destroying geometry. This property turns out to be particularly useful, since it provides new opportunities for general purpose computations on the GPU. This subject will be elaborated in the next paragraph.

2.5 General-purpose computation on GPU

Increasing programmability and arithmetic precisions provoke a growing interest in the GPU as a platform for general purpose computations. A new research field investigating the abilities of GPU for general purpose computations originated and is named *General Purpose Computing on GPU* (GPGPU). Using the GPU for computational expensive operations potentially yields a considerable performance gain. However the approach differs from traditional CPU computations, since the hardware is directed to graphics processing. The remainder of this paragraph will introduce the operations that are currently available on the GPU and the basic techniques that stem from GPGPU field. Lastly this paragraph will provide an overview of the analogies between the GPU and the CPU when considering arithmetic computations.

2.5.1 Available operations & techniques

In the early days of GPU programmability, the shader program needed to be specified in a *GPU assembly language*. Besides default assembly instructions, like additions and multipliers, this language also provides specific graphics related instructions. For instance there is a native call for the dot product of vectors, the reciprocal quotient and a multiply-accumulate instruction. These are all significant instructions for graphics processing. Implementing general purpose computations involves adjusting the arithmetic problem to use these instructions. Using these native instruction will improve the performance of the computations.

Implementations in an assembly language is usually a profound and time-consuming job. Fortunately there are *higher level languages* available nowadays, abstracting away from the assembly instructions. Such languages offer a complete set of functions, optimised for the GPU. One can think of common functions, such as calculating a minimum or maximum. But also there are geometric, exponential, trigonometrical, vector relational and matrix functions available. These functions, together with sufficient precision, form the fundament for general purpose computations on the GPU.

The mentioned operations alone do not suffice to perform any real-world arithmetic task. The graphics pipeline requires a stream of input to process and some buffer to return the computation results. Dealing with the *input and output* in terms of the graphics pipeline forces some constraints to the data structures and hence to the computation itself.

Computations can be executed in one of the programmable stages of the GPU pipeline. Reading of an input stream in one of these stages is called *gathering* of data. All shader stages are able to gather values from an buffer, which usually comes in the form of a texture. Gathering input data is similar to reading a value from an array, denoted as:

$$gather : p = a[i]$$

Similar we require the graphics pipeline to output the data, which is referred to as the *scattering* of data. In the starting days of the GPU existence, the only stage that was able to do any form of scattering was the vertex shading stage. The vertex shader is able to transform the positions of the vertices, changing the position in the output buffer. The flexibility of the scatter operation greatly improved by the introduction of the geometry shader. Not only can vertices be displaced, vertices can also be created or destroyed, providing the developer with more control of the output data. Scattering the data is similar to writing a value to an array, denoted as:

$$scatter : a[i] = p$$

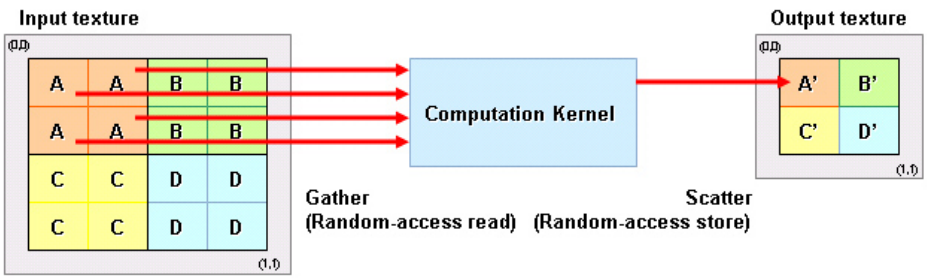


Figure 2.9: Gather, compute and scatter approach

Several *higher level techniques* have been defined, based on the gather, compute and scatter approach. This approach is schematically depicted in figure 2.9. The figure shows that a computational kernel, executed by a shading stage, is able to read values from the input texture at an arbitrary position, Furthermore the kernel might be able to store a value to a texture, dependent on the current shading stage. In this example the A values are gathered and taken as input for some computation. The result of that computation, here labelled by A', is scattered to the output texture. Similarly this holds for the values B, C and D.

The most obvious technique, based on this approach, is called a *mapping*, simply applying the same operation to each value. The pipeline is given some input buffer for which all values get processed and written to a separate output buffer. A slightly more complicated technique is the *reduction*, which diminishes a stream in a given number of passes. For instance two values can be gathered from the input stream, scattering only one value after a multiplication or summation. This divides the buffer size in halve, which can be continued until only one value remains. In case of the addition operation this results in the global sum of the initial buffer. Program 1 presents a reduction shader in pseudo code for one-dimensional input and output textures. Please note that the output buffer is defined before starting the graphics pipeline, recording the scattered output values.

```
buffer input;

main()
{
  texel     = gather(buffer, texelPos);
  neighbour = gather(buffer, neighbourPos);

  sum = texel + neighbour;

  scatter(sum);
}
```

Program 1: Pseudo code: Reduction shader

Some other high-level techniques include *searching* and *sorting* approaches, suitable for the GPU. Both techniques were implemented for our framework and will be discussed extensively in the designated sections describing the hardware rendered framework.

2.5.2 CPU-GPU analogies

The previous section discussed the global GPGPU approach, comparing the graphics pipeline to a CPU based computation. This section deepens this conception of resemblance, by listing analogies between the CPU and the GPU. These analogies are particularly helpful when reasoning about software-rendered algorithms that need to be ported to the GPU, which is exactly what we expect for the algorithms of the VolumeFlies framework.

Input = Texture / Buffer Objects

Usually a computation either takes a stream or a data structure for an input. For the GPU however the data structure should be converted into either a texture or texture buffer object. Implementing more complex structures like lists, trees or hash tables on the GPU is a non-trivial task. Typically computations have to cope with a simple array-like data input. Reading from the arrays is similar to a texture or buffer fetch on the GPU.

Computation = Rendering

For a given calculation the kernel, which can either be a loop body or algorithmic step, is performed on the GPU by the shader stage. The actual computation of such kernel is then executed by initiating a rendering pass, triggering the shader stages. A vertex shader gets triggered for each vertex that is being rendered, whereas a geometry shader gets triggered for each primitive. Similarly the fragment shader gets triggered for each fragment.

Output = Texture / Buffer Objects

The final step is to output the computation results to a separate buffer. Traditionally a render-to-texture approach is performed to feedback the output values. Such an approach involves rendering a full-screen quad, for which each pixel represents one of the output values. This method requires a precise implementation, taking care of correct dimensions and interpolation settings. A new approach was found in the transform feedback, presented in section 2.4.4. This method provides a more flexible way to output values into a texture buffer after the vertex or geometry shading stage.

Chapter 3

Project requirements

The previous chapters introduced the main concepts involved in this project. The line of thought is based on the illustrative rendering framework called 'VolumeFlies'. This framework was introduced by S. Busking in his master's thesis [5] and proposes a 'smart particle' approach to realise an IVR environment. The main task for this continuation project was to accelerate the 'VolumeFlies' framework, by exploiting the capabilities of modern day graphics hardware. This involves a major rewrite of the algorithms, fitting them to the graphics pipeline. Despite the notable difference from an implementation point of view, the fundamental approach remains.

A set of requirements is needed, in order to describe what the system is ultimately supposed to do. This project is concerned with the implementation of an ad hoc research application, for which user requirements are most often loosely defined. In contrast to commercial software there is no direct customer or end user. For that reason the user requirements are not explicitly specified. Instead the *functional requirements* are based on the project description and the preliminary framework design. The grouped functional requirements are listed in Appendix A.

Besides the functional requirements, there is a set *extra-functional requirements*. Although the functional requirements describe *what* the software will do, the extra-functional requirements describe *how* the system will actually perform its task. There is a multitude of extra-functional requirements that can be assigned to the framework. To oppose supernumerary, only the most important extra-functional requirements are listed in Appendix A.

In general the software engineering process takes these requirements as a starting point to start building the software architecture. Requirements are transformed into specifications, which are the basis for an architecture. This process is executed iteratively, evaluating the architecture with the requirements. For this project however the architecture of the preliminary framework is available, which forms a decent reference for the gpu-driven framework. The existing architecture offers a modular approach to easily configure the desired illustrative pipeline. The architecture is extended with several classes offering an interface to the graphics hardware. The software architecture can be found in Appendix B.

The presented architecture evolves around the 'Volume Mapper'. This mapper allows the user of the framework to set up a pipeline of modules, that together form the illustrative visualisation of the volume. In order to clarify the approach, a generalised depiction is given in figure 3.1. The figure shows on a conceptual level how the framework connects several modules, resulting in a visualisation pipeline.

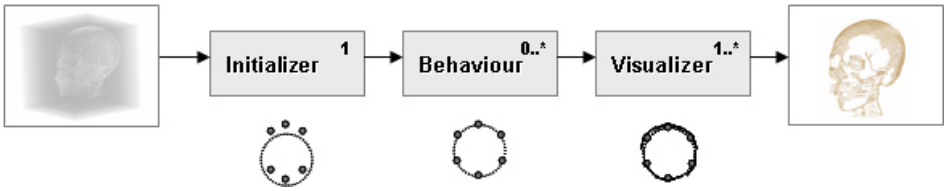


Figure 3.1: Framework

Since we have a volume rendering approach, the volume data is the main input to the system. The first step in the system is the *'Initialiser'*. The initialiser is responsible for the creation of the initial particles. In particle system terminology this is called the particle *birth*. This is schematically depicted in the figure, where the dashed line represents the feature of interest and the small dots represent the newly originated particles. The particle set can be transformed by a set of *'Behaviours'*. Usually this transformation comprises a displacement of the particles, in order to improve the particle placement regarding the feature. However a behaviour might as well adjust any other particle parameter. In figure 3.1 the schematical depiction shows that the behaviour places the particles onto the feature. Finally one or more *'Visualisers'* perform the actual illustrative rendering, based on the current state of the particles set. The schematic depictions shows a simplified hatching approach, where lines are drawn along the particles.

The next chapter will address the software-based implementation of the modules of the VolumeFlies framework. Subsequently chapter 5 will present the same modules, describing the new hardware-based implementation.

Chapter 4

Software rendered VolumeFlies

Recall from the previous chapter that the proposed framework exists of a number of modules, that together create the graphics pipeline for an illustrative visualisation. This chapter addresses the software implementation of these modules. In general, the complexity of the algorithms in these modules is in the order of the number of particles ($\mathcal{O}(m)$). The reader should be aware that this is merely a brief recapitulation of the work presented by S. Busking [5] [6], which is needed as a starting point to discuss the new hardware implementation. This hardware-based approach will be discussed in chapter 5.

4.1 Initialisation

Figure 3.1 shows that the framework requires exactly one initialisation module, which is responsible for the initial particle placement. The particles should be placed somewhere on the desired features in the volume. These features form a whole of various aspects of the volume. Be aware that they do not necessarily consist of a single object. Several approaches were considered in order to locate the desired features. Inspired by the 'smart particle' concept, presented by Pang and Smith [28], Busking's initial idea was to originate particles from a fixed position. The particles should be 'smart' enough to move towards interesting features in the volume. The major drawback is that such an approach comes with a reasonable chance to overlook small objects inside the volume. This is due to the fact that the particles have only local sight in the volume. A more rigorous approach was required to detect the desired features.

4.1.1 Brute-force particle placement

A typical feature the particles should be heading for in the volume is the *iso-surface*. An iso-surface at iso-value v consists of those points for which $f(x) = v$. Here $f(x)$ is the scalar value associate with point x in the volume. The proposed feature location method is inspired by the marching cubes approach, initially proposed by Lorensen and Cline in 1987 [23]. Marching cubes proceeds through the scalar field of the volume, considering six neighbouring voxels at a time. Within this 'cube', the polygons that represent the local part of the surface are determined.

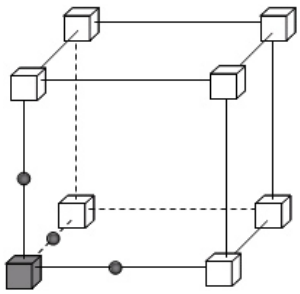


Figure 4.1: Marching particle initialisation approach

Marching cubes traverses through the adjacent cubes, that potentially contain a part of the iso-surface. Busking's brute-force initialisation however, addresses a 'cube' for every voxel, which guarantees that even the smallest objects will be found. Instead of determining the actual part of the surface, a new particle is created near the iso-surface. This method is schematically depicted in figure 4.1. Assume that the desired iso-value v, resides between the gray voxel and the neighbouring white voxels. Initially the particles are placed halfway between the voxels, resulting in a rectilinear grid of particles, near the iso-surface. A redistribution step will resolve this drawback. A summary of the advantages and disadvantages of the proposed method is presented in table 4.1.

Complexity: $\mathcal{O}(n)$ with n = number of voxels
Advantages: + Guaranteed feature location
Disadvantages: - Volume resolution dependent - Artificial rectilinear placement

Table 4.1: Initialiser overview

4.2 Behaviours

The modular approach, presented in figure 3.1, allows a number of behaviours, adjusting the particle set. A proposed behaviour comprises a redistribution of the particles, moving them onto the iso-surface with an optimal distribution.

4.2.1 Particle redistribution

Busking's redistribution of the particles relies on a repulsion scheme, inspired by the work of Meyer et al. [25]. Particles repel nearby particles to minimise an energy function. This function has to be chosen carefully in order to overcome unevenly distributed particle sets. Meyer et al. define three requirements for an energy function:

- Energy functions should be continuous
- Energy functions should be compact
- Energy functions should be scale invariant

Continuity of the function results in a smooth function, which is always defined for the necessary first order derivative. An energy function is scale invariant, whenever the ratio of energies between particles, is independent of choice of units in the system. Furthermore, compactness of an energy function bounds the domain of the function, avoiding global influences. As a result, the function can be applied locally, which is computationally more efficient.

The energy at particle position p_i, is the sum of energy potentials of the neighbouring particles p_j. These neighbours reside within a certain radius from p_i, exerting a repulsion force on the particle at position p_i. This energy is a function of the euclidean distance $|\vec{r}_{ij}|$ between p_i and a local neighbour p_j. For a set of m particles, the energy E_i at particle p_i is defined as:

$$\vec{r}_{ij} = p_i - p_j \tag{4.1}$$

$$E_i = \sum_{j=1, j\neq i}^{m} E_{ij}(|\vec{r}_{ij}|) \tag{4.2}$$

The total system energy E is the sum of the individual particle energies.

$$E = \sum_{j=1}^{m} \sum_{i=1, i\neq j}^{m} E_{ij}(|\vec{r}_{ij}|) \tag{4.3}$$

The particles are expected to move to a locally lower energy state. This requires a displacement direction v_i, that moves the particle p_i into the direction that minimises the energy function. Therefore particles are moved along an *energy gradient*. This gradient points in the direction of the greatest rate of increase in the energy field. A steepest descent on the energy can be applied, in order to move a particle to a lower energy state. If we define the particle position to be $p_i = (x_i, y_i, z_i)$, then the displacement vector v_i is denoted by:

$$\vec{v}_i = -(\frac{\partial E_i}{\partial x_i}, \frac{\partial E_i}{\partial y_i}, \frac{\partial E_i}{\partial z_i}) = -\sum_{j=1, j\neq i}^{m} \frac{\partial E_{ij}(|\vec{r}_{ij}|)}{\partial |\vec{r}_{ij}|} \frac{\vec{r}_{ij}}{|\vec{r}_{ij}|} \tag{4.4}$$

The energy function E_{ij}, proposed by Meyer et al. [25], is a modified cotangent with a single free parameter σ. This free parameter defines the maximal radius of repulsion with neighbouring particles. The function nicely satisfies the requirements listed at the start of this section. However, the energy function can be replaced by any other energy function with similar characteristics.

$$E_{ij}(|\vec{r}_{ij}|) = \begin{cases} cot(\frac{|\vec{r}_{ij}|}{\sigma}\frac{\pi}{2}) + \frac{|\vec{r}_{ij}|}{\sigma}\frac{\pi}{2} + \frac{\pi}{2} & \text{, if } |\vec{r}_{ij}| \leq \sigma \\ 0 & \text{, if } |\vec{r}_{ij}| > \sigma \end{cases} \tag{4.5}$$

Following the approach by Meyer et al., VolumeFlies adopts a two-step particle update scheme. In the first step of the update scheme, each particle is updated in the local *tangent plane*. This is the plane perpendicular to the *gradient direction* $\vec{g}_i$ in the volume f, at the particle position p_i. A tangent plane is schematically depicted by square planes touching the surfaces in figure 4.2. Be aware that we assume column vectors by default; denoted by the transpose (T) of the row vector in the following equation.

$$\vec{g}_i = \nabla f(p_i) = (\frac{\partial f}{\partial x}(p_i), \frac{\partial f}{\partial y}(p_i), \frac{\partial f}{\partial z}(p_i))^T \tag{4.6}$$

The normal vector n_i is the vector perpendicular to the surface at particle position p_i. Assuming that the scalar values of f increase while moving further into the feature, the outward pointing normal vector n_i at position p_i is defined as:

$$\vec{n}_i = -\frac{\vec{g}_i}{|\vec{g}_i|} \tag{4.7}$$

The two-step update scheme, proposed by Meyer et al. [25], is depicted in figure 4.2 and defined as:

$$\text{Step 1: } p_i \leftarrow p_i + (I - \vec{n}_i \cdot \vec{n}_i^T)\vec{v}_i \tag{4.8}$$

$$\vec{g}_i \leftarrow \nabla f(p_i) \tag{4.9}$$

$$\text{Step 2: } \quad p_i \leftarrow p_i - f(p_i)\frac{\vec{g}_i}{|\vec{g}_i|^2} \tag{4.10}$$

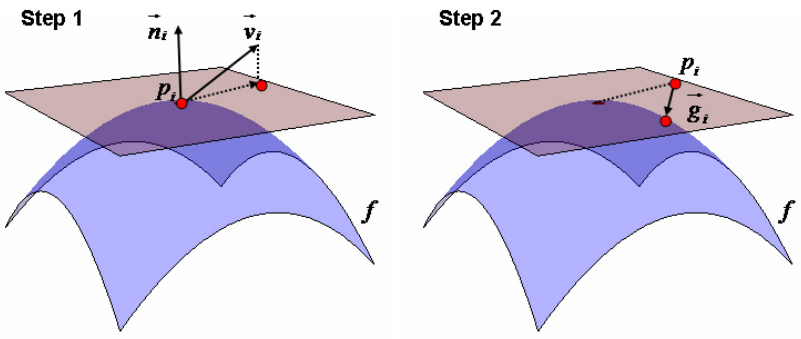


Figure 4.2: Two-step repulsion scheme

The first step, updates the particle position p_i in the local tangent plane. The displacement direction $\vec{v}_i$ is projected onto the tangent plane, using the projection matrix $I - \vec{n}_i \cdot \vec{n}_i^T$, where I is the 3x3 indentity matrix. As a result of the first displacement, the particle is likely to be pushed away from the surface. Therefore, the second step in the update scheme, reprojects the particle back towards the surface, along the local gradient $\vec{g}_i$. This gradient $\vec{g}_i$ is measured at the new position p_i, resulting from the first particle displacement (equation. 4.8). The reprojection comprises one step of a Newton-Raphson approximation, moving the particle back to the iso-surface. A description of the Newton-Rhapson method can be found in the book 'Numerical methods that work' [1].

This two-step update scheme is executed for each particle. In VolumeFlies, the entire process of updating all particle positions is repeated for a fixed number of iterations. In the work of Meyer et al. a different stop criterion based on the difference of the global energy between subsequent steps, is used.

Unfortunately, this energy minimisation approach has a few disadvantages. First of all the particle displacement is constrained with a maximal step size. This step size is a parameter δ, that needs to be tuned carefully by the user of VolumeFlies. A small step size results in slow convergence of the system, whereas a large step size results in a particle set jumping around the equilibrium. The system is constrained as follows:

$$\vec{v}_i = \begin{cases} \vec{v}_i & |\vec{v}_i| \leq \delta \\ \delta\dfrac{\vec{v}_i}{|\vec{v}_i|} & |\vec{v}_i| > \delta \end{cases} \tag{4.11}$$

The second sensitive parameter is the repulsion radius σ. For a larger radius more neighbouring particles will be considered, resulting in substantial increase in the computational effort. On the other hand, when the radius becomes too small no neighbours will be detected, as a result of which there will be no repulsion at all.

Both the step size and the radius affect the speed of convergence. But even with a good tuning of these parameters the overall performance of the approach is rather slow. In a new paper Meyer et al. [26] propose some computational optimisations. First of all they introduce a *binning structure*, reducing the environment of interactions to a smaller subspace. This has been incorporated in VolumeFlies. Furthermore, Meyer et al. propose to keep track of the neighbours of each particle. The list of neighbours helps to quickly locate the nearby particles and increases the overall performance. This is due to the fact that the list does not need to be updated after every iteration. An overview of the redistribution is given in table 4.2.

Complexity: Per iteration: $\mathcal{O}(mb)$ with b = average number of neighbours
Advantages: + Evenly distribution of particles + Applies to complex implicit surfaces
Disadvantages: - Sensitive user parameter: particle displacement step size - Sensitive user parameter: repulsion radius σ - Computational expensive

Table 4.2: Redistribution overview

4.3 Visualisation

The final step in the illustrative framework, depicted in figure 3.1, is the actual stylised visualisation. At this point the particle system must have been initialised and possibly have been transformed by a number of behaviours. This section describes the method of working for all illustrative styles, included in the VolumeFlies framework.

4.3.1 Cone splatting

To some extend the cone splatting module, listed in this chapter, is an exception to the other modules. Not only does the module serve as a visualiser in the illustrative framework; moreover can it be applied as a filter. In that case the module will only show the particles that reside near the visible side of the implicit surface of a feature in the volume. Firstly the cone splatting approach will be described, before addressing the filtering application.

In section 2.1 the concept of surface splatting was introduced. The usual approach to surface splatting is to situate, orient and scale a flat disc, at evenly spaced positions on the surface. In the context of surface splatting, such a disk is called a splat, which replaces the concept of an accumulated footprint, as seen in the volume splatting approach. With good parametrisation, the splats create a closed surface.

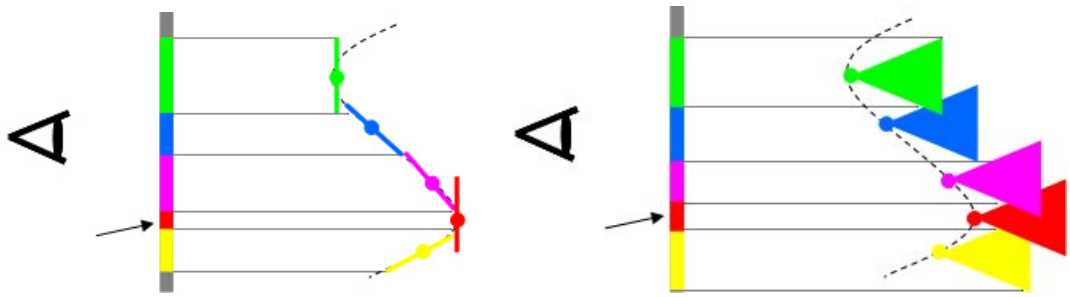
In VolumeFlies a *surface splatting* approach was adopted, with the evenly distributed particles as a starting point. In contrast to the usual approach, S. Busking proposed to render cone splats instead of discs. He observed that discs might overlap, hiding a number of particles that should have been visible. The idea is to render oriented cones in the direction of the viewpoint, at each particle position. Figure 4.3 depicts both surface splatting approaches and shows that the use of oriented cones prevents the occlusion of particles.

When using cones, overlapping problems occur at the edges of the feature, which can be resolved by scaling the cones. This scaling depends on the normal vector $\vec{n}_i$ (equation 4.7), the normalised viewing vector $\vec{e}_i$ and the maximal cone radius r. The viewing vector $\vec{e}_i$ points from the particle position to the viewpoint. At present, the cone-splatting requires an orthographics projection. As a result, all viewing vectors $\vec{e}_i$, starting at the particle position p_i, comprise the same unit-length vector, perpendicular to the viewing plane. The directions for anisotropic scaling of the cones are defined as:

$$\vec{r}_i = r \cdot \frac{\vec{n}_i \times \vec{e}_i}{|\vec{n}_i \times \vec{e}_i|} \tag{4.12}$$

$$\vec{r'}_i = r(\vec{n}_i \vec{e}_i) \frac{\vec{r}_i \times \vec{e}_i}{|\vec{r}_i \times \vec{e}_i|} \tag{4.13}$$

A detailed motivation for the use of cone splats, and the deduction of the scaling can be found in the master's thesis describing VolumeFlies [5].



(a) Oriented disc might overlap. The particle at the arrow is nearly occluded.

(b) Oriented cones prevent the occlusion problem.

Figure 4.3: Cone splatting schematic (based on VolumeFlies master's thesis [5])

In this approach, there are two parameters involved that need to be selected carefully. Firstly there is the radius of the cone. Currently this radius has to be manually adjusted by the user. The radius should be large enough to close the surface, but not larger than the distance between the particles. The second parameter is the depth of the cones. When the cones are too deep, this might cause surfaces, closely behind the front surface, to bleed through. Fortunately surfaces in medical data are usually not very close together.

The surface, created with the cone splatting approach, is used as the starting point to apply visible surface calculation (VSC). The first step in this *filtering* approach, is to render the front-faces of cones to an off-screen colour-buffer, assigning to each cone a unique colour. Subsequently, this colour-buffer can be scanned for each particle, verifying if the associated cone colour occurs. A particle is hidden by a surface nearer to the viewport, whenever the colour of the associated cone does not occur in the colour-buffer.

A disadvantage of this approach is the unstable nature of the particles near the feature edges. Because of the anisotropic scaling, one of the scaling directions might become zero. The cone will no longer be visible, and hence cannot be detected in the colour-buffer during scanning. Furthermore, if the features in a volume contains parts of varying sizes, it becomes hard to choose a good cone radius. Usually this results in a radius, which is too large for the finer parts of the features, resulting in overlapping problems. The overview for this module is presented in table 4.3.

Complexity: surface splatting $\mathcal{O}(m)$; filtering $\mathcal{O}(m * k)$ with k = number of pixels
Advantages: + Fast surface representation + Visible surface calculation
Disadvantages: - Sensitive user parameter: cone splat radius - Sensitive user parameter: cone splat depth - Variety in size of parts of the feature leads to overlapping problems - Unstable feature edges (whenever a radius direction turns to zero) - Image space complexity; hence resolution dependent

Table 4.3: Cone splatting overview

4.3.2 Stippling

Successively applying the initialisation, redistribution, and the cone-splat filter results in an evenly distributed set of particles on the visible side of the surface. These particles will be used to engage several illustrative styles. The first style that will be addressed is called stippling, which comprises a point representation of the particles. This concept was introduced in section 2.2, and is particularly useful to provide a context to a visualisation.

VolumeFlies includes two different stippling approaches, both accentuating the contours of the features in the volume. For the first approach the density of the particles is altered, creating a difference in tonality of the resulting image. This *density-based stippling* technique is typically used in traditional illustrations. In order to apply this technique to the particle set, we assume that the density of the redistributed particle set is dense enough to represent the darkest tone.

Adjusting the particle density, is achieved by using a basic diffuse lighting, where the light source is situated in the position of the camera. Be aware that the particles do not move, but are either visible or invisible, based on the local lighting. The lighting equation is defined as:

$$L_d(i) = max(\vec{n}_i \cdot \vec{e}_i, 0) \tag{4.14}$$

In order to imitate hand-made traditional illustrations, randomness is introduced in the stippling model. The set of particle properties is extended with a value v_i, which is obtained from a uniform random distribution, ranging between 0 and 1. The random value v_i, associated with the particle, is compared with the brightness $L_d(i)$:

$$v_i > (L_d(i))^c \tag{4.15}$$

The power parameter c is user configurable. Higher values of c allow more visible particles, resulting in more dark areas. This holds only when both the normal vector $\vec{n}_i$ and the viewing vector $\vec{e}_i$ are of unit-length. Furthermore, the power parameter c should always be greater than zero. Next to the power parameter c, the model includes the user parameter $t \in [0, 1]$. Whenever the brightness $L_d(i)$ at a particle position is above a this threshold t, the particle will not be shown. This ensures that no particles will be visible in the bright areas. An overview of the density-based stippling approach is given in table 4.4.

Complexity: $\mathcal{O}(m)$
Advantages: + Stippling applied similar to traditional illustrations + Suitable for rendering of contexts
Disadvantages: - Initial particle set needs to be dense

Table 4.4: Density-based stippling overview

Alternatively, stippling can be applied by scaling the size of the point representation of the particles. This representation comprises a point primitive, for which the size can be changed in image-space. The *scale-based stippling* approach scales these point primitives, based on lighting equation 4.14. Areas with larger point primitives will result in darker areas.

The size of the point representation is based on the optimal packing of particles; originating from the particle redistribution. The size s_i is deduced in the VolumeFlies master's thesis [5], and is defined as:

$$s_i = \sigma(\frac{\sqrt{3}}{2\pi}(1 - L_d(i)))^c \tag{4.16}$$

By default the parameter σ is set to the average distance between the evenly distributed particles. By putting this parameter under user control, the maximum size of the point primitives can be adjusted. As a consequence, the brightness of the resulting image can be changed by the user. The power parameter c, which is also configurable by the user, is similar to the power paremeter in the density-based stippling approach. Adjusting c changes the contrast between light and dark areas. This parameter should be greater than zero and is

typically set to 0.5. This follows from the deduction presented in the VolumeFlies master's thesis [5]. Lastly, also the scale-based stippling approach is subject to a threshold parameter t, removing all particles in the areas where the brightness $L_d(i)$ ends up above the given threshold. An overview of the scale-based stippling approach is given in table 4.5.

Complexity: $\mathcal{O}(m)$
Advantages: + Good results with average number of particles + Suitable for rendering of contexts
Disadvantages: - Point representations might overlap the actual feature edge

Table 4.5: Scale-based stippling overview

Recall that a particle is merely an object, that behaves based on a given set of properties. Throughout the sections in this chapter, several properties where assigned to the particles. For clarity, these properties are listed in table 4.6.

Properties for each particle
• p_i: particle position
• $\vec{n}_i$: normal vector at the particle position
• $\vec{e}_i$: viewing vector from particle to view-point
• v_i: random value between 0 and 1

Table 4.6: Particle properties

4.3.3 Hatching

The second style, included in the VolumeFlies framework, is called hatching. Hatching was briefly introducted in section 2.2, and resembles a pen and ink based drawing style. Traditionally, the subject within a hatched image is represented by sensibly placed pen strokes. These pen strokes, also known as hatches, are used to shade the image. The density of the hatches is used to vary the tone in the resulting image, similar to the density-based stippling approach. Furthermore, the trace-direction of each pen stroke can be used to highlight, for instance, curved areas. The two different hatching approaches, included in VolumeFlies, will be described in this section. Again the evenly distributed and filtered particle set is the starting point to employ this illustrative style.

Direction-based hatching

The direction based hatching approach, consists of two steps. First the hatches, which follow a fixed direction over the implicit surface, are generated and stored to a buffer. Subsequently, the generated hatches are rendered to screen.

Step 1: Tracing the hatches

The hatches are traced, commencing in a fixed direction. A directed segment of the hatch is created, starting at the particle position. Subsequently the hatch is projected onto the local tangent plane. Successively adding segments in the given direction results in a set of strokes that follow the feature in a single direction. Be aware that the segments are not reprojected to the surface, which yields a small error. In practice however this error is hardly noticeable.

This approach was extended to include *cross-hatches*. Besides the fixed hatching direction, the hatch will also be traced, in the direction perpendicular to the fixed initial direction. In that case, the hatch strokes cross at the particle position. The overview for the direction-based hatching approach can be found in table 4.7.

Step 2: Visualise the generated hatches

When the hatches are generated, they can be rendered to screen. During the visualisation step the appearance of the hatches can be changed. For instance the line thickness and colour can be adjusted.

Complexity: $\mathcal{O}(d * m * h)$ with h = number of hatch segments, d = number of hatch directions
Advantages: + Hatching applied similar to traditional illustrations + Suitable for rendering of contexts
Disadvantages: - Sensitive user parameter: Step size - No reprojection to the surface yields tracing error - Hatches do not trace the feature curvature

Table 4.7: Direction-based hatching

Curvature-based hatching

Furthermore, VolumeFlies includes a curvature-based hatching approach. For this approach the trace-direction of the hatch follows the curvature of the feature. In order to do so, the principal curvature needs to be determined at each new position of the hatch trace. The three-step approach, presented by Kindlmann et al. [17], is adopted to measure and compute the principal curvature values and directions. This approach to principal curvature computation is introduced in interlude 1, and comprises prerequisite material for the hardware-based curvature estimation.

"Curvature describes the variation of the normal vector, when moving a small distance from the current position. Since there is a whole angular range of directions, there is also a range of corresponding *normal curvatures*. These curvatures have a well-defined minimum and maximum, orthogonal to each other. These curvatures are called the *principal curvatures*." (Condensed definition as described in 'Real-Time Volume Graphics' [13], page 368)

The variation of the normal vector n is defined as (Kindlmann et al. [17]):

$$\nabla \vec{n}^T = -\frac{1}{|\vec{g}|} PH$$

First of all, the first and second partial derivatives, comprising the gradient $\vec{g}$ and the Hessian matrix H, need to be measured.

$$\vec{g} = \nabla f = [\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}]^T$$

$$H = \begin{bmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial x \partial z} & \frac{\partial^2 f}{\partial y \partial z} & \frac{\partial^2 f}{\partial z^2} \end{bmatrix}$$

The normal $\vec{n}$ and the projection matrix P, on a plane with normal $\vec{n}$, can be calculated.

$\vec{n} = -\frac{\vec{g}}{|\vec{g}|}$; $P = I - nn^T$, where I is the identity matrix.

The principal curvature can be found through eigen-analysis of the shape operator S:

$$S = \nabla \vec{n}^T P = -\frac{PHP}{|\vec{g}|}$$

The steps needed to compute the curvature at an arbitrary point in the scalar field:
(Based on work by Kindlmann et al. [17])

1. Measure first partial derivatives ($\vec{g}$) and compute $\vec{n}$ and P
2. Measure second partial derivatives (H) and compute S
3. Compute the trace T and Frobenius norm F ($\sqrt{trace(SS^T)}$) of S.
   The curvature values κ_1 and κ_2 can then be computed:

   $$\kappa_1 = \frac{T + \sqrt{2F^2 - T^2}}{2} \text{ and } \kappa_2 = \frac{T - \sqrt{2F^2 - T^2}}{2}$$

   The curvature directions $\vec{k}_1$ and $\vec{k}_2$ can now be found by solving the eigen-system:
   $S\vec{k}_1 = \kappa_1 \vec{k}_1$
   $S\vec{k}_2 = \kappa_2 \vec{k}_2$

Interlude 1: Principal curvature

Using the method, presented in interlude 1, it is possible to estimate the curvature in each position of the scalar field. In the hatching approach, this is used to determine the trace-direction for each segment. However, simply starting the hatch in one of the principal curvature directions, and subsequently tracing along that principal curvature direction, yield messy results. This is resolved by smoothing the direction field, obtained from the principal curvature directions. After this smoothing step, the hatch will be traced along the surface curvature.

Step 1: Smoothing the direction field

The smoothing of this direction field is precomputed in two steps. The first step comprises a smoothing of the starting directions, at the particle positions. The directions are smoothed, based on the local reliability of the surface. This *field reliability* $\rho \in [0, 1]$, defines how suitable the surface shape is for hatching. This measure is based on the *shape index* $s \in [-1, 1]$, which indicates the shape of the local surface. This shape index was defined by Koenderink and Van Doorn [19].

$$s = \frac{2}{\pi} arctan \frac{\kappa_2 + \kappa_1}{\kappa_2 - \kappa_1} \quad (\kappa_1 \geq \kappa_2 \text{ and } |\kappa_1| + |\kappa_2| > 0) \tag{4.17}$$

$$\rho(\kappa_1, \kappa_2) = \begin{cases} 0 & \text{if } |\kappa_1| < \epsilon \text{ and } |\kappa_2| < \epsilon \\ 1 - |2(|s| - \frac{1}{2})| & \text{otherwise} \end{cases} \tag{4.18}$$

Here κ_1 and κ_2 are the principal curvature magnitudes. In order to determine the reliability of the surface, it is important to know if a surface is flat or nearly flat. The surface is considered flat, and hence unreliable, if both principal curvature magnitudes are less that a small constant ϵ. At each particle position, the principal curvature directions in the local neighbourhood are averaged, using the reliability ρ as a weight.

Step 2: Tracing the hatches

Secondly the hatch needs to be traced, starting in the smoothed field direction. In order to 'guide' the hatch in the direction of the smoothed field, the directions of the local neighbourhood are again averaged, but now for each segment. The distance to the particle under considerations is applied as a weight. Also this is typically a step that needs to be precalculated.

Generating the hatches is computationally expensive, since the local neighbourhood needs to be addressed many times. After the hatches are generated, the visualisation is interactive. An overview of the curvature-based approach is given in table 4.8.

Step 3: Visualise the generated hatches

The visualisation step is identical to the step in the direction-based hatching approach. Again the generated hatches are rendered to screen, possibly changing the appearance of the hatches.

Furthermore, basic diffuse lighting (Equation 4.14) is applied to the hatches, similar to the density-based stippling approach. Whenever a particle is marked invisible, based on equation

4.15, the entire hatch will be discarded. The density approach delivers better results for the hatches, since the initial set of hatches appears denser, compared to a point representation.

The lighting can now be extended to a two-level threshold. These thresholds determine, whether there should be either a single, crossed or no hatch at the position of the particle. This results in a shading where the density of the hatches successively changes from bright to dark in three levels, based on the local illumination.

Complexity: $\mathcal{O}(d * l^2 * m * h)$ with h = # hatch segments, d = # hatch directions, l = average # neighbours
Advantages: + Hatching applied similar to traditional illustrations + Suitable for rendering of contexts + Accentuates highly curved areas
Disadvantages: - Sensitive user parameter: Step size - No reprojection to the surface yields tracing error - Smoothing the direction field is computationally expensive

Table 4.8: Curvature-based hatching

4.3.4 Contours

The last illustrative style, included in VolumeFlies, draws the contours of a feature in the volume. The concepts and terminology, used within contours related literature, were briefly introduced in section 2.2. VolumeFlies currently creates the complete set of contours, which includes both the silhouette and the lines demarcating the highly curved areas. At this point, the silhouettes can not be distinguished from the inward contours. Furthermore, VolumeFlies does not include suggestive contours.

The way VolumeFlies deals with the contours, is very similar to the hatching approach. Also here, small segments are hatched; but only for the visible particles that reside near a contour. A particle is located exactly on the contour, if $\vec{n_i} \cdot \vec{e_i} = 0$ at the particles position.

The principal curvature is needed, in order to determine the hatch direction $\vec{d_i}$. This direction should follow the surface, in the direction of the contour. Therefore the change of the normal vector, should remain perpendicular to the view vector. Recall that the change of the normal vector, is contained in the principal curvature. A derivation for the hatch direction $\vec{d_i}$, is included in the VolumeFlies master's thesis [5].

$$\vec{d_i} = -\kappa_{2_i}(\vec{k}_{2_i} \cdot \vec{e_i})\vec{k}_{1_i} + \kappa_{1_i}(\vec{k}_{1_i} \cdot \vec{e_i})\vec{k}_{2_i} \tag{4.19}$$

Since particles will never reside exactly on the contour, a threshold t is introduced, which includes particles within a certain margin from the contour. Although proper tuning of the parameters delivers good results, there is still one major drawback. Areas with low curvature show wide contours, whereas highly curved areas show small contours. Inspired by the work by Kindlmann et al [17], a new parameter τ_i was introduced in the VolumeFlies master's thesis [5]. Putting a threshold on this parameter τ_i, instead of $\vec{n_i} \cdot \vec{e_i}$, results in contours of approximately constant width.

$$\tau_i = \frac{\vec{n_i} \cdot \vec{e_i}}{\sqrt{(-\kappa_{1_i}(\vec{k}_{1_i} \cdot \vec{e_i}))^2 + (-\kappa_{2_i}(\vec{k}_{2_i} \cdot \vec{e_i}))^2}} \tag{4.20}$$

An overview of the contours approach is included in table 4.9.

Complexity: $\mathcal{O}(m * h)$ with h = number of contour segments
Advantages: + Contour creation reuses the hatching approach
Disadvantages: - Contours are always view dependent (no pre-computation possible) - Sensitive parameter: Step size - Sensitive parameter: Threshold - Real-time curvature estimation computationally expensive - Method can't distinguish between silhouettes, contours and suggestive contours

Table 4.9: Contours

Chapter 5

Hardware rendered VolumeFlies

The previous chapters addressed the VolumeFlies approach to illustrative volume rendering. Different topics of interest were introduced, including some relevant terminology. Furthermore, the modules in the framework, depicted in figure 3.1, were each thoroughly described. This chapter again addresses the VolumeFlies modules, roughly maintaining the structure from chapter 4. In contrast to the previous chapter, now the hardware-rendered approach will be addressed.

The main goal for the hardware-rendered implementation of the VolumeFlies framework, is to achieve interactive volume inspection, applying a variety of illustrative depictions of the data. Therefore, the fundamental algorithms contained in the modules, need to be adjusted to optimally fit the graphics hardware. The resulting GPU-based modules are the main contribution of this thesis.

Regardless of the performance gain the graphics hardware can deliver, a performance analysis of the software-rendered VolumeFlies is necessary, before actually porting the framework to the GPU. In this analysis, the main bottlenecks should be identified, so that the hardware-implementation can anticipate on these performance flaws.

In order to oppose confusion, the name VolumeFlies will still refer to the software-rendered framework. The hardware-rendered implementation, presented in this chapter, will be named *VolFliesGPU*.

5.1 Performance analysis

Accurate measurement of the performance of a program, is a non-trivial task. It is hard to determine the time the program actually spends on the task under consideration. This holds especially for preemptive operating systems, which have the ability to suspend the measured process for higher priority tasks.

For our analysis we can afford some precision error, since we are mainly interested in identifying the performance bottlenecks in the system. The duration of each module is measured using the CPU tick counter, which gives a reasonable wall-clock time estimation, when dividing the number of ticks by the CPU clock frequency. The number of running processes were limited during the measurements, preventing our program to be preempted. Furthermore all measurements were executed in three runs, where the average counts as the final measurement.

From the measurement results, presented in table 5.1, two modules can be identified as a performance bottleneck. The most time consuming process is the particle redistribution, which will be the main module of interest for our optimisation. Secondly, the generation of the hatches consumes a considerable amount of time.

Both performance bottlenecks are modules that currently execute a pre-calculation step, which does not need to be executed on a frame-to-frame basis. In case the feature location and redistribution could be executed in real-time, then interactively changing the iso-level would become possible, while maintaining an evenly distributed particle set. In case the performance of the computationally expensive modules can not be sufficiently sped up, then still a performance gain improves the user interaction with the framework.

More critical are the modules that are processed each frame. They determine the level of interaction. A slight optimisation in one of these modules, immediately improves the overall frame-rate. Be aware that the measurements of these visualising modules are executed with the cone-splatting filter disabled, measuring the worst case execution time. For instance the visualisation of the contours on average becomes faster by a factor of four, when enabling the filter.

VolumeFlies General Information:		
Processor	Intel Core 2 Duo 2.4 GHz; 3GB RAM	
Dataset	CT Head (256^3 voxels x 8 bits)	
Number of particles	60.000	
Initialiser:		
Load volume data	3.92 sec	
Brute-force particle placement	9.83 sec	
Behaviour:		
Particle redistribution	**253.34 sec** (25 steps)	
Visualiser:		
Cone splatting	0.39 sec (per frame)	3 fps
Stippling (Density-based)	0.15 sec (per frame)	7 fps
Stippling (Scale-based)	0.15 sec (per frame)	7 fps
Hatch smooth field directions	7.37 sec	
Hatch generation (Direction-based)	**52.65 sec**	
Hatch generation (Curvature-based)	**53.05 sec**	
Hatch visualisation	1.32 sec (per frame)	1 fps
Contours	22.71 sec (per frame)	$<$ 1 fps

Table 5.1: VolumeFlies performance analysis

5.2 General GPU approach

The previous section points out that optimisation is essential, mainly for the particle redistribution and hatching approaches. In section 2.4 the computational power of the GPU was brought to the attention, which could potentially lead to the desired performance gain. In order to exploit this computational power, it is necessary to find a way to match the VolumeFlies modules to a GPU-based implementation.

The main mind-set is adopted from the GPGPU research field, introduced in section 2.5. Most of the tasks within the modules consist of some general computation, which should now be performed by the GPU. We propose a general approach, based on the extensions introduced in section 2.4.4. This approach allows to execute any computation on an 'array' of data, which performs at remarkable high speed, as long as the computations are as independent as possible.

In section 2.4.2, the way to implement programs for the GPU was introduced. Software designed for the GPU, consists of a series of shader programs, which together compose the programmable GPU-pipeline. The new general approach, which is depicted in figure 5.1, is based on this GPU-pipeline. This general approach will be elaborated here, since it is fundamental to all the VolFliesGPU modules.

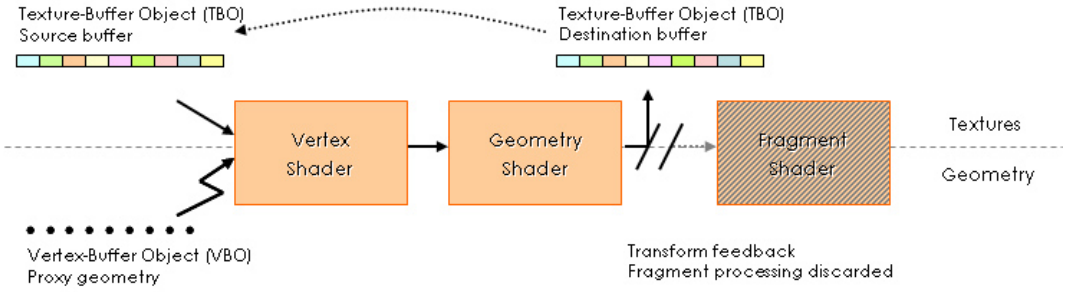


Figure 5.1: General GPU approach

First of all, the distinction between textures and geometry in figure 5.1 needs to be pointed out. Above the dashed line, along the pipeline, only textures or buffer objects are presented. Underneath the dashed line only geometry is depicted. There might be situations where geometry information is contained in a buffer, in which case the buffer is depicted above the dashed line.

Input

Firstly the inputs for the GPU-pipeline needs to be considered. They are depicted on the left-hand side in figure 5.1. In the upper-left corner the actual input data is presented, contained in a texture buffer object (TBO). The vertex shader, which is responsible for the processing of all vertices, can now gather data from an arbitrary position in the texture buffer. In most cases, the vertex shader thread is supposed to execute some computation for each data value in the texture buffer. In order to accomplish this, a vertex needs to be created for each data value in the buffer. This vertex then triggers the vertex shader, which is schematically depicted by the lightning-shaped arrow.

All vertices together form a *proxy geometry*. The proxy geometry is merely used as a placeholder to initiate the GPU-pipeline, without describing any object shape. As soon as the amount of values in the buffer is known, the proxy geometry can be generated CPU-side.

Subsequently, the proxy geometry is placed in memory on the graphics card, by using a vertex buffer object.
Besides the texture buffer, also other inputs can by provided to the GPU-pipeline. Be aware that inputs given to the GPU-pipeline, become available in all shader threads. This functionality can be used to provide several parameters to the GPU-pipeline, influencing the computations in the shader threads.

This seemingly artificial approach starts the execution of the entire pipeline for each vertex. Since the GPU is capable of processing many vertices in parallel, this immediately implies that many data values from the texture buffer can now be processed in parallel. The GPU is neatly used to do many computations simultaneously, where the shaders act as the computation kernel.

Output

Before considering the actual computations, the right-hand side of figure 5.1 should be addressed. In contrast to traditional approaches, the fragment shader is completely discarded. This is depicted in figure 5.1, by the darkened box for the fragment shader thread. Applications with image-space complexity, for instance GPU-based raycasting applications, rely heavily on the fragment shader. Older graphics cards even assumed a higher workload for the fragment shader, by assigning a considerable amount of processing units specifically to the fragment shader. Nowadays graphics cards are based on a unified shader architecture, where all processing units accept any shader thread. The unified shader architecture, which was introduced in section 2.4.3, substantially supports the proposed general GPU approach.

The transform feedback, depicted by the upward pointing arrow, accounts for the discarded fragment processing thread. The transform feedback returns computation results, by placing them into a buffer object. These values can be used for a next stage, without ever rendering the results to screen, which makes the fragment shader superfluous. The possibility of using a returned buffer as an input for the next rendering stage is depicted by the dashed arrow.

In practice there is a duality in the distinction between geometry and data values. The input texture buffer might indeed consist of arbitrary values, that need to be transformed by some function. Vertices then trigger the vertex shader stage to do so. The vertex shader however, is not designed to process general input values. Instead it expects to transform vertices, which come with a position and opacity value, denoted by a vector (x, y, z, a). As a consequence, the input needs to be adjusted to the needs of the vertex processing stage. Similarly, this holds for the geometry shader. Also the transform feedback returns the output values in the same vector shape.

Computation

Lastly we should distinguish between the vertex and the geometry processing stage. The vertex shader is only capable of transforming vertices. In the context of our computational approach, this means that either component of the vector can be adjusted by a function contained in the vertex shader. The geometry shader however, can also create and destroy vertices, during the GPU-pipeline execution. When looking at the computational analogy, this means that during a computation multiple results can be returned from a single input value. Furthermore it is possible to return no result at all.

5.3 Initialisation

Once again, the modules presented in the VolumeFlies framework (figure 3.1), will be addressed. The presentation is roughly consistent with the structure of chapter 4. All GPU-based modules are somehow based on the general GPU approach, presented in the previous section. For each module a figure will be presented, similar to figure 5.1, including the details for that particular module. The first module in the framework that needs to be considered, comprises the initialisation of the particle system.

Section 4.1 presented a brief argumentation, that pleads in favour of a brute-force placement of the particles. Although again other options were considered, the brute-force approach is preserved in VolFliesGPU.

5.3.1 Brute-force particle placement

It turns out that the brute-force initialisation is particularly suitable to be ported to the GPU. At first glance the task seems quite profound, since it addresses a considerable amount of voxels. This module executes relatively small tasks, locally in the volume. These small tasks need to be performed for a substantial amount of data, preferably for each voxel in the volume. Fortunately, the GPU is designed just to do this kind of work. It is manufactured for graphics processing, dealing with small operations on vertices, fragments and pixels.

The brute-force initialisation approach can be divided into two parts. The first part needs to provide a way to traverse the volume, addressing the voxel positions where the feature location will be executed. Secondly there is the actual particle placement, where the neighbours of each voxel will inspected, in order to determine if the implicit surface is crossed.

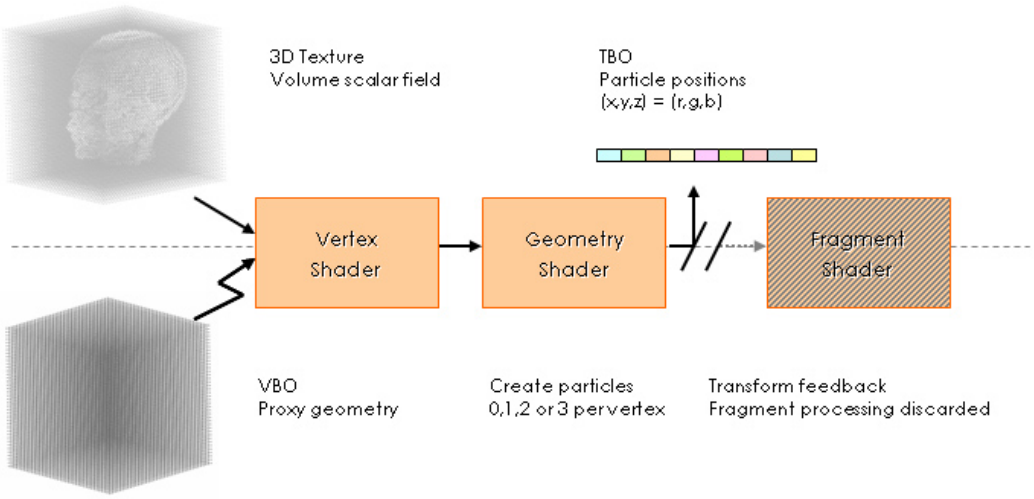


Figure 5.2: Brute-force particle placement on GPU

Volume traversal

The first issue that needs to be solved, is the traversal through the voxels. Therefore we need to return to the general GPU approach, presented in section 5.2. Recall that the GPU-pipeline needs to be triggered, in order to do any computation in the shader thread. This implies the need for a cubic grid of vertices, where each vertex triggers the GPU-pipeline. This cubic grid is used here as the proxy geometry. The shader thread can subsequently start the particle placement task. This requires that the volume is available as an input to the GPU-pipeline, and resides on GPU memory. This is schematically depicted on the left-hand side in figure 5.2.

One of the main benefits of this approach, is the way the vertices connect to the volume. The vertices are created at equally spaced positions between zero and one. Recall that such a proxy geometry is created CPU-side, and placed in GPU memory using a vertex buffer object. Furthermore, also the three-dimensional volume texture has texture coordinates clamped between zero and one. As a consequence, the coordinates of the vertices in object-space can be used as indices for the voxels in texture-space.

For the volume traversal, each vertex needs to trigger a new shader thread. The vertex position will be used as an index, to obtain the corresponding voxel, and its neighbours, from the volume texture. The main performance gain is obtained by the fact that many vertices, and hence many voxels, can be addressed in parallel. The amount of parallel threads that can be issued, depends on the amount of (unified) processing units on the graphics hardware. In case of the NVidia GeForce 8800GTX, there are 128 stream processors available.

Particle placement

Now that we can address each voxel in the volume, the second goal is to implement the particle placement task. Recall figure 4.1, showing that at most three particles will be created at each voxel position. Besides, there are also cases where only two, one or zero particles will be created.

The general GPU-approach (figure 5.1) shows, that a stream of vertices can be returned, using the transform feedback extension. This concept is valuable to the brute-force initialiser, since it provides a way to return the object-space coordinates of the newly created particles. The only trouble that remains is the arbitrary number of particles, that can be returned for each voxel location.

This arbitrariness can be resolved, by using the recently introduced geometry-shader extension, which allows us to create or destroy geometry. For each located particle a vertex can be created, which position is set to the corresponding particle position. In case no particles are located, the geometry shader will return no vertex at all. As a result, the geometry shader can return zero, one, two or three particle positions. A transform feedback is applied to return these vertices, which resemble the particle positions, to a new one-dimensional texture buffer.

The GPU-based implementation of this module does not require any extra steps. The complexity of the algorithm is equal to the complexity of the VolumeFlies approach (table 5.2).

Complexity VolumeFlies:	$\mathcal{O}(n)$ with n = number of voxels
Complexity VolFliesGPU:	$\mathcal{O}(n)$ with n = number of voxels

Table 5.2: Initialiser complexity

5.4 Behaviours

The second stage of the framework, depicted in figure 3.1, consists of the behaviour modules. The only behaviour module that is currently available in the framework, is the particle redistribution. This section describes the hardware-rendered version of the particle redistribution.

5.4.1 Particle redistribution

As opposed to the brute-force initialisation, the particle redistribution does not naturally fit the GPU. In literature, similar redistribution approaches are typically software-rendered. In this section, we propose a GPU-based particle redistribution, based on the concepts of hardware-rendered particle systems. The work by Kruger et al. [20] presents a GPU-based particle system of particular interest.

The previous section shows, that the brute-force initialisation results in a texture buffer object, containing the particle positions that need to be redistributed. A texture, and thus a texture buffer, usually stores the colour values ($r$ed,$g$reen,$b$lue) and an opacity values (a). The scalars values of the colour can be substituted by the object-space position of the particle (x, y, z).

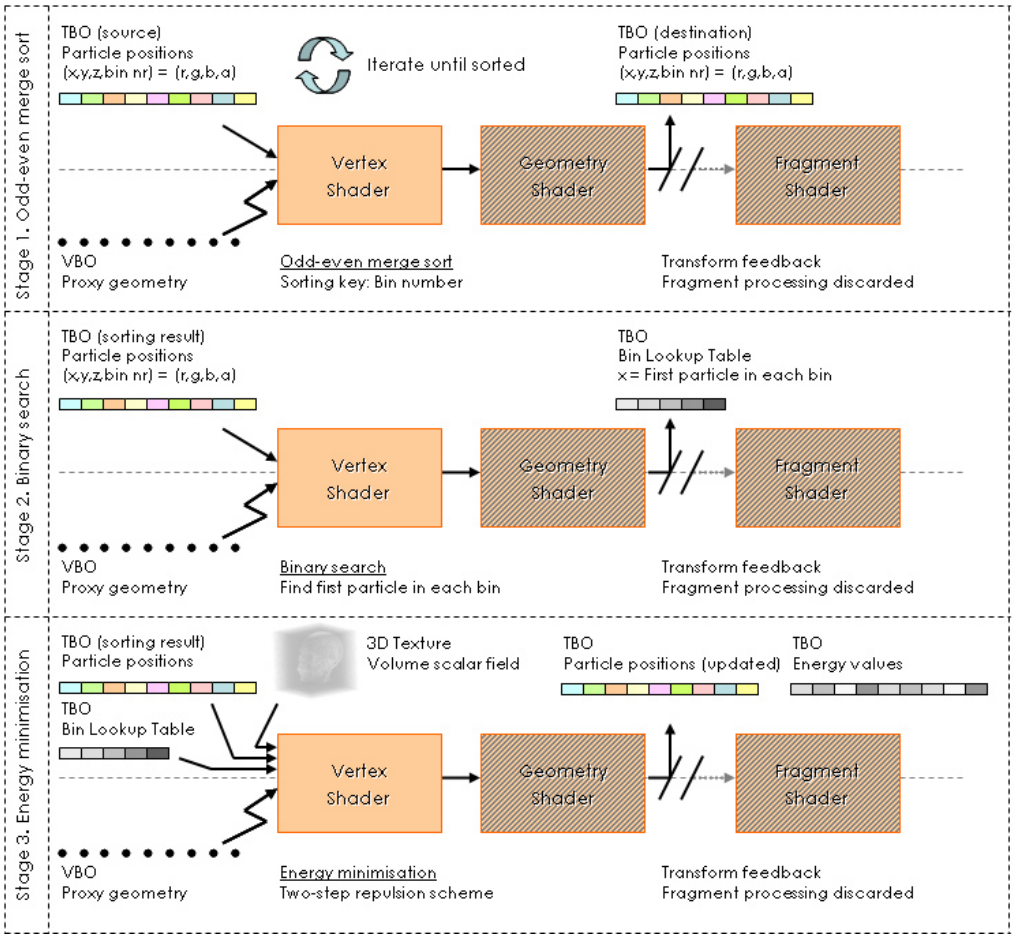


Figure 5.3: Particle redistribution on GPU

The texture buffer, returned by the brute-force initialiser, is used as an input for the redistribution module. The goal is to apply the two-step repulsion scheme, using the power of the GPU. Efficient implementation of this repulsion scheme, which was presented in section 4.2.1, requires a binning structure. This structure divides the volume in equally sized cubic regions, which are called bins. Each particle now resides within one of the bins. The binning structure can be used to keep the computations on the particle local, by selecting a neighbourhood of 3^3 bins, surrounding the particle position. Each bin in the binning structure is assigned a unique id number, which makes it easier to quickly identify a particular region in the volume.

Again the implementation relies on the general GPU approach, depicted in figure 5.1. The transformations on the particles, are initiated by a 1D proxy geometry, which indexes the input particle texture buffer. The proxy geometry needs to be created only once, since the number of particles do not change during redistribution. A schematic layout of the texture buffer object is depicted in figure 5.4.

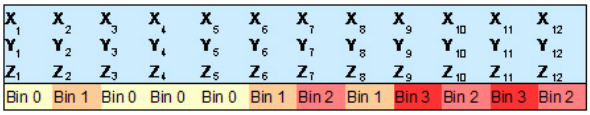


Figure 5.4: Schematic memory layout of particle TBO

The repulsion scheme comprises an iterative approach, which is executed in four stages. In the first stage the particle are sorted. Subsequently, a lookup table is created that allows quick lookup of the particles in a particular bin. At this point, the repulsion scheme can applied, displacing all particles. The last stages verifies whether the system has reached an equilibrium. This stage was not incorporated in VolumeFlies. Instead a fixed number of iterations was executed.

Stage 1: Sorting the particles

The first step in the repulsion scheme (equation 4.8), comprises a displacement of each particle, based on the emitted energy of neighbouring particles. Therefore, the positions of the particles in the neighbourhood should be known during computation. However, searching for all the neighbours in the neighbourhood is computationally expensive. It requires to search through the entire buffer, for each particle. Such a step would be of complexity $\mathcal{O}(m)$ and should be executed for each particle, where m is the number of particles. The complexity and the time needed to search for neighbours can be reduced, by sorting the particles by the associated bin number. That way, the particles are sorted, based on their location in the volume. As a consequence, in the sorted buffer it suffices to gather exactly the particles in the neighbouring bins, instead of gathering all the particles in the buffer.

Of course, also the sorting algorithm takes up a considerable amount of time. In order to speed-up this process, a parallel sorting algorithm is adopted, exploiting the capacity of the GPU. The implemented sorting algorithm, is the *odd-even merge sort*, presented in chapter 46.3 of the 'GPU Gems II' book [29]. The bin number is used as the sorting key. The algorithm is not data-dependent, which means that the particles are always sorted according to the same scheme, regardless of the data. It comprises an iterative process, comparing the order of both the odd and the even positions in the particle buffer. After each iteration, the intermediate results are merged, and returned for the next iteration. The comparison tasks can be executed completely parallel, which perfectly suits the GPU.

The odd-even merge sort can be executed in the vertex shader, since each iteration returns exactly the same amount of particle positions. The sorting stage requires two texture buffers to store the particles. The texture buffer containing the intermediate results of the last iteration, is taken as an input for the next iteration. New results are stored in the second texture buffer object, using the transform feedback extension. For each new iteration the buffers are swapped. This is called a *ping-pong* approach. A schematic memory layout of the sorted texture buffer object is depicted in figure 5.5.

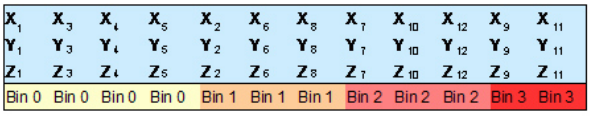


Figure 5.5: Schematic memory layout of sorted particle TBO

Stage 2: Creating the bin-lookup table

The sorted texture buffer is taken as an input for the second stage of the particle redistribution, depicted in figure 5.3. Although the particles in the texture buffer are now sorted, the shader thread can not determine the buffer index of the first particle in a particular bin. This is due to the fact that each bin contains an arbitrary number of particles. As a result, for each particle the shader thread needs to search for the first occurrence of a particle in a bin. This involves a search through all particles, until the first particle in the required bin is found. In the worst case situation, this implies addressing all particles, annulling the profit of the sorting algorithm. Various search algorithm can speed-up this process of finding the first particle in a bin. Nevertheless, searching the first particles in all neighbouring bins has to be executed for each particle, which is a time-consuming process.

This problem can be resolved by creating a specific lookup table, which lists the indices of the first particles contained in each bin. Subsequently, the neighbours of each particle can be obtained, by directly looking up the indices of the first particles of neighbouring bins. A bin-lookup table is schematically depicted in figure 5.6. For instance, it directly follows from the bin-lookup table, that the first particle of bin 1 is located at index 4 in the buffer.

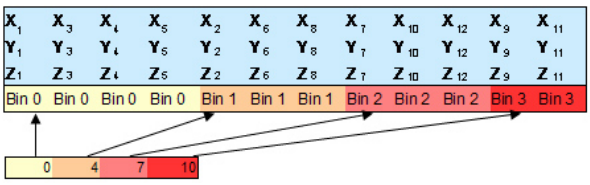


Figure 5.6: Schematic representation of the bin-lookup table

Be aware that the bin-lookup approach comes at a cost of creating the table. The creation of this lookup table, is executed in the second stage of the GPU-based particle redistribution. Figure 5.3 shows that the creation of the lookup table involves a *binary search* through the particles buffer, for each bin in the binning structure. The proxy geometry triggers the vertex shader thread for each bin. This starts parallel executions of the binary search, looking for the first particle in the bin belonging to the shader thread. Again the parallelism implies a fast execution of the algorithm, when using the GPU. Moreover, the merit of using the bin-lookup table, is the fact that we need to search for the first particles in a bin only once for all bins, instead of multiple times for all particles. In general this infers a performance gain, since there are typically less bins than particles.

Stage 3: Particle repulsion

At this point, the first two stages delivered a sorted set of particles and a bin-lookup table. Both are schematically depicted in figure 5.6. Together with the volume, these are the inputs for the third stage of the GPU based particle redistribution. This stage actually performs the two-step repulsion scheme. The vertex shader thread is triggered for each particle, after which the particles are displaced in the tangent plane, based on the repulsion of the neighbouring particles. These neighbouring particles are quickly resolved, by using the bin-lookup table. Secondly the particle is reprojected back to the surface (equation 4.10). This energy minimisation stage, depicted in figure 5.3, results in texture buffer containing the displaced particles. The new locations should bring the particle system one step closer to an equilibrium.

At this point all stages can be repeated, until the particles are evenly distributed over the surface. We have seen that the texture buffer that results from the third stage contains the displaced particles, after one step of the particle repulsion scheme. This buffer should now be used as an input for the first stage of a new iteration. The particles will be sorted again, and a new bin-lookup table needs to be created. Subsequently, the particles are displaced again, moving them one step closer to an equilibrium. The resulting buffer can again be passed on to the first stage for a new iteration, closing the iteration loop. The effect of the particle redistribution is depicted in figure 5.8. Furthermore, a complete pseudo-code of the presented stages in included in appendix C.

Stage 4: Stop criterion

In VolumeFlies, the number of iteration for the complete process was defined by the user. Typically the entire redistribution was executed 25 times. In VolFliesGPU a stop criterion was introduced, similar to the stop criterion proposed by Meyer et al. [25]. Verification of the stop criterion is executed during an additional fourth stage. As a consequence, the intermediate particle buffer resulting from the energy minimisation stage is passed on to the first stage, just after the stop criterion is verified.

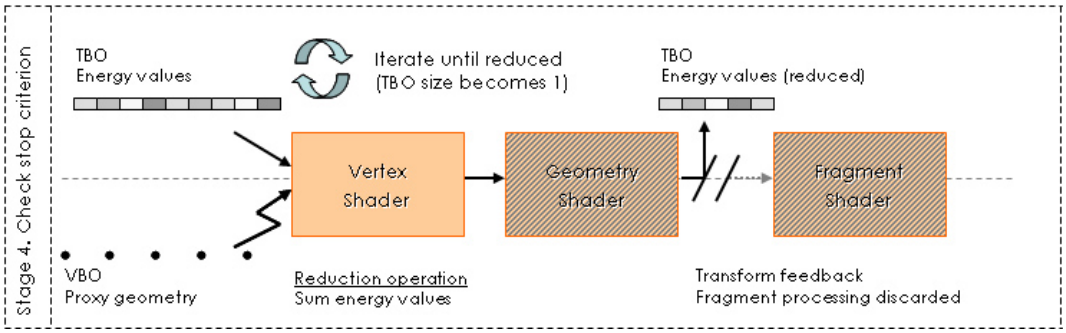
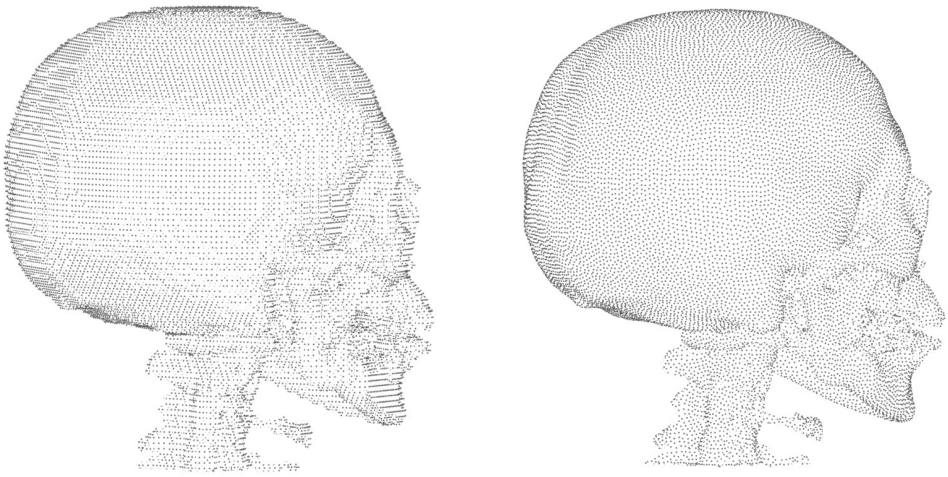


Figure 5.7: Redistribution stop criterion on GPU

The change of the total system energy is verified after each iteration. When this change becomes less than a small fraction of the total energy, the redistribution is stopped. Furthermore, a maximum number of iterations is defined, to make sure the system will not keep moving around its equilibrium. Typically, a redistribution now converges within 10 to 15 steps.

In order to verify this stop criterion, the energy values for all particles should be available. Observe in figure 5.3 that during the third stage of the redistribution approach, the energy values are returned to a texture buffer. This shows that the transform feedback is capable of returning values to multiple buffers, since the intermediate buffer with particle positions is returned simultaneously.

Figure 5.7 depicts the GPU-based stage that verifies the stop criterion. Using the buffer with energy values as an input, the total system energy can be calculated. Summation of the particle energies is a serial operation, which is hard to execute on the GPU. Recall from chapter 2.5.1, that a reduction technique can be applied to add up large amounts of values using the GPU. This typical GPGPU technique, presented by pseudo code 1, is executed iteratively. During each iteration pair-wise values are added. The reduction continues until only one value remains. This value is the required total system energy and is fetched from GPU memory. Subsequently, it is decided CPU-side if the system has reached an equilibrium.



(a) CT head dataset, after brute-force initialisation

(b) CT head dataset, after particle redistribution

Figure 5.8: Particle repulsion evenly distributes the particles over the surface

It is clear that the GPU-based implementation of this module requires some extra steps. The complexity of the total redistribution is worse than the complexity of the VolumeFlies approach (table 5.3). Nevertheless, the redistribution is still much faster, because of the parallelism and computational power of the GPU are exploited.

Complexity VolumeFlies (Per iteration):	
Per iteration: $\mathcal{O}(mb)$ with b = average number of neighbours	
Complexity VolFliesGPU (Per iteration):	
Odd-even merge sort:	$\mathcal{O}(m\ log^2(m) + log(m))$
Binary search:	$\mathcal{O}(d\ log_2(m))$ with d is the average number of bins
Two-step repulsion:	$\mathcal{O}(mb)$ with b = average number of neighbours

Table 5.3: Redistribution complexity

5.5 Visualisation

For the first steps of the framework, depicted in figure 3.1, the hardware-rendered approaches were presented in the previous sections. This section will present the GPU-based approaches for the modules that take care of the illustrative visualisation. Similar to the visualisation section in chapter 4, first the cone-splatting approach is addressed.

5.5.1 Cone splatting

Recall that the cone-splatting approach is not just a visualisation module, moreover it can act as a particle filter. This filter removes all particles that do not reside on a visible surface. In the approach proposed in VolumeFlies, the implicit surface was created, by placing directed and scaled cones at the particle positions. These cones were colour coded, using a unique colour for each cone. Subsequently, each particle uses the resulting colour buffer to verify whether the particle should be visible or not.

Again the concepts of the general GPU approach, depicted in figure 5.1, are used to come up with a GPU-based version of this module. The first stage, presented in figure 5.9, creates the implicit surface. The GPU-pipeline is triggered for each particle, after which the cones will be created for each particle. The geometry shader extensions provides a legitimate solution for creating the cones. Subsequently, the second stage of the cone-splatting approach determines which particles reside on a visual surface.

Stage 1: Colour buffer creation

After the GPU-pipeline is triggered, a vertex will be created by the geometry shader, which is located at the associated particle position. Using this particle position as a starting point, the geometry shader adds geometry, creating a cone by using a triangle strip. The cones are positioned perpendicular to the viewing plane, and are scaled according to equations 4.12 and 4.13. As a unique colour coding, the object-space particle position coordinates are assigned, where the colour (r, g, b) equals the positions (x, y, z). Figure 5.10a shows that the cones are positioned perpendicular the viewing plane, with the tops pointing towards the viewer. Furthermore, the scaling and colour coding is depicted.

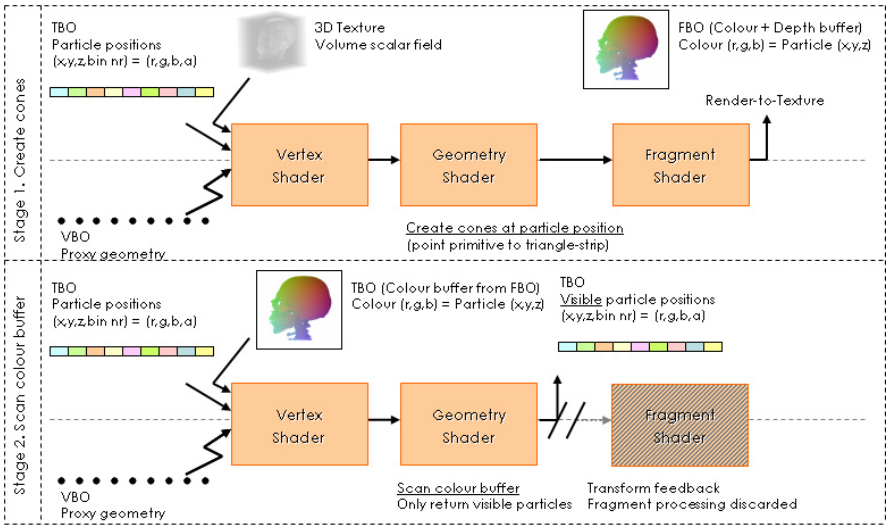


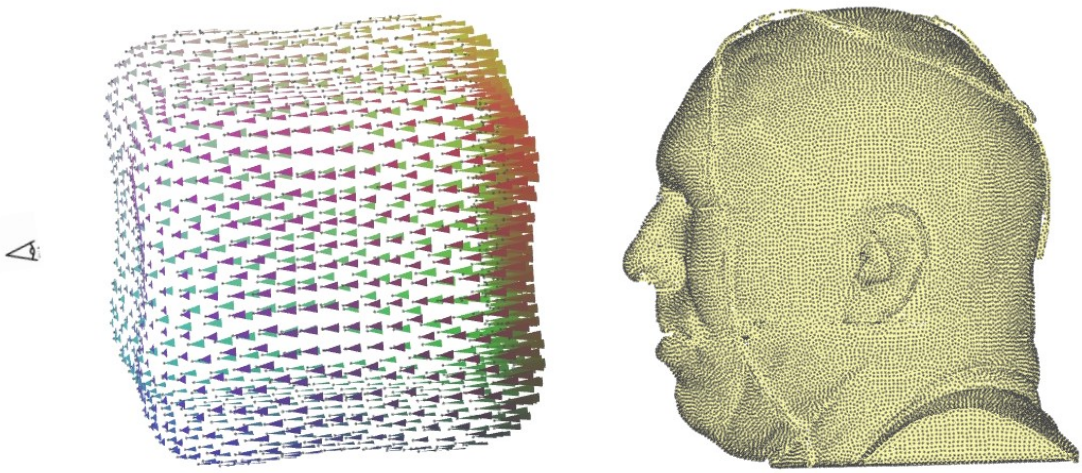
Figure 5.9: Cone-splatting on GPU

All the cones together form the implicit surface of the feature in the volume. This is depicted in figure 5.10b. For each frame, this surface needs to be rendered to a two-dimensional off-screen buffer. A frame-buffer object (FBO) is used as a render target. Later on, this off-screen frame-buffer can be transformed into a texture buffer object. This method of rendering to an off-screen buffer, which can be used as a texture, is an example of a *render-to-texture* approach. The resulting texture buffer object is required as an input for the filtering approach, which is the second stage, depicted in figure 5.9.

Stage 2: Determine particle visibility

The second stage of the GPU-based cone-splatting approach, filters the particles that reside on the visible part of the surface. Again the GPU-pipeline is triggered for each particle. The colour buffer needs to be checked per particle, to see whether the particle position occurs as a colour. If the colour occurs, the particle is part of the visible surface, and should returned to the screen. However, no particle should be returned in case it is part of a non-visible surface. Again the geometry shader extension is applied, which destroys the geometry of the particles that should not be visible. By using the transform feedback, this results in a new texture buffer object containing only the particles that are part of a visible surface.

As opposed to VolumeFlies, the colour buffer will not be scanned to find the colour. Instead, the particle position is projected to the viewing plane, resulting in eye-space coordinates. These coordinates can be used to verify the colour at the correct position in the colour buffer, without scanning through the buffer. This results in a considerable performance gain.



(a) Cones direction, scaling and colour coding

(b) Cones create a closed iso-surface

Figure 5.10: Cone-splatting approach

The GPU-based implementation of this module does not require any extra steps. The complexity of the first stage is equal to the complexity of the VolumeFlies approach. However, the complexity of the filtering is reduced, by directly indexing the colour buffer (table 5.4).

Complexity VolumeFlies:	
Surface splatting	$\mathcal{O}(m)$
Filtering	$\mathcal{O}(mk)$ with k = number of pixels
Complexity VolFliesGPU:	
Surface splatting	$\mathcal{O}(m)$
Filtering	$\mathcal{O}(m)$

Table 5.4: Cone-splatting complexity

5.5.2 Stippling

The stippling module is the first module that actually performs a stylistic visualisation. In VolumeFlies, two different stippling approaches were presented. Both these approaches are adopted in the hardware-rendered framework. This section describes both GPU-based approaches, which are depicted in figure 5.11.

Recall that for the first approach, the shading of the feature is altered, by changing the density of the particles. The density of the redistributed particle set is assumed to represent the darkest shade. Using a basic diffuse lighting, particles are removed in the brighter areas.

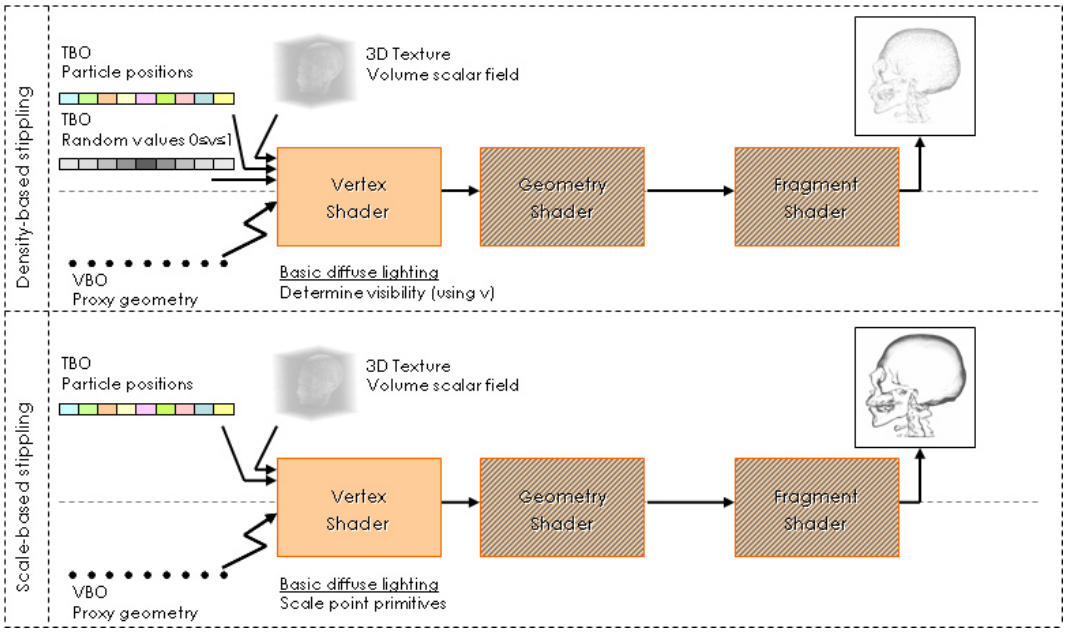


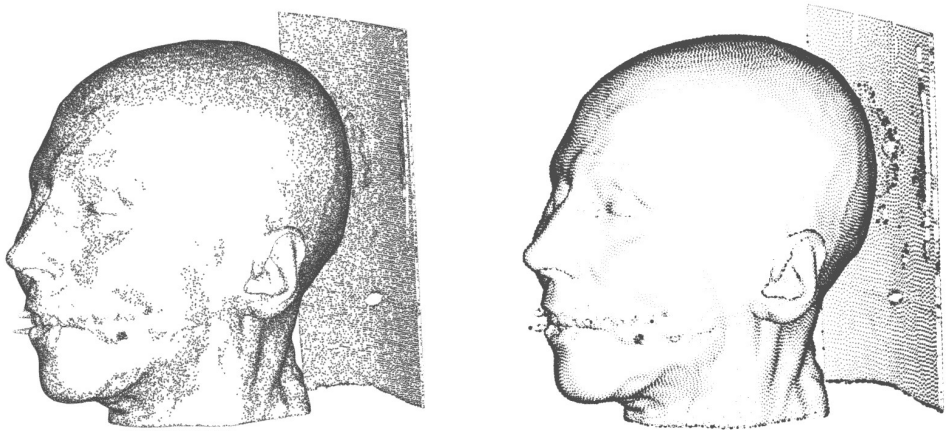
Figure 5.11: Stippling on GPU

For the GPU-based implementation of this module, again the general GPU approach can be applied (figure 5.1). The GPU-pipeline can be triggered for each particle, after which the shader thread has to determine whether a particle should be visible. This visibility is measured, based on the basic diffuse lighting and the random value v. The basic diffuse lighting is defined by equation 4.14, and the value v was a fixed property assigned to each particle. Adding properties to the particles in the GPU-based system, requires a new texture. This property texture should have the same length as the particle texture. For each particle, a property can be obtained from the property texture, by using the current buffer index.

The visibility of the particles is determined in the vertex shader thread, identical to the VolumeFlies approach. Removed particles are displaced to a location, which is out of sight of the user. Therefore, no geometry shader is necessary to destroy the unwanted particles. The visible particles can be rendered directly to the screen. The result of the density-based approach is depicted in figure 5.12a.

The second approach, is the scale-based stippling approach. The point primitives representing the particle, are scaled based on the same basic diffuse lighting. The approach is almost similar to the GPU rendered density-based stippling. In this case however, the random values v are not used. The vertex shader thread deals with the scaling, defined by equation 4.16. The result is again directly rendered to screen, and is depicted in figure 5.12b.

The visibility threshold t, which was introduced in section 4.3.2, is incorporated in both stippling methods. This parameter is provided as an input to the GPU-pipeline, and allows the user to control the intensity of the applied lighting.



(a) Density-based stippling (b) Scale-based stippling

Figure 5.12: Stippling approach

The GPU-based implementation of this module requires no extra steps. The complexity of the module is equal to the complexity of the VolumeFlies approach (table 5.5).

Complexity VolumeFlies:	$\mathcal{O}(m)$
Complexity VolFliesGPU:	$\mathcal{O}(m)$

Table 5.5: Stippling complexity

5.5.3 Hatching

The next visualising module illustratively depicts features in the volume, using a traditional technique called hatching. From the performance comparison we observe that both proposed hatching approaches are time consuming. This section describes, how the performance of generating and visualising the hatches, was improved using the GPU.

Direction-based hatching

The first hatching approach, presented in VolumeFlies, traces the hatches in a single direction. Recall from section 4.3, that this direction-based hatching is executed, by projecting segments of the hatch to the tangent plane. Tracing a series of these projected segments creates the hatch. The hatches are pre-calculated and stored to a buffer, because generating the hatches is computationally expensive. The approach consists of two stages. First the hatches will be traced and stored to a buffer. Subsequently, the generated hatches will be visualised.

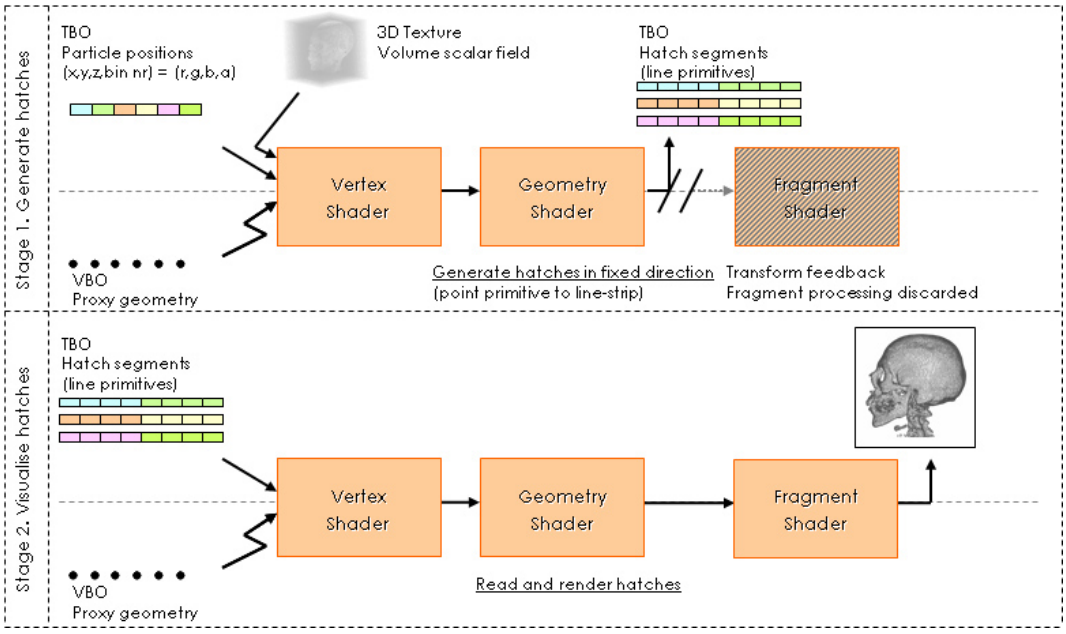


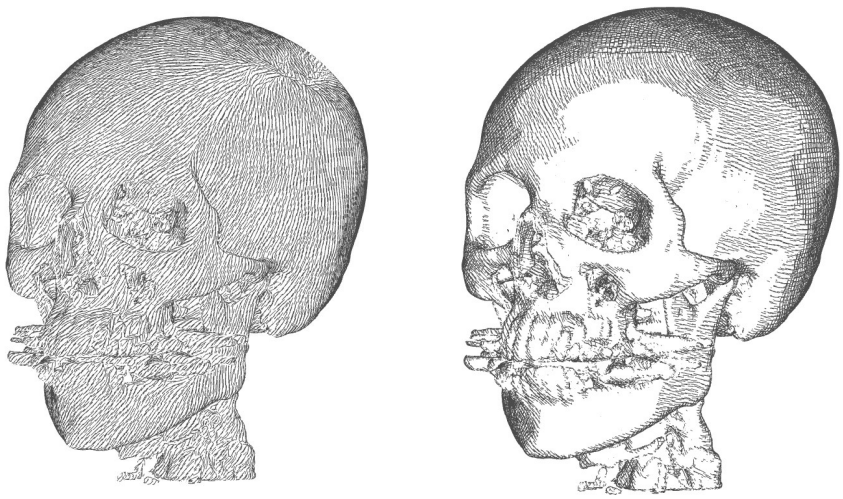
Figure 5.13: Direction-based hatching on GPU

Stage 1: Tracing the hatches

Similar to the VolumeFlies approach, the generation of the hatches comprises the first stage of the GPU-based approach. In figure 5.13, this stage is depicted. Based on the general GPU approach (figure 5.1), the GPU-pipeline is triggered per particle. The particle positions, gathered from the texture buffer, are used as the starting points for tracing the hatches. Adding segments to the hatch, implies the creation of new geometry. Such a task is well suited for the geometry shader extension. Each geometry shading thread returns a series of connected vertices, called a line-strip. These line-strips represent the hatches, and are stored in a texture buffer. Storing the line-strips is again performed through a transform-feedback operation.

Stage 2: Visualise the generated hatches

The second stage of the direction-based hatching is responsible for the visualisation of the hatches. The line-strips that were stored in the texture buffer, are available as an input for the second stage. Again, the GPU-pipeline is triggered per particle position. During the shader thread, the hatches are gathered from the texture buffer and subsequently rendered to screen. During this visualisation stage, the colour and line thickness of the hatches can be changed. Results of the direction-based hatching approach are depicted in figure 5.14.



(a) Direction-based hatching, using single hatches

(b) Direction-based hatching, using cross-hatches and lighting

Figure 5.14: Hatching approach

The GPU-based implementation of this module requires no extra steps. The complexity of the module is equal to the complexity of the VolumeFlies approach (table 5.6).

Complexity VolumeFlies:	$\mathcal{O}(d\,m\,h)$ with $h = \#$ hatch segments, $d = \#$ hatch directions
Complexity VolFliesGPU:	$\mathcal{O}(d\,m\,h)$ with $h = \#$ hatch segments, $d = \#$ hatch directions

Table 5.6: Direction-based hatching complexity

Curvature-based hatching

Moreover, VolumeFlies presented a hatching approach, which follows the curvature of the implicit surface. Recall from interlude 1, that calculating the curvature requires measuring the first and second order partial derivatives in the volume. Typically, these computations are time consuming.

A fast approximation of the surface curvature is valuable, although calculating the curvature can be performed in a pre-computation step. In case the computation time of the curvature can be improved, this would lead to a significant speed-up of the process for generating the curvature-based hatches. Furthermore, other properties like the hatch colour could be based on the principal curvature information in real-time.
A GPU-based approach curvature-estimation was proposed by Sigg and Hadwiger in the GPU-GEMS II book [29]. This method approximates the curvature in real-time, at any position in the volume. This approach was implemented in VolFliesGPU, and can be used as a general module for different kinds of applications. The real-time curvature approach is presented in interlude 2. It describes the GPU-based approach of the curvature estimation, which was presented in interlude 1.

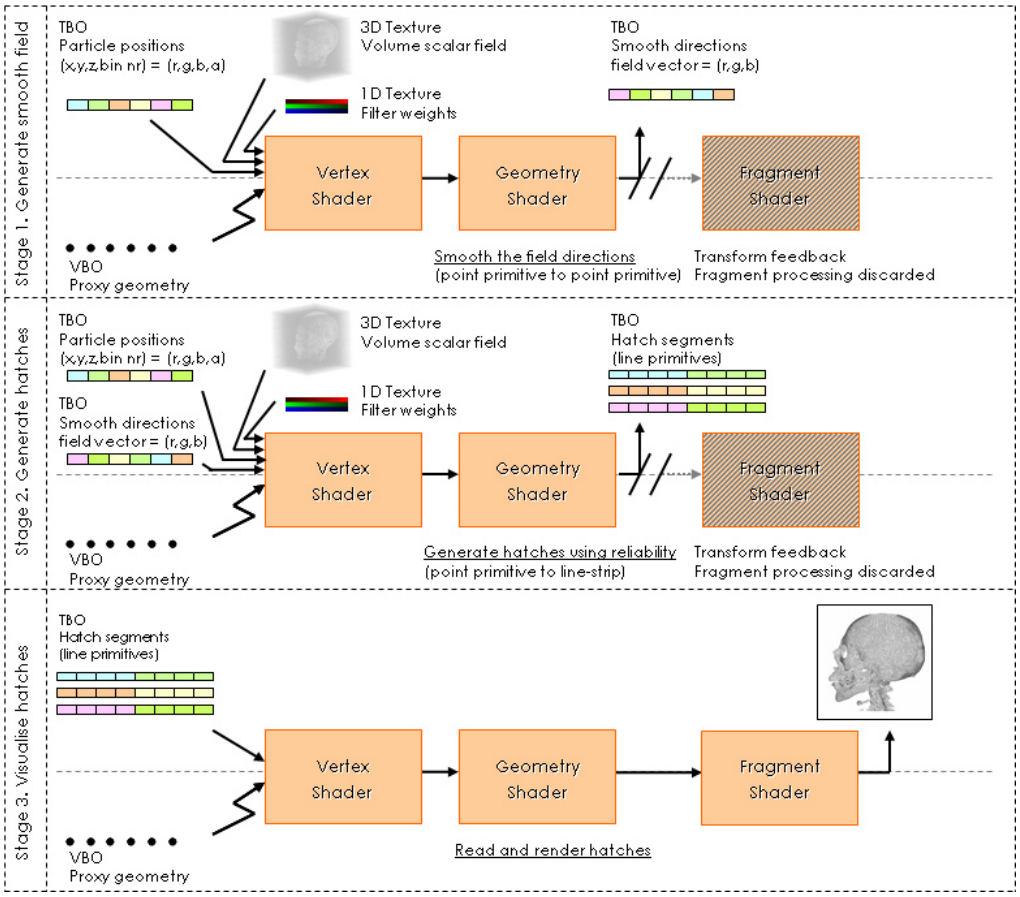


Figure 5.15: Curvature-based hatching on GPU

For real-time curvature estimation, the approach presented by Sigg and Hadwiger is adopted in VolFliesGPU. In chapter 20 of GPU-Gems II [29], they present a convolution based filtering approach, which can be applied for derivative reconstruction in the volume. These derivatives can be used to calculate the implicit surface curvature.

1. Fast cubic convolution
   Performance of the convolution is optimised for the GPU, by rewriting the convolution sum. The filter kernel is given by a cubic B-spline, which comprises a piecewise function defined by polynomials. For such a kernel, the one-dimensional convolution can be rewritten as:
   $w_0(x)\ k_{i-1} + w_1(x)\ k_i + w_2(x)\ k_{i+1} + w_3(x)\ k_{i+2}\ =\ g_0(x)\ k_{x-h_0(x)} + g_1(x)\ k_{x+h_1(x)}$

   Where the data samples k_i are weighted by $g_0(x)$, $g_1(x)$, $h_0(x)$ and $h_1(x)$.
   These are defined as:
   $g_0(x) = w_0(x) + w_1(x); \quad h_0(x) = 1 - \dfrac{w_1(x)}{w_0(x) + w_1(x)} + x$
   $g_1(x) = w_2(x) + w_3(x); \quad h_1(x) = 1 - \dfrac{w_3(x)}{w_2(x) + w_3(x)} - x$

   In the one-dimensional case, the rewritten convolution sum, results in gathering of two data samples, and the execution of one linear interpolation. This is faster compared to a straightforward approach, especially in higher dimensions. Moreover, the functions $g_i(x)$ and $h_i(x)$ can be pre-computed.
   The B-spline filter kernel weights $w_i(x)$ are defined as:
   $w_0(x) = \frac{1}{6}(-x^3 + 3x^2 - 3x + 1) \quad w_1(x) = \frac{1}{6}(3x^2 - 6x + 4)$

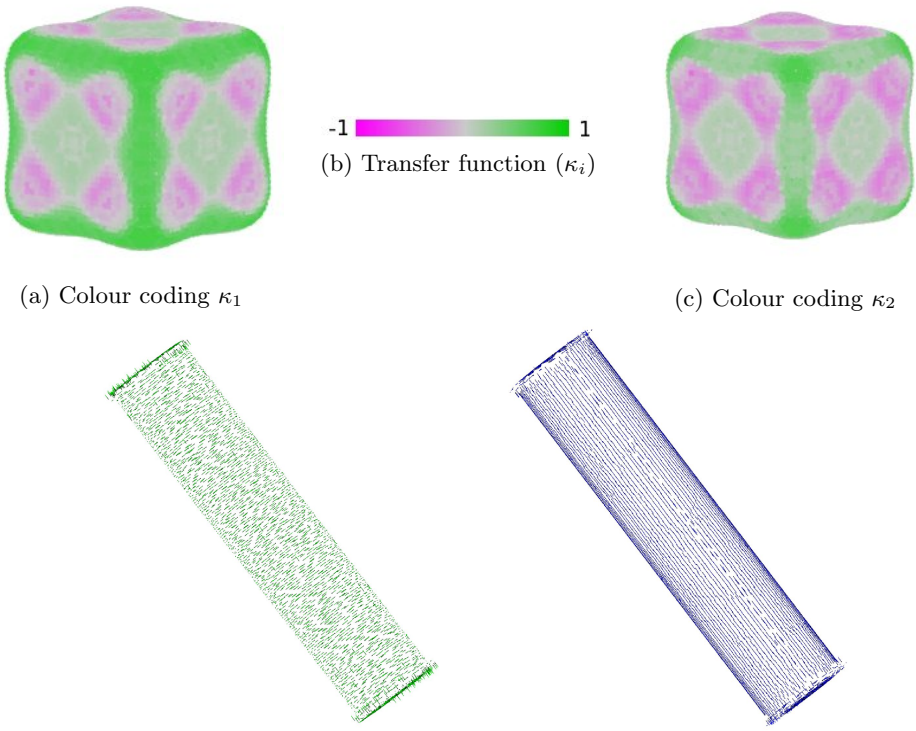
   $w_2(x) = \frac{1}{6}(-3x^3 + 3x^2 + 3x + 1) \quad w_3(x) = \frac{1}{6}x^3$
2. Derivative reconstruction
   Recall from interlude 1, that for the curvature calculation the first and second order partial derivatives are necessary. These can be reconstructed, by convolving the volume data with the correct derivative of the filter kernel. As a result, the gradient $\vec{g}$, and the hessian matrix H can be measured for each location in the volume.
3. Curvature computation Sigg observed that the calculation of the curvature can be executed, entirely in tangent space. Recall that the principal curvatures where computed by eigen-analysis on the shape operator S (interlude 1). The normal vector $\vec{n}$ is by definition one of the eigenvectors. Hence the remaining eigenvectors, comprising the principal curvature directions, can be derived in two-dimensional tangent space. This simplifies the complexity of the calculation, and increases the overall performance.
   The shape operator (A) is transformed by some orthogonal basis $(\vec{u}, \vec{v})$:
   $A = (\vec{u}, \vec{v})^T \dfrac{H}{|\vec{g}|} (\vec{u}, \vec{v})$
   The principal curvatures can now be computed as follows (T=trace(A), D=det(A)):
   $\kappa_1 = \frac{1}{2}(T + \sqrt{T^2 - 4D}); \quad \kappa_2 = \frac{1}{2}(T - \sqrt{T^2 - 4D})$
   $\vec{k}_1 = a_{12}\vec{u} + (\kappa_1 - a_{11})\vec{v}; \quad \vec{k}_2 = a_{12}\vec{u} + (\kappa_2 - a_{11})\vec{v}$
   An extensive derivation can be found in appendix D

Interlude 2: Principal curvature estimation

The curvature approach, presented in interlude 2, offers the sought-after fast curvature estimation method. However, the method has to be incorporated in VolFliesGPUframework. Therefore, the transformed convolution weigths, given by functions $g_i(x)$ and $h_i(x)$ in interlude 2, need to be pre-computed once. The results are stored in a texture, which can be used as an input for any module that requires curvature information. The derivative reconstruction and curvature computation are performed by a shader thread. This thread requires the information of the weigths, and thus the input texture. In VolFliesGPU, these computations are typically performed in the vertex shader thread.

In order to demonstrate the results of the GPU-based curvature estimation, the dataset presented by Kindlmann et al [17] was adopted. An identical colour coding was executed, by means of a one-dimensional transfer function on the principal curvature values. The results, depicted in figure 5.16a and 5.16c, are very similar to the the curvature estimation results presented by Kindlmann et al.

Moreover, the principal curvature directions are demonstrated using a synthetic data set. The data set defines a cylinder, for which the curvature directions are clearly defined. In figure 5.16d and 5.16e, the principal curvature directions are depicted as unit vectors on the cylinder. Be aware that κ_2 in this example is zero, since there is no curvature in that direction.



(a) Colour coding κ_1

(b) Transfer function (κ_i)

(c) Colour coding κ_2

(d) Principal curvature direction $\vec{k}_1$

(e) Principal curvature direction $\vec{k}_2$

Figure 5.16: Real-time curvature evaluation

At this point the GPU-based curvature estimation can be used for the curvature-based hatching module. The remainder of this section describes the hardware-rendered approach for this module. Similar to the VolumeFlies approach, the GPU-based method is divided into three stages. However, the algorithms in the stages are slightly different. The GPU-based approach is depicted in figure 5.15.

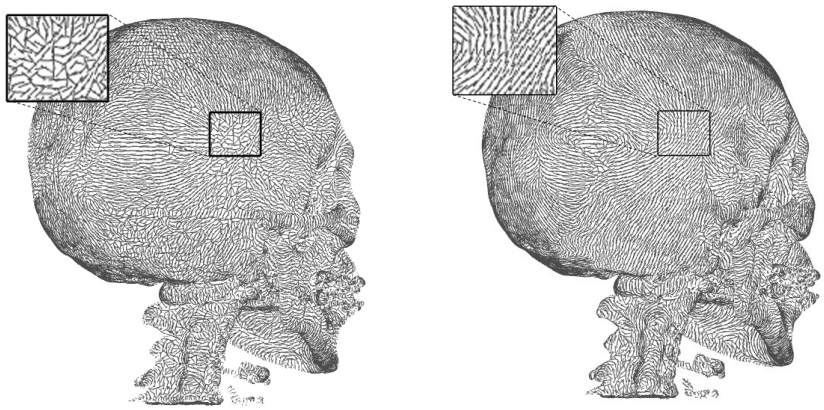
Stage 1: Smoothing the direction field

The first stage, creates a smooth field of directions, starting from the particle positions. At each particle position, the principal curvature directions are calculated. These curvature directions form the initial field, before smoothing. The smoothing step computes a weighted average s_i, based on the principal curvature directions $\vec{k}_{1_j}$ of particles in the neighbourhood. Similar to the redistribution step, the bin-structure is used to determine the nearby particles, that should be considered in the weighting. The averaging of the directions is defined as:

$$\vec{s}_i = (w_T)\frac{\sum w_j \vec{k}_{1_j}}{\sum w_j} + (1 - w_T)\vec{s}_T, \text{ where } w_T = (\frac{\sum \rho_j}{b}) \tag{5.1}$$

The smoothing weights w_j are defined by the reliability ρ of the surface, at the particle position p_i. This reliability was defined in equation 4.18, and locally indicates to what extend the surface is suited for hatching. For areas where there is little or no curvature, an arbitrary but fixed direction $\vec{s}_T$ is introduced. The field-direction is guided toward this fixed direction, dependent on the weight w_T, where b is the total number of neighbours.

The hardware-rendered field smoothing, again relies on the general GPU-approach. The first stage, depicted in figure 5.15, is triggered for each particle. Subsequently, the curvature directions at the particle position need to be measured. The real-time curvature estimation requires the pre-computed texture, containing the transformed convolution weights. Since for each particle position, two curvature directions will be calculated, the geometry shader extensions is applied. During the geometry shading thread two vertices are created. Their position represent the two measure curvature directions. Lastly the transform feedback returns the curvature direction into a new texture buffer. The effect of smoothing the direction field is depicted in figure 5.17.



(a) Curvature-based hatching, using principal curvature directions

(b) Curvature-based hatching, using smoothed field based on the principal curvature

Figure 5.17: Smoothing the field directions

Stage 2: Tracing the hatches

Based on the smoothed direction field, the hatches can be generated. This second stage (figure 5.15) traces the hatches along the surface, based on the surface curvature. As opposed to VolumeFlies, the direction of the segments are not weighted with the smoothed directions of the particles in the local neighbourhood. For performance reasons, this approach was replaced by a tracing method, that depends only on the smoothed starting direction and the principal curvatures along the hatch. A drawback of disregarding the directions in the local neighbourhood, is possibly overlapping hatches; especially when traces get longer.

$$\vec{h}_0 = \vec{s}_i$$

$$\vec{h}_{j+1} = \begin{cases} (1-w_j)\vec{h}_j + w_j\rho_j\vec{k}_{1_j} & ,|\vec{k}_{1_j}\cdot\vec{h}_j| > |\vec{k}_{2_j}\cdot\vec{h}_j| \\ (1-w_j)\vec{h}_j + w_j\rho_j\vec{k}_{2_j} & ,|\vec{k}_{1_j}\cdot\vec{h}_j| \leq |\vec{k}_{2_j}\cdot\vec{h}_j| \end{cases} \tag{5.2}$$

$$w_j = \frac{j}{s}$$

Equation 5.2 describes the change of directions of the segments, for a single hatch starting at some particle position p_i. The equation shows that the direction at the starting position of the hatch $\vec{h}_0$ equals the smoothed field direction $\vec{s}_i$. Subsequently, for each new segment the direction $\vec{h}_{j+1}$ is weighted by the reliability ρ_j of the surface. Furthermore, the distance from the starting-point of the hatch plays a role. This is achieved by including the weighting factor w_j, where j is the index of the segment direction and s denotes the total number of hatch segments. For the segments further away from the starting-point, the direction of the previous hatch becomes less important. This results in a hatch that starts in a rather fixed direction, but is gradually allowed to move more freely. During the trace, for each new segment of the hatch, the best curvature direction is chosen, based on the direction of the predecessing segment. In practise it might be necessary to incidentally invert the direction of the curvature, in order to smoothly trace the hatch.

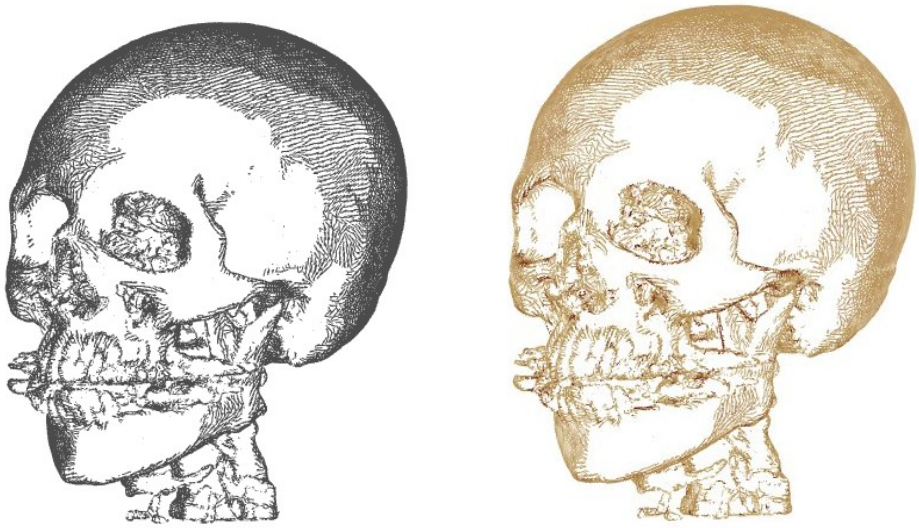
The GPU-approach of the second stage is nearly identical to the first stage. Again the GPU-pipeline is triggered for each particle. In contrast to the first stage, the smoothed field directions are provided as an input, apart from the volume data and the texture containing the transformed weights. The geometry shader extension is used to trace the hatches. Similar to the direction-based approach, geometry is created starting from each particle position, resulting in a line-strip that resembles the hatch. When a hatch is generated, a transform feedback operation stores the line-strips into a new texture-buffer.

Stage 3: Visualise the generated hatches

The final stage of the curvature-based approach, visualises the generated hatches. This stage is completely identical to the visualisation stage of the direction-based curvature approach. The texture buffer containing the hatches, is given as input to the GPU-pipeline. Starting from each particle position, the hatch is rendered to screen. This stage allows to change the appearance of the hatches, by varying the colour and line thickness. The result of the curvature based hatching, using cross-hatches and lighting, is depicted in figure 5.18a.

Curvature colour mapping for hatches

For this last stage of the curvature-based hatching approach, an additional feature was added to VolFliesGPU, which was not present in the VolumeFlies framework. The real-time curvature information can be used to shade the hatches. A two-dimensional transfer function was created, defining the colour of the hatch, based on the principal curvature values (κ_1 and κ_2) at the starting point of the hatch. As a result, areas with high curvature are emphasised, which was also observed by Kindlmann et al. [17]. This colouring approach could be pre-computed, by storing the colour for each hatch in a buffer. However, currently the colouring is applied for each render frame, demonstrating that indeed the curvature estimation operates in real-time. The result of this colouring technique is depicted in figure 5.18b. Observe the darkened areas around the eyesockets and the cheekbones.



(a) Curvature-based hatching, using cross-hatches and lighting

(b) Curvature-based hatching, using cross-hatches, lighting and colour transfer-function

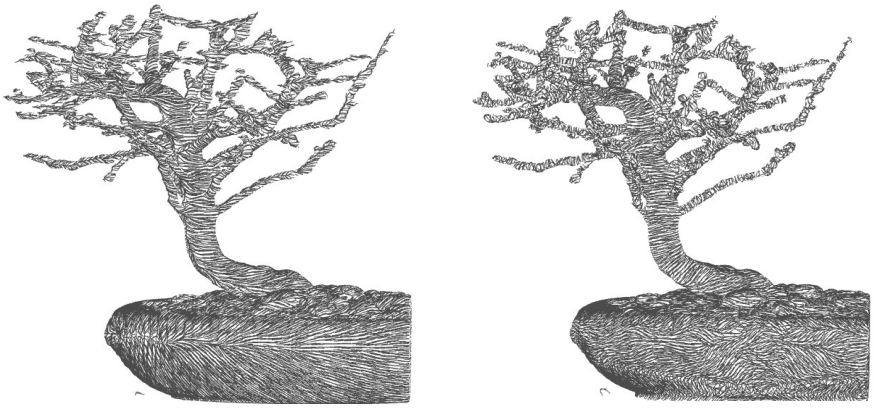
Figure 5.18: Curvature-based hatching

The GPU-based implementation of this module, is slightly changed, compared to the VolumeFlies approach. The complexity of the module was decreased, for performance reasons (table 5.7).

Complexity VolumeFlies:	
Smooth field directions	$\mathcal{O}(m\ b)$ with h = # hatch segments, b = average # neighbours
Hatch generation	$\mathcal{O}(d\ m\ h\ b)$
Hatch visualisation	$\mathcal{O}(d\ m\ h)$
Complexity VolFliesGPU:	
Smooth field directions	$\mathcal{O}(m\ b)$ with h = # hatch segments, b = average # neighbours
Hatch generation	$\mathcal{O}(d\ m\ h)$
Hatch visualisation	$\mathcal{O}(d\ m\ h)$

Table 5.7: Curvature-based hatching complexity

To conclude this section, a comparison is made between the direction-based and the curvature-based hatching approach. The hatches in figure 5.19b follow the direction of the principal curvature directions, whereas the hatches in figure 5.19a follow a fixed direction over the surface. The bonsai dataset contains several regions, with particular high curvature. For instance the trunk and branches of the bonsai tree are highly curved. In these regions, the hatches in figure 5.19b actually follow the circumference of the trunk or branches, as opposed to the fixed hatch direction depicted in figure 5.19a.



(a) Direction-based hatching (b) Curvature-based hatching

Figure 5.19: Comparison with direction-based hatching

5.5.4 Contours

The last module, incorporated in VolumeFlies, visualises the contours of features in the volume. Recall from the description in section 4.3, that the contours are based on the hatching approach. Segments are hatched along the contour, starting at the particle positions near a contour.

In contrast to the hatching approach, the contours can not be pre-computed. Visualisation of the contours is view-dependent, which implies that the contours need to be generated for each frame. As a result, the GPU based approach consists of only one stage, which directly generates and visualises the contours. This stage is depicted in figure 5.20.

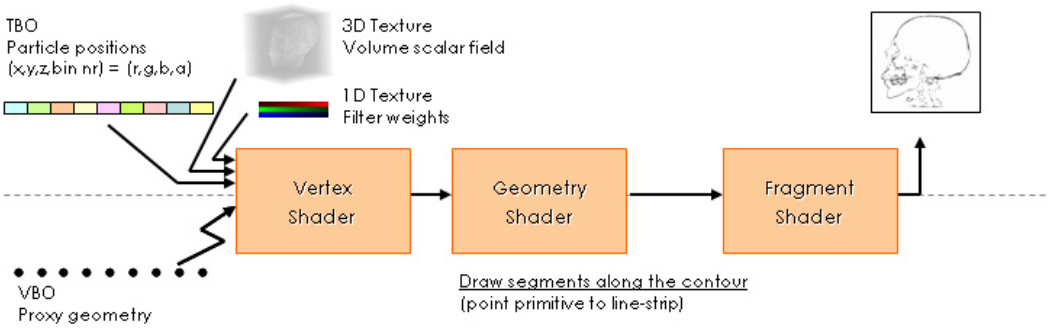


Figure 5.20: Contours on GPU

The hatch direction $\vec{d}$ was defined in equation 4.19. This direction is based on the change of the normal vector, perpendicular to the view vector. A good estimate of this direction $\vec{d}$, requires principal curvature information. Again, the real-time curvature methods is applied, based on the general GPU-approach (figure 5.1). Similar to other modules that incorporate the curvature estimation, a texture with the transformed weights is needed as an input.

The GPU-pipeline is triggered for each particle position, after with the geometry shader thread is responsible to create the hatch segments along the contour. First the particle that reside within a reasonable distance from the contour need to be determined. Next, the segments are traced in two directions, starting from the particle position. The results are directly rendered to screen, and will be updated per frame.

Figure 5.21 depicts the contours results, for both a synthetic data set and the CT head dataset. For these results, the constant width approach was applied, putting a threshold on τ (equation 4.20). Be aware that obtaining good results for the contours, involves a considerable amount of parameter tuning. Fortunately, the effect of the parameters can be interactively inspected.

(a) Contours for a synthetic dataset (b) Contours for the CT head dataset

Figure 5.21: Contours approach

The GPU-based implementation of this module requires no extra steps. The complexity of the module is equal to the complexity of the VolumeFlies approach (table 5.8).

Complexity VolumeFlies:	$\mathcal{O}(m\ h)$ with $h = \#$ hatch segments
Complexity VolFliesGPU:	$\mathcal{O}(m\ h)$ with $h = \#$ hatch segments

Table 5.8: Contours hatching complexity

Chapter 6

Results

In the previous chapter the GPU-based modules of the VolFliesGPU framework, were presented. The results of this framework need to be evaluated for both performance and functionality.

First of all, the performance of the framework will be evaluated. The main objective for this thesis was to speed-up the VolumeFlies framework. Therefore the performance results are of great importance to the overall success of the project. The performance measures of VolFliesGPU will be compared to VolumeFlies framework, after which the performance gain of the GPU-based approach can be evaluated.

Subsequently, the project requirements will be verified, in order to determine to what extend the project goals are fulfilled. The functionality included in VolFliesGPU will be described per requirement. Both the functional, and the extra-functional requirements, were introduced in chapter 3.

6.1 Performance evaluation

Recall from section 5.1, that measuring performance is a non-trivial task because of the non-deterministic behaviour of the operating system. For the performance analysis of the VolumeFlies framework, a profiling approach was taken, based on the CPU tick counter. This approach is subject to some precision errors. Therefore, all presented results consists of an average of a series of measurements. We have chosen to execute at least three runs per measurement.

The performance measurements for VolFliesGPU are executed similar to the measurements presented in section 5.1. The results are presented in table 6.1. Some modules are altered for speed, others are newly added to VolFliesGPU. These modules are marked with an '*'.

VolFliesGPU General Information:		framerate	speed-up
Processor	Intel Core 2 Duo 2.4 GHz; 3GB RAM		
Dataset	CT Head (256^3 voxels x 8 bits)		
Number of particles	60.000		
Initialiser:			
Load volume data	0.14 sec		28x
Brute-force particle placement	0.14 sec		70x
Behaviour:			
Particle redistribution	4.00 sec (25 steps)		63x
Particle redistribution (with stop criterion) *	2.58 sec (15 steps)		
Visualiser:			
Cone splatting	0.03 sec (per frame)	34 fps	11x
Stippling (Density-based)	0.0006 sec (per frame)	1717 fps	245x
Stippling (Scale-based)	0.0008 sec (per frame)	1135 fps	162x
Hatch smooth field directions	1.61 sec		4x
Hatch generation (Direction-based)	0.07 sec		752x
Hatch generation (Curvature-based)	0.29 sec		183x
Hatch visualisation	0.06 sec (per frame)	18 fps	18x
Hatch visualisation (Curvature colour) *	0.06 sec (per frame)	17 fps	
Contours	0.07 sec (per frame)	15 fps	324x

* Not included in VolumeFlies

Table 6.1: VolFliesGPU performance

From the results above, we observe that the overall performance is drastically improved, when comparing to VolumeFlies (table 5.1). The comparison between both frameworks is graphically depicted in figure 6.1. Be aware that results of modules that consume more than 25 seconds are truncated, in order to show the measurements at a comparable scale.

The first step, necessary for each module, is uploading the volume data. For the GPU-based approach this comprises a data transfer from hard disk to GPU memory. Despite this transfer, loading the volume of 16 megabytes, is executed considerably faster. Subsequently, the brute-force initialisation is performed. This rather elaborate approach, performs exceptionally well. The brute-force initialisation needs to be executed, each time the iso-value is changed. Because of the performance gain in the new approach, the iso-value can now be changed interactively. The new features are located instantly.

One of the main performance bottlenecks in VolumeFlies, was the particle redistribution. The GPU-based approach, without the new stop criterion, is more that 60 times faster, compared to the software-rendered approach. The parameters are configured to match the VolumeFlies approach, as much as possible. Moreover, we observe that including the new stop criterion often decreases the total number of iterations. In this particular example, the stop criterion decreases the execution time with more than a second. In general, the approach with a fixed number of iterations would be faster, when the required number of steps to reach an equilibrium was known on beforehand. Since this number of iterations is usually unknown, the number of fixed iterations has to be increased to guarantee that the particle system will converge. The new stop criterion automatically verifies if the system a near its equilibrium, at the cost of an elaborate reduction operation for each iteration.

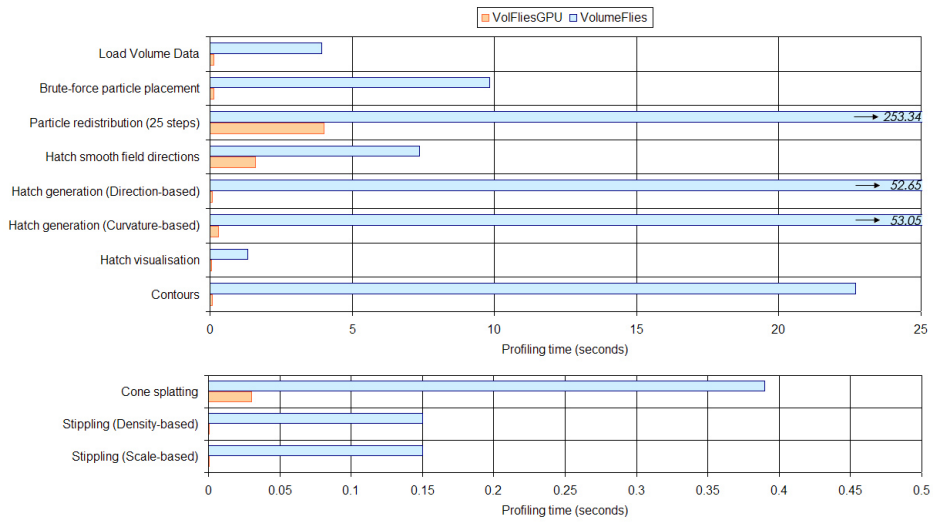


Figure 6.1: Performance comparison

The render times for visualiser modules are in general more time-critical, since the algorithms are executed for each frame. In particular this holds for the cone-splatting module. When the cone-splatting is applied to filter the particles on the visual surfaces, it needs to be combined with other visualisation modules, while maintaining interactive framerates. The 34 fps for only the cone-splatting module is sufficiently fast, to be combined with the illustrative styles.

The cone-splatting approach was optimised, by eliminating the extensive scanning operation through the colour buffer. Recall that the colour buffer, generated by a render-to-texture approach, was used as a lookup table to verify which particles reside on a visual surface. The scanning approach included in VolumeFlies would result in an immense amount of gather operations on the texture buffer. Instead the particle position is projected on the viewing plane, and transformed to texture-space coordinates. As a result, the lookup directly verifies the right location in the colour buffer. In practice a very small region of the texture is considered, surrounding the lookup position. In this way the results become more stable, especially for cones near the edge of a feature, which are almost flat by scaling.

For the stippling modules, we observe that these operations do not require any speed-up. The stippling approaches have a complexity in the order of the number of particles. The workload per particles is limited to a single operation on the appearance of the particle, without addressing any information of neighbouring particles. Furthermore no extra geometry

needs to be created, and no pre-processing passes are necessary. As a result, the entire approach is very suitable for the GPU to execute in parallel. This results in the exceptional framerates of more than 1000 frames per second.

In contrast to the stippling approach, generating and visualising the hatches requires much more computational effort. In section 5.1, the hatching approach was identified as the second major bottleneck in the VolumeFlies framework. The new approach, based on the geometry-shader extensions, shows a significant performance gain of more than 50 seconds. The direction-based hatches are generated within a fraction of a second (0.07 sec), whereas VolumeFlies requires tens of seconds.

The curvature-based hatching approach was also time-consuming in the VolumeFlies framework. Again a new approach was proposed based on the geometry shader, combined with real-time curvature estimation. Recall from section 5.5.3, that the approach requires two stages to actually generate the hatches. The first stage smooths the field of principal curvature directions. This involves calculating the principal curvature for each particle in the system, and subsequently averaging the curvature direction with particles in the local neighbourhood. From the redistribution module, we have learned that interaction with neighbouring particles typically is time-consuming, because it counteracts the parallelism in the graphics card. Nevertheless the redistribution was subject to a considerable performance gain. Therefore a similar strategy was applied to smoothen the field directions. Combined with the faster real-time curvature estimation, again a significant performance gain was achieved. Smoothing the direction field was executed more than 6 times faster compared to VolumeFlies.

The second stage of the curvature-based hatching approach, traced the hatches along the principal curvature directions. This approach was altered, in comparison to the VolumeFlies approach. In VolumeFlies, the tracing of the hatches again considered particles in the local neighbourhood; smoothing the directions for each segment of the hatch. Preserving this approach on the GPU might still result in a performance gain, compared to the software-rendered approach. However, the execution time for generating the hatches might still be in the order of tens of seconds. Instead the trace of the hatch was weighted, based on the reliability of the surface and the distance to the starting particle. This approach only considers the previous segments of the hatch, instead of all directions of various particles in the neighbourhood. As a result, the execution time for tracing the hatches could be reduced to less than 0.3 seconds for the case considered in table 6.1, which is over 180 times faster compared to VolumeFlies.

After the hatches are generated, the results need to be rendered to the screen. The visualisation stage gathers the generated hatches, and renders them using the geometry shader extension. Per particle, more geometry needs to be rendered to the screen. Therefore the visualisation of the hatches is more time-consuming, compared to the stippling approach. Furthermore, the transform feedback currently supports only high precision buffers, which results in large buffers and intensive data handling. Nevertheless the resulting visualisation performs at an interactive speed of 18 frames per second, which was not possible in VolumeFlies. Moreover, after enabling the colouring approach, which calculates the principal curvature at all particle positions for each frame, the framerate drops with no more than 1 frame per second.

Lastly there is the contour module, which requires a slightly different approach. In contrast to the hatching approach, the segments can not be pre-computed. As a consequence, the hatch directions along the contours need to be generated directly for each frame. For the VolumeFlies framework, these contours where far from interactive. Fortunately the use of the

filtering approach has shown to be valuable to the visualisation of the contours. Rendering of a contour segment only needs to be performed for the particles that reside on a visual surface. As a result, the expensive computation of the contour segments can be discarded for all invisible particles. The performance measures exclude the filtering approach, because otherwise the performance gain of the filtering approach would be falsely considered in the performance comparison of the contours module. The VolumeFlies approach performs at less than one frame per second, while the VolFliesGPU approach on average manages to render 15 frames per second.

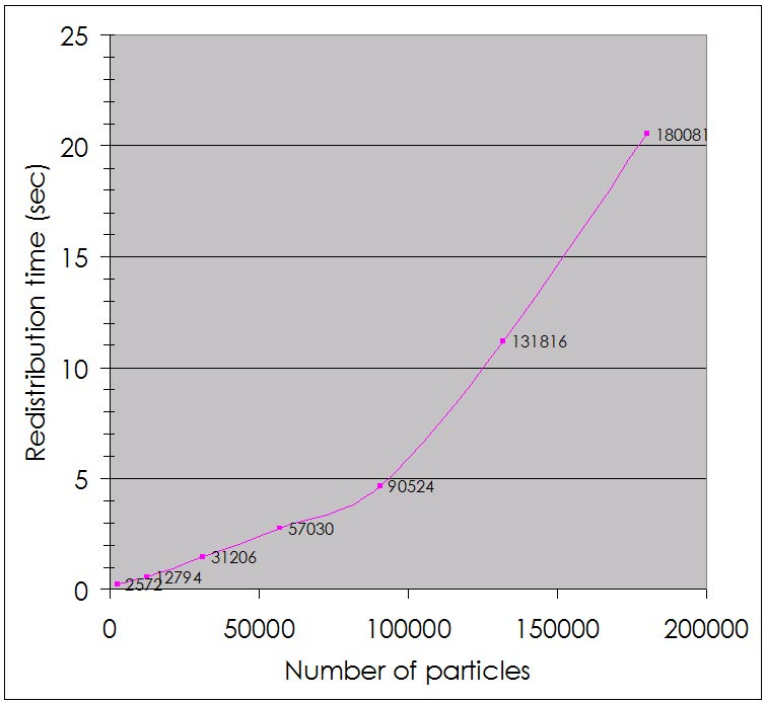
Overall, the GPU-based framework shows a substantial performance gain for each module. The performance bottlenecks, found in the redistribution and the generation of the hatches, were all reduced to execution times in the order of seconds instead of tens of seconds. Furthermore, all visualisation modules perform on average at more than 15 frames per second, which is generally considered to be interactive.

Fortunately, hardly any quality compromises needed to be made to accomplish the performance gain. Most modules implement exactly the VolumeFlies functionality, preserving the same quality. Only for tracing of the curvature-based hatches a different approach was taken, which slightly compromises the resulting quality in order to achieve the desired performance gain. For short hatches, the resulting images will be nearly consistent with the VolumeFlies approach. For longer hatches however, the new approach might lead to overlapping hatches, whereas VolumeFlies will keep them alongside each other.

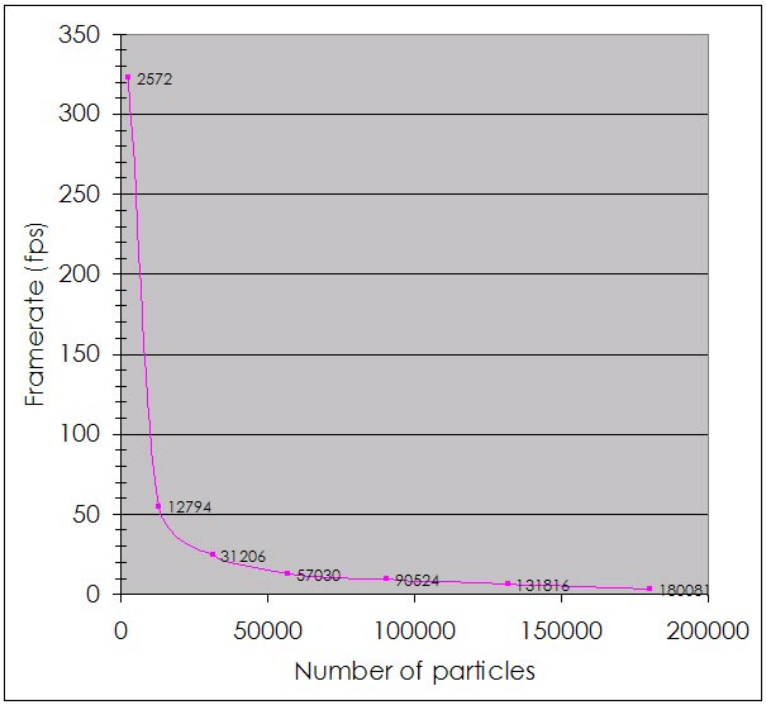
Be aware that combining visualisation styles results in a drop of the framerate, leading to non-interactive depictions of the volume. Only a limited number of particles can be present in the system; typically no more than 200.000. These particles can be positioned at different features, visualising the features with different illustrative styles. The limitation is mainly due to the fact that the transform feedback requires a high-precision buffer, consuming a notable amount of memory. This could be resolved by applying a more elaborate memory management, which swaps the GPU-side buffers.

Finally it is important to observe that the complexity of the algorithms, contained in the modules, are all dependent on the amount of particles. As a consequence, the performance of both the VolumeFlies and the VolFliesGPU framework diminish when increasing the amount of particles in the system. On the other hand, the VolFliesGPU framework is principally independent of the screen resolution. The visualiser modules have to process more fragments for higher screen resolutions, but none of the algorithms is computed with pixel-order complexity. As a result, the performance of the VolFliesGPU framework hardly decreases when enlarging the screen resolution.

Figure 6.2 presents two plots, depicting the change of the performance depending on the number of particles in the system. In plot 6.2a, the duration of the redistribution time is plotted against the number of particles in the system. Observe that the redistribution rapidly consumes more time, when the number of particles in the system becomes larger than approximately 100.000. Similarly plot 6.2b measures the performance of the curvature based hatching, again depending on the total number of particles in the system. The plot clearly shows a drop of the framerate, when more particles are added to the system. When the system contains about 100.000 particles, the visualisation is still interactive. However, further increasing the number of particles leads to framerates that are no longer interactive. In practice, approximately 150.000 is the maximum number of particles that can be visualised at decent framerates. Lastly, table 6.2 shows the hatching results of the configurations used for the framerate measurements in plot 6.2b.

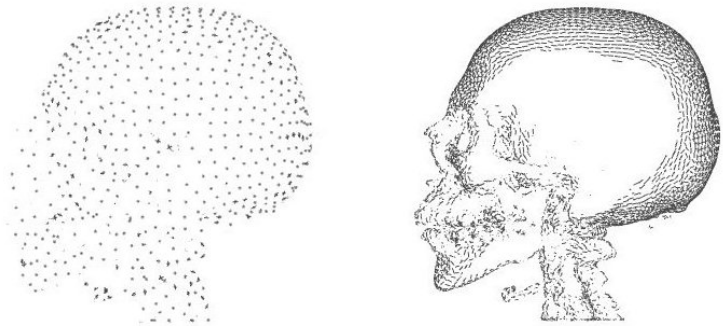


(a) Performance plot (Redistribution time)



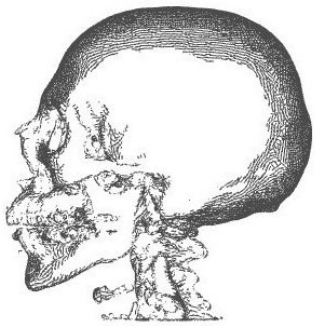
(b) Performance plot (Hatching framerate)

Figure 6.2: VolFliesGPU performance related to the number of particles

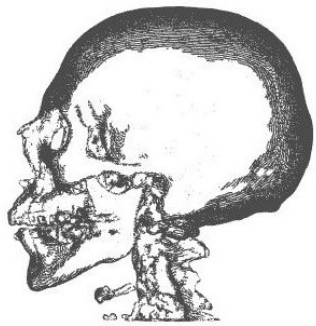


2572 particles

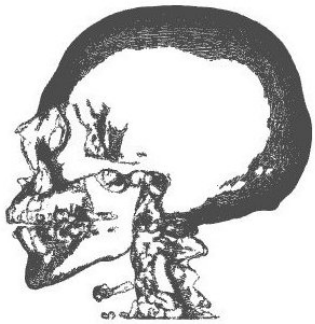
12794 particles



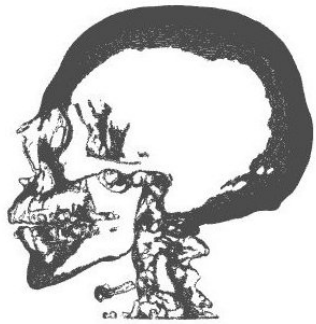
31206 particles



57030 particles



131826 particles



180812 particles

Table 6.2: VolFliesGPU results of different particle densities

6.2 Requirement verification

The VolFliesGPU framework was implemented in the programming language C++. The framework is based on the OpenGL 3D graphics library and the Visualisation ToolKit (VTK). VTK is a special toolkit for 3D computer graphics, which is used as a basis for rendering the illustrative results. Moreover the VTK functionality to interact with the data is incorporated. The framework includes a graphical user interface (GUI), which is based on the QT library. The user interface is depicted in figure 6.3.

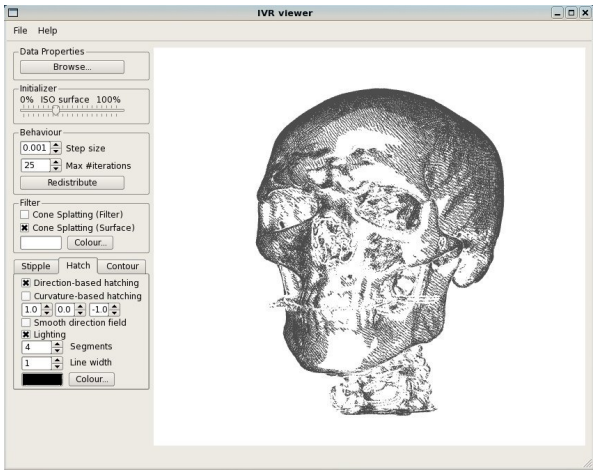


Figure 6.3: VolFliesGPU GUI

In this section the requirements will be verified, starting with the functional requirements. For each of the requirements, listed in appendix A, the result will be briefly described. Comments starting with a '+' generally fulfil the requirement, whereas comments starting with a '-' describe disadvantages or missing features.

fr01:	Loading a 3D dataset to the GPU *must* be possible
+	This requirement is obviously fulfilled, since it is fundamental to the entire GPU-approach.
-	Nevertheless, several improvements can be made. At this point, only 8-bits texture can be uploaded to the volume. Furthermore, it is assumed that the entire volume fits on the GPU memory.
fr02a	Particles initialisation *must* be reimplemented on the GPU (feature location)
+	The feature location is ported to the GPU, preserving the brute-force approach presented in VolumeFlies.
fr02b	Particles redistribution *must* be reimplemented on the GPU
+	The particle-redistribution is ported to the GPU, preserving the functionality from VolumeFlies.
fr03a	Cone-splatting (Hidden surface removal) *must* be reimplemented on the GPU
+	The cone-splatting approach is ported to the GPU, preserving the functionality from VolumeFlies.

fr04a/b	Stippling *must* be reimplemented on the GPU
+	The stippling approaches are ported to the GPU, preserving the functionality from VolumeFlies.
fr04c	Hatching must be reimplemented on the GPU
+	The hatching approaches are ported to the GPU, preserving the VolumeFlies functionality.
-	Density based shading for hatches is currently not available. Only lighting by a two-level threshold.
fr04d	Contours must be reimplemented on the GPU
+	The contour approach is ported to the GPU, preserving the VolumeFlies functionality.
-	Results of the contours can be improved. Good results require much parameter tuning.
fr04e	Other visualisations *should* be added
+	The VolFliesGPU framework includes a general approach to real-time curvature estimation. The curvature information was used to colour the hatches, using a two-dimensional transfer function.
-	No new visualisation modules are added
fr05a/b/c	The framework *must* be written in C++, using VTK. It should have a QT GUI
+	VolFliesGPU was implemented using C++, based on VTK. Furthermore a QT GUI was supplied.
fr05d/e	The framework *should* be integrated with a direct volume renderer / DTI tool
-	Due to timing constraints, VolFliesGPU was not integrated with any other tool.
fr06a	Changing the iso surface *must* be interactive
+	Changing the iso-surface is completely interactive.
fr06b	Visualisation of the dataset *must* be interactive
+	All visualisation modules perform at a framerate greater than 14 fps, which generally is accepted as interactive. By combining illustrative styles, the framerate might decrease to a point, at which the system is no longer interactive.
fr07a	New elements *should* be added to the framework
+	Real-time curvature estimation is added to the framework, based on the approach by Sigg and Hadwiger [29].
-	No new modules were added to the framework
fr07b	New elements *should* be added to the framework
+	A new general GPU-based approach was proposed, suitable for particle systems
-	No new rendering techniques were added to the framework

Overall, we can conclude that the expected functionality is included in VolFliesGPU. The resulting framework comprises a prototype, demonstrating the performance gain that can be obtained by using consumer graphics hardware. Several improvements can still be made, and the framework can be extended with other visualisation styles. Moreover, it should be integrated with a GPU-based direct volume renderer. This integration will show the value of the presented illustrative styles, as a context of more realistic visualisations.

Besides the functional requirements, also a set of extra-functional requirements was specified in chapter 3. These will be verified, similar to the functional requirements.

---

efr01: Performance

\+ All visualisers perform at a framerate above 14 fps. Pre-computation durations are decreased

efr02: Responsiveness

\+ Although responsiveness is a subjective notion, we feel that the system responds naturally. The interaction with the volume is based on the VTK 'trackballcamera' style. This is a efficacious style, which delivers a good response depending on the framerate. (See efr01)

efr03: Maintainability

\+ The VolFliesGPU framework is based on libraries, which are commonly used within the research group. Therefore, maintenance and integration is conceivable. Furthermore, the source code contains a considerable amount of comments, which can be used as documentation by using the 'doxygen' tool (http://www.doxygen.org).

efr04: Reusability

\+ The general VolumeFlies framework is maintained.

efr05: Extensibility

\+ The general VolumeFlies framework is maintained. A new module should derive from the abstract module class.

efr06: Generality

\+ The general VolumeFlies framework is maintained. Moreover, a general GPU-approach is proposed, suitable for particle systems. Furthermore, a general real-time curvature estimation is incorporated.

efr07: Portability

\+ The VolFliesGPUwas successfully compiled for both the Linux (Fedora Core 6) and the Windows XP platforms. Be aware that portability is limited by the GPU. Only the latest NVidia 8-series graphics cards suffice.

From the above evaluation, we observe that all extra-functional requirements are taken into consideration during the development of the GPU-based framework. In many cases, extra-functional requirements are hard to quantify. Therefore, it is difficult to determine to what extend a extra-functional requirement is fulfilled. In general the VolFliesGPU framework is based on a general and extensible approach that was proposed by Busking for VolumeFlies. Moreover, VolFliesGPU relies on libraries that have shown to be valuable in various other projects.

We conclude this chapter with table 6.3, showing various visualisations that can be created with the VolFliesGPU framework. Several styles were combined in these images, possibly on different implicit surfaces in the volume. Many other variations are possible.

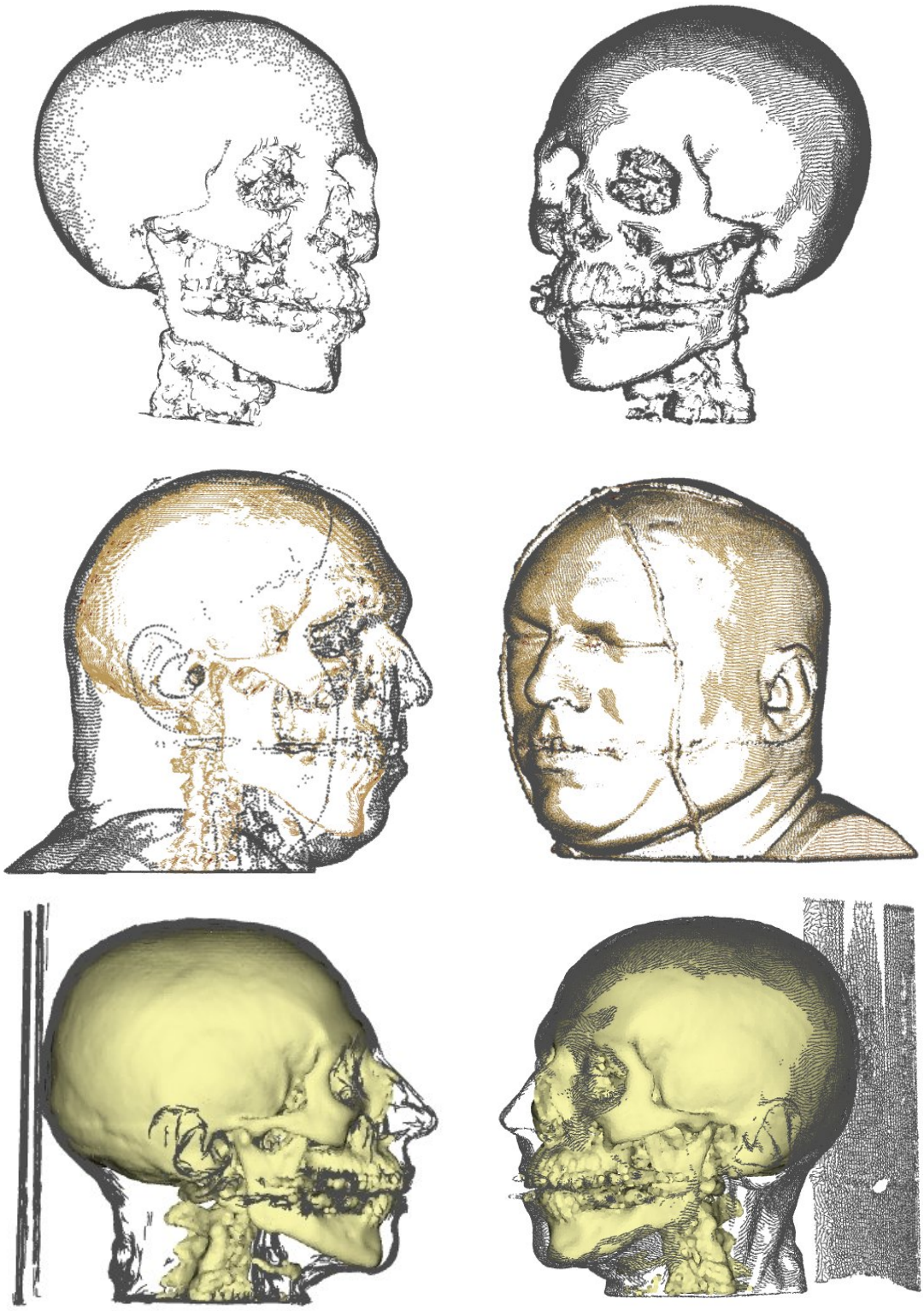


Table 6.3: Examples of illustrative visualisations

Chapter 7

Conclusions

The previous chapter presented the overall results of the project. The performance of VolFliesGPU was discussed and compared to the performance of the software-rendered VolumeFlies. All modules were subject to a considerable performance gain. In comparison to VolumeFlies, the execution time of the pre-computation steps was substantially improved, and the individual visualising modules operate at an interactive framerate, while assuming similar parametrisation. Furthermore, the requirements of the system were verified with the actual results of the framework. Most requirements were fulfilled, possible leaving some space for improvement. Unfortunately the integration of the illustrative visualisation with other volume rendering approaches was not feasible within the time span of this project.

This chapter recapitulates the material presented in the previous chapters. First of all, the contribution to the existing VolumeFlies framework is listed. Subsequently the limitations of the current VolFliesGPU framework will be addressed. Finally, this chapter concludes with a list of recommendation and future work, that may be of interest for further research.

7.1 Contribution

The main objective for this project, was to speed up the VolumeFlies framework, using the capabilities of modern consumer graphics hardware. The following contributions result from the process of developing the new GPU-based framework:

- The main contribution of this project, is the general approach for fast GPU-based computations, presented in section 5.2. In general we have investigated the GPU as a platform for fast computations. The capabilities of the latest graphics cards were studied, and performance measures were evaluated. VolFliesGPU delivers a completely GPU-based framework implementing the VolumeFlies functionality, based on the general GPU approach. As a result, the volume can be inspected interactively, while changing illustrative styles and their parameters.

- VolFliesGPU relies on a particle system for illustrative visualisation. The approach of the new GPU-based particle system is generic, and can be applied to improve the performance for many other particle systems. Currently, the only restriction for adopting this GPU-based particle system approach, is the limited amount of graphics card that support the required functionality. This is due to the fact that the new approach exploits

techniques that are exclusively contained in the latest graphics cards. Fortunately, new graphics cards will likely preserve or improve the currently applied techniques.

- Furthermore, VolFliesGPU incorporates a new implementation of the real-time curvature estimation, presented by Sigg and Hadwiger in the GPU Gems II book [29]. The GPU-based principal curvature estimation is included as a generic shader program, which can be adopted by other applications. The new implementation, which is based on the general GPU approach (figure 5.2), uses a higher level shading language called GLSL. This results in a more accessible and maintainable implementation, compared to the former GPU assembly code. The GLSL implementation of the curvature estimation requires substantially more assembly operations. The unified shader model (figure 2.8), together with the increase of computational power in the latest graphics cards, largely compensates for the increase in operations. The curvature-based shading of the hatches, using a two-dimensional transfer function (figure 5.18b), demonstrates that the new implementation of the curvature estimation still operates in real-time.

7.2 Limitations

Despite the significant performance gain in the VolFliesGPU framework, there are still some limitation. These limitation are listed below.

- This prototype implementation currently includes a rather modest memory management. Buffers are allocated reasonable sizes, and unwanted buffers are removed if necessary. However, since the transform feedback requires high precision buffers, the framework is limited with respect to the total amount of particles. A more advanced memory model could resolve this problem, by timely swapping in the parts of the buffers, required by the GPU. Furthermore, new developments of the transform feedback extension could provide a solution to free memory for an extra particle sets on different features.

- At this point, only 8-bits volumes are allowed that are smaller than 512^3 voxels. Currently, support for 16-bit volumes is not yet implemented in VolFliesGPU, but there seems to be no technical limitation. The volume is restricted to 512^3 voxels, only for memory reasons. Performance of the framework mainly depends on the number of particles in the system, except for the brute-force initialisation. As a result, visualising features of a 256^3 volume might perform worse than visualising a 512^3 dataset, depending on the size the features in the volume.

- Considering the visualisation approaches, there are some limitations. For instance the filtering approach using the conesplats provides an instable result near the contours of the feature. When the particles are on the edge of turning away from the viewer, the particles start blinking, because the colour buffer lookup gives varying results for the heavily scaled cones. Furthermore, varying sizes of parts of the feature make it hard to determine a general cone radius. In practice this leads to a cone radius, which is rather large for small parts of the feature. As a result, some particles are occluded unintentionally. Lastly, there is no good estimate to scale the depth of the cones, which might result in particles 'bleeding' through the visual surface.

- Furthermore, there are some limitations to the current outlines approach. The current approach requires a considerable amount of parameter tuning, in order to come to a good contour visualisation. The current approach also does not distinguish between contours, suggestive contour and silhouettes.

- Lastly there are some limitations in GPU-based approaches in general. The developments in the graphics card industry is rapidly evolving. Therefore, currently used extensions might change or even get outdated soon. However, considering the latest developments, and the amount of interest that is shown in extensions such as the geometry shader and the transform feedback, it is likely that these extensions will be included in a future shader model. In that case, only implementational aspects need to be adjusted, and possibly new improvements can be exploited. At present, only NVidia graphics cards of the G80 series are supported.

7.3 Recommendations & Future work

To conclude this chapter, a series of recommendations and possible research directions for future work are listed in this section.

- In the previous section, the memory usage was identified as a limitation of the VolFliesGPU framework. The GPU-based approach requires many duplicates of several buffers in the memory of the GPU. More efficient handling of these buffers would be a valuable extension of the current framework.

- Furthermore, some limitations of the current cone-splatting approach for visual surface calculation were observed. So far these issues could not be resolved. New scaling approaches could be investigated to improve the results. One could think of scaling the cones based on the principal curvature at the particle positions. Moreover, a scaling of the depth of the cones could be introduced that takes into account that there might be nearby surfaces that should be occluded. Lastly, the orientation and scaling of the cones might be adjusted for use with a perspective projection.

- Although the contours yields good result, after tuning the parameters, other approaches might be considered. For instance the approach presented by Burns et al. [4] provides a method that distinguishes between contours, suggestive contours and silhouettes. The presented approach is rather elaborate, although they achieve interactive framerates. Perhaps a similar approach could be adopted in the VolFliesGPU framework, based on the general GPU approach.

- The presented illustrative styles can serve as context for other volume visualisation approaches. Integrating VolFliesGPU with a direct volume renderer, results in images where the region of interest can be visualised with a more photorealistic approach, whereas the context is presented by one of the illustrative styles.

- Furthermore, the illustrative styles might be applied to multi-field data sets, such as diffusion tensor imaging (DTI) scans. In general, particle systems are flexible, and could be useful for visualising features within such volumes. Currently, the particle redistribution is optimised to evenly distribute on an implicit surface, by minimising an

energy function. There may be ways to make the particles follow the diffusivity in the volume, by changing the redistribution steps and the energy function.

- Busking observed several aspects that might be subject for future work. A known issue with the current approach, is the fact that the number of particles does not scale with the zooming factor. While zooming out, the illustrative depictions appear darker, since the amount of point primitives or hatches appear more dense when they are rendered on a smaller area of the viewing plane. A hierarchical particle set was proposed in the VolumeFlies master's thesis [5], folding and unfolding sets of particles to adjust the perceived density. Unfortunately, initial attempts to create such a hierarchy delivered disappointing results.

- Furthermore, Busking observed that the work by Meyer et al., includes several other particle related methods, that could be adopted in the framework. For instance, the density of the particles could be adapted to local surface properties, such as the principal curvature. This might help visualising small structures in the dataset.

- Lastly, Busking observed that the use of animation is not yet included. Particle systems are quite suitable for creating dynamic visualisation. The GPU-based approach delivers new possibilities to animate particles in real-time. An animation approach could be implemented as a behaviour of the framework, similar to particle flow simulations.

List of Figures

Bibliography

[1] Forman S. Acton. *Numerical methods that work*. pub-MATH-ASSOC-AMER, pub-MATH-ASSOC-AMER:adr, 1990. Cover title: Numerical methods that usually work. Updated and revised from the 1970 edition published by Harper and Row.

[2] Pascal Barla, Joëlle Thollot, and Lee Markosian. X-toon: An extended toon shader. In *International Symposium on Non-Photorealistic Animation and Rendering (NPAR)*. ACM, 2006.

[3] Stefan Bruckner and Meister Eduard Gröller. Volumeshop: An interactive system for direct volume illustration. In H. Rushmeier C. T. Silva, E. Gröller, editor, *Proceedings of IEEE Visualization 2005*, pages 671–678, October 2005.

[4] Michael Burns, Janek Klawe, Szymon Rusinkiewicz, Adam Finkelstein, and Doug DeCarlo. Line drawings from volume data. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 24(3):512–518, aug 2005.

[5] Stef Busking. Volumeflies - a smart-particle-inspired framework for illustrative volume rendering. Master's thesis (TU/e Eindhoven, The Netherlands), 2006.

[6] Stef Busking, Anna Vilanova, and Jarke J. van Wijk. Particle-based non-photorealistic volume visualization. *The Visual Computer Journal (to appear)*, 2007.

[7] Nelson S.-H. Chu and Chiew-Lan Tai. Moxi: real-time ink dispersion in absorbent paper. *ACM Trans. Graph.*, 24(3):504–511, 2005.

[8] Derek Cornish, Andrea Rowan, and David Luebke. View-dependent particles for interactive non-photorealistic rendering. In B. Watson and J. W. Buchanan, editors, *Proceedings of Graphics Interface 2001*, pages 151–158, 2001.

[9] NVidia Corporation. Nvidia geforce 8800 gpu architecture overview, 2006.

[10] Balzs Csbfalvi, Lukas Mroz, and Helwig Hauser. Fast visualization of object contours by non-photorealistic volume rendering. Eurographics, 2001.

[11] Doug DeCarlo, Adam Finkelstein, Szymon Rusinkiewicz, and Anthony Santella. Suggestive contours for conveying shape. *ACM Transactions on Graphics (Proc. SIGGRAPH)*, 22(3):848–855, jul 2003.

[12] Feng Dong, Gordon J. Clapworthy, Hai Lin, and Meleagros A. Krokos. Nonphotorealistic rendering of medical volume data. *IEEE Computer Graphics and Applications*, 23(4):44–52, 2003.

[13] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Aaron E. Lefohn, Christof Rezk Salama, and Daniel Weiskopf. Real-time volume graphics. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 29, New York, NY, USA, 2004. ACM.

[14] Bruce Gooch and Amy Gooch. *Non-Photorealistic Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2001.

[15] Markus Hadwiger, Christian Sigg, Henning Scharsach, Katja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Comput. Graph. Forum*, 24(3):303–312, 2005.

[16] Aaron Hertzmann and Denis Zorin. Illustrating smooth surfaces. In Kurt Akeley, editor, *Siggraph 2000, Computer Graphics Proceedings*, pages 517–526. ACM Press / ACM SIGGRAPH / Addison Wesley Longman, 2000.

[17] G Kindlmann, R Whitaker, T Tasdizen, and T Möller. Curvature-based transfer functions for direct volume rendering: Methods and applications. In *Proceedings of IEEE Visualization 2003*, pages 513–520, October 2003.

[18] Peter Kipfer, Mark Segal, and Rüdiger Westermann. Uberflow: a gpu-based particle engine. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 115–122, New York, NY, USA, 2004. ACM.

[19] Jan J. Koenderink and Andrea J. van Doorn. Surface shape and curvature scales. *Image Vision Comput.*, 10(8):557–565, 1992.

[20] Jens Kruger, Peter Kipfer, Polina Kondratieva, and Rüdiger Westermann. A particle system for interactive visualization of 3d flows. *IEEE Transactions on Visualization and Computer Graphics*, 11(6):744–756, 2005.

[21] T. Van Laerhoven, J. Liesenborgs, and F. Van Reeth. Realtime watercolor painting on a distributed paper model, 2004.

[22] I. Linnemeijer. Entertainment and media outlook towards 2010: Gaming, 2006.

[23] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM.

[24] A. Lu, C. Morris, D. Ebert, P. Rheingans, and C. Hansen. Non-photorealistic volume rendering using stippling techniques, 2002.

[25] M. Meyer, P. Georgel, and R.T. Whitaker. Robust particle systems for curvature dependent sampling of implicit surfaces. In *In Proceedings of the International Conference on Shape Modeling and Applications (SMI)*, pages 124–133, June 2005.

[26] M. Meyer, B. Nelson, R. Kirby, and R. Whitaker. Particle systems for efficient and accurate high-order finite element visualization. *IEEE Trans Vis Comput Graph*, 13(5):1015–26.

[27] Z. Nagy, J. Schneider, and R. Westermann. Interactive volume illustration, 2002.

[28] A. Pang and K. Smith. Spray rendering: Visualization using smart particles. pages 283–290.

[29] Matt Pharr and Randima Fernando. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation (Gpu Gems)*. Addison-Wesley Professional, 2005.

[30] Adrian Secord. Weighted voronoi stippling. In *NPAR '02: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 37–43, New York, NY, USA, 2002. ACM.

[31] Christian Sigg. Representation and rendering of implicit surfaces. PhD Dissertation, 2006.

[32] Dirk vanden Boer. General-purpose computing on gpus. Master's thesis, 2005.

[33] Georges Winkenbach and David H. Salesin. Computer-generated pen-and-ink illustration. *Computer Graphics*, 28(Annual Conference Series):91–100, 1994.

[34] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. Surface splatting. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 371–378. ACM Press / ACM SIGGRAPH, 2001.

Acknowledgements

I would like to express sincere gratitude to both my supervisors, dr. A. Vilanova and dr. ir. H. van de Wetering, for their intellectual support and continual encouragment through my studies. Their intensive guidance gave me the opportunity to grow as a student and engineer. This thesis was made possible, thanks to their unremitting effort.

I also would like to thank ir. S. Busking for his technical support. His delibarate advising brought in meaningful insights that helped me to accomplish my goals.

Furthermore, I would like to express my appreciation to dr. ir. M. Hadwiger, dr. ir. C. Sigg and dr. ir. M. Meyer, for their cordial support and interest in my work. Their sound advice delivered a substantial contribution to my work.

Special thanks to all my collegues from the BMIA research group. Their advice and assistance resulted in new views on a variety of topics, helping me to achieve my objectives. Furthermore, I wish to thank them for all for the pleasurable times we spend together on various social occasions.

I wish to thank my parents, who provided the item of greatest worth - opportunity. Thank you for standing by me through many trials and decisions of my educational carreer.

Appendix A: Requirements

Number	Functional requirement
fr01	Loading a 3D dataset to the GPU *must* be possible
fr02	Particle system
fr02a	Particles initialization *must* be reimplemented on the GPU(feature location)
fr02b	Particles redistribution *must* be reimplemented on the GPU
fr03	Filtering
fr03a	Cone-splatting (Hidden surface removal) *must* be reimplemented on the GPU
fr04	Rendering
fr04a	Stippling (Scale based) *must* be reimplemented on the GPU
fr04b	Stippling (Density based) *must* be reimplemented on the GPU
fr04c	Hatching *must* be reimplemented on the GPU
fr04d	Contours *must* be reimplemented on the GPU
fr04e	Other visualization *should* be added
fr05	Integration
fr05a	The framework *must* be implemented in the programming language C++
fr05b	The framework *should* be using a QT graphical user interface
fr05c	The framework *must* be based on the VTK library
fr05d	The framework *should* be integrated with a direct volume renderer
fr05e	The framework *should* be integrated with the DTI tool
fr06	Interaction
fr06a	Changing the iso surface *must* be interactive
fr06b	Visualisation of the dataset *must* be interactive
fr07	Renewal
fr07a	New elements *should* be added to the framework
fr07b	New insights *should* be added to the framework

Table 1: Functional requirements

Number	Extra-functional requirement	Description
efr01	Performance	The framework should on average perform at interactive framerates ($\geq 10fps$, soft real-time)
efr02	Responsiveness	The framework should have an intuitive response to the user. This includes both mouse and keyboard interaction with the application.
efr03	Maintainability	The framework should be maintainable within the research group. Documentation and source code comments are required
efr04	Reusability	The framework should be modular. Modules can be reused at a later stage.
efr05	Extensibility	The framework should allow easy extensions. Adding new IVR techniques should be relatively easy.
efr06	Generality	The framework should provide some general components that can be reused at a later stage.
efr07	Portability	The framework should be cross-platform

Table 2: Extra-functional requirements

Appendix B: Software architecture

This appendix provides a brief overview of the VolFliesGPU class library. The main structure is preserved from the VolumeFlies framework, found in Buskings master's thesis [5]. The architecture, depicted in figure 1, is considerably simplified. Only the most important classes and function headers are included. The prefix 'IVR' is used for each class, denoting that it is part of the Illustrative Volume Rendering framework.

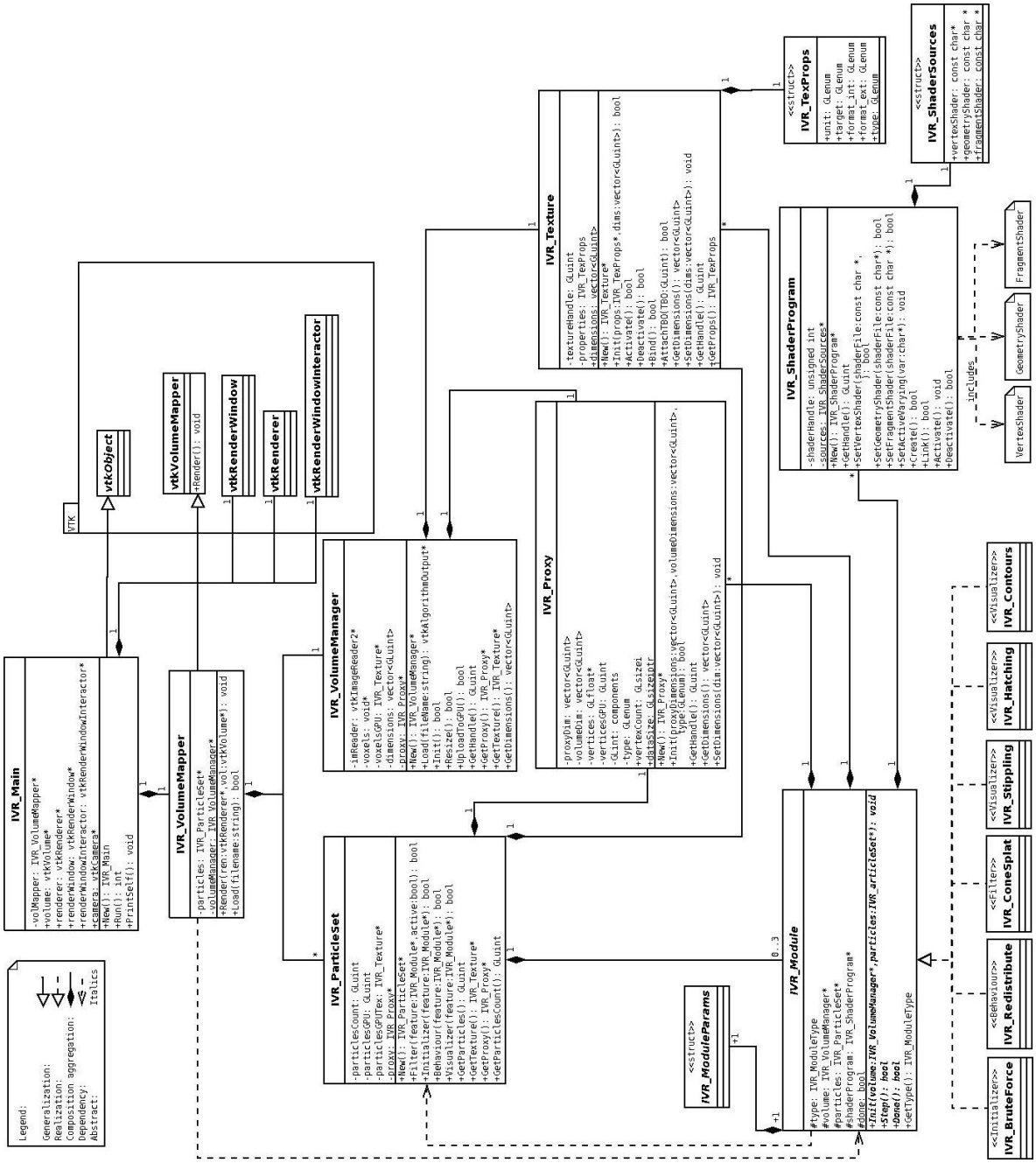


Figure 1: Software architecture

The classes presented in the simplified architecture diagram (fig. 1) will be described below.

IVR_Main
The first class that needs to be instantiated is the main class. This class is responsible to start the execution of the program. This involves initialising the VTK render environment and eventually starting the render loop. VTK provides the necessary rendering environment, such as a RenderWindow, Renderer and RenderWindowInteractor.

IVR_VolumeMapper
The IVR_Main object will instantiate an IVR_VolumeMapper class. This class forms the main component, connecting the volume information to the render engine. The IVR_VolumeMapper contains the main render loop, which outputs the results to the render window. Two modules need to be instantiated, providing the basis for the visualisation. Firstly, this is the class that keeps track of the volume IVR_VolumeManager. Secondly, this is the class that maintains the particles IVR_ParticleSet.

IVR_VolumeManager
The IVR_VolumeManager object contains necessary functionality to work with volume data. The volume manager is capable of reading raw volume data from file. The volume properties, such as dimensions, precision and scaling, are available through the volume manager. Furthmore, the volume managers support uploading a volume to GPU memory, using functionality from the OpenGL API.

IVR_ParticleSet
Visualisation in the VolFliesGPU framework is based on a particle system. The set of particles is described by the IVR_ParticleSet class. This class can by instantiated multiple times, creating multiple particle sets for the visualisation. Each particle set contains an individual rendering pipeline, that processes the available modules (with IVR_Module as base class) for that particular particle set. Furthermore, a IVR_ParticleSet object can transfer the set of particles to the GPU memory.

IVR_Module
This abstract base class provides the general functionality, that should be represented in each of the modules in the framework. Each new module that will be added to the framework needs to derive from this class. A module mainly consists of three stages. First, a module needs to be initialised. Relevant objects can be created, and parameters can be set. After the initialisation, the main functionality of the module will be executed per step. During the main render loop, executed by the volume mapper, the module functionality will be executed for each render frame. After each render frame, the module will be checked to see whether its task is finished. The last stage can put the module to a state, indicating the module is done with its work.

IVR_ModuleParams
Each module contains a series of user parameters. These should be available in some uniform way. Therefore, the abstract class IVR_Module contains a structure IVR_ModuleParams, which needs to be overwritten by a derived class. A uniform approach for the user parameters could be used as a fundament for a plug-in approach in the future.

IVR_ShaderProgram
In constrast to the modules in VolumeFlies, the modules in VolFliesGPU are implemented based on the GPU. Therefore, the IVR_ShaderProgram class provides the necessary functionality to develop a module for the GPU, encapsulating the OpenGL functionality. The IVR_ShaderProgram keeps track of source code, written in the GLSL shader language, and reads them from file. Furthmore, shader programs can be activated and deactived by means of this class.

IVR_ShaderSources
The IVR_ShaderSource structure keeps track of the location of the shader code source files. This structure is used by IVR_ShaderProgram.

IVR_Texture
For many GPU-based computations, textures are required. This class encapsulates the OpenGL texture functionality, which eases the uses of textures. The class keeps track of properties, such as dimensions and precision of the texture. Furthermore, this class support to bind a texture buffer object to this texture.

IVR_TextureProps
The main texture properties, defined by the OpenGL standard, are gathered in the IVR_TextureProps structure. This structure serves the IVR_Texture class.

IVR_Proxy
The general GPU approach requires a proxy geometry, that triggers the execution of the pipeline. This proxy geometry is maintained by the IVR_Proxy class, which supports both 1D, 2D and 3D proxy geometries. New proxy geometries can be created, and uploaded to the GPU. At a later stage, the proxy geometry can be resized.

Appendix C: Particle redistribution (Pseudo code)

```
Stage 1: Sorting particles → Odd even merge sort

COMPL.: 𝒪(n log₂²(n) + n log₂(n)) with n=#particles
INPUT:  source particle buffer (Texture Buffer Object bound to TEXTURE2)
OUTPUT: if #passes%2=0 source      particle buffer (Texture Buffer Object unbound)
        if #passes%2=1 destination particle buffer (Texture Buffer Object unbound)

(1a)  Odd even merge sort (stages) CPU
SET #passes =  1/2 2log₂(#particles) + 1/2 2log(#particles)

FOR each pass until #passes
  DECREMENT pass
  IF pass < 0
    INCREMENT stage
    pass = stage
  ENDIF
  execute pass (1b)
ENDFOR

(1b)  Odd even merge sort (pass) GPU
SET p2stage       = scale stage (x2)
SET p2pass        = scale pass  (x2)
SET sortScope     = sorting scope for current stage       (2 x p2stage)
SET passScopeLow  = lower bound of scope for current pass (p2pass%p2stage)
SET passScopeHigh = upper bound of scope for current pass (sortScope  passScopeLow  1)

FOR each particle PARALLEL

  SET self        = get current particle from buffer
  SET compare     = 0
  SET posInRange  = particle position within range to merge

  IF posInRange not within the scope for current pass
    SET compare = 0
  ELSE
    IF   posInRange is on the left  side of the partnership
      SET compare =  1
    ELSE posInRange is on the right side of the partnership
      SET compare = -1
    ENDIF
  ENDIF

  SET partner = get partner particle at position

  // left  side of partnership: <  operation
  // right side of partnership: >= operation
  // sorting key is the bin number
  IF compare * self.key < compare * partner.key
    WRITE self TO OUTPUT
  ELSE
    WRITE partner TO OUTPUT
  ENDIF

ENDFOR
```

Program 2: Pseudo code: Particle redistribution stage 1

```
Stage 2: Creating a bin-lookup table -> Binary search

COMPL.: O(b 2log(n)) with n=#particles and b=#bins
INPUT:  output particle buffer from (1) (Texture Buffer Object bound to TEXTURE2)
OUTPUT: bin lookup table buffer         (Texture Buffer Object unbound)

(2)  Binary search GPU
FOR each particle PARALLEL

  SET low   = 0
  SET high  = #particles

  SET key   = key we are searching for in the list // key it the bin number

  SET probe = the center of the particle list between low and high

  WHILE low <= high
    IF particle.key at probe > key
      SET high = probe  1
    ELSE
      IF particle.key at probe < key
        SET low = probe + 1
      ELSE
          // one item has been found, keep searching for first item
  SET high = probe -1;
        WRITE probe TO OUTPUT
      ENDIF
    ENDIF
    SET probe = the center of the particle list between low and high
  ENDWHILE

ENDFOR
```

Program 3: Pseudo code: Particle redistribution stage 2

```
Step 3: Particle redistribution -> Meyer et al. (2005)  [25]

COMPL.: 𝒪(n log2(27*a)) with n=#particles and a=avg #particles per bin
INPUT:  volume texture                  (3D Texture bound to TEXTURE0)
        bin lookup table buffer         (Texture Buffer Object bound to TEXTURE1)
        output particle buffer from (1) (Texture Buffer Object bound to TEXTURE2)
OUTPUT: INPUT==source?destination:source particle buffer  (Texture Buffer Object)

(3)  Meyer et al. (2005) GPU
FOR each particle PARALLEL
  SET self      = current particle coordinates
  SET binCoord  = current bin coordinates

  // 1) calculate repulsion velocity
  SET velocity = 0
  FOR each neighbour bin in z direction
    FOR each neighbour bin in y direction
      FOR each neighbour bin in x direction
        SET currentBin      = bin number of the currently observed bin
        SET firstInCurrent  = first particle in the current bin
        SET firstInNext     = first particle in the next bin

        FOR all particles in the current bin [firstInNext-firstInCurrent]
          SET neighbour = current neighbour particle to consider
          IF neighbour != self
            SET distance = distance between self and neighbour (i.e. Euclidean)
            IF distance <= predefined repulsion radius
              SET Ed = energy value between the self and the current neighbour
              SET velocity = velocity - change due to neighbour
            ENDIF
          ENDIF
        ENDFOR
      ENDFOR
    ENDFOR
  ENDFOR

  // 2) Update particle position in tangent plane
  SET gradient = gradient value obtained in the volume at self

  SET velocity = update based on velocity in the tangent plane // (Meyer, equation 8)
  IF velocity > predefined delta
    normalize velocity
  ENDIF

  SET self = self + velocity

  // 3) Reproject particles back to the surface
  SET gradient = gradient value obtained in the volume at self
  SET velocity = update based on reprojection //(Meyer equation 9)
  SET self = self + velocity

  // 4) Check if particle moved to another bin
  SET binCoordNew = bin coordinates of the new particle
  IF binCoordNew != binCoord
    SET self.bin = binCoordNew
  ENDIF

  WRITE self TO OUTPUT
ENDFOR
```

Program 4: Pseudo code: Particle redistribution stage 3

Appendix D: Derivation of curvature computation

This derivation, which was executed in Mathematica 5.2, is based on work by C. Sigg [31]
The transformation of the shape operator S to some orthogonal basis (u, v) of the tangent space,
is given by matrix A: $(u, v)^T.H.(u, v)$

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{12} & a_{22} \end{pmatrix};$$

The principal curvature magnitudes and directions are obtained through eigen-analysis of A.
The steps of the eigen-analysis are investigated. First the eigenvalues κ_1 and κ_2 are calculated.
$K = k/.\text{Solve}[\text{CharacteristicPolynomial}[A, k] == 0, k]$
$\text{k1} = K[[2]];$
$\text{k2} = K[[1]];$

$$\left\{\frac{1}{2}\left(a_{11} + a_{22} - \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right), \frac{1}{2}\left(a_{11} + a_{22} + \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right)\right\}$$

Subsequently, the eigenvectors p_1 and p_2 are calculated (in tangent space)
$\text{p1} = \text{NullSpace}[\text{k1} * \text{IdentityMatrix}[2] - A][[1]]$
$\text{p2} = \text{NullSpace}[\text{k2} * \text{IdentityMatrix}[2] - A][[1]]$

$$\left\{-\frac{-a_{11}+a_{22}-\sqrt{a_{11}^2+4a_{12}^2-2a_{11}a_{22}+a_{22}^2}}{2a_{12}}, 1\right\}, \left\{-\frac{-a_{11}+a_{22}+\sqrt{a_{11}^2+4a_{12}^2-2a_{11}a_{22}+a_{22}^2}}{2a_{12}}, 1\right\}$$

The tangent-space eigenvectors will be transformed back to object-space. Multiply with the orthonormal basis (u,v,w)
$\text{e1} = \{u, v, w\}.\{\text{p1}[[1]], \text{p1}[[2]], 0\};$
$\text{e2} = \{u, v, w\}.\{\text{p2}[[1]], \text{p2}[[2]], 0\};$
Simplify[e1]
Simplify[e2]

$$v + \frac{u\left(a_{11}-a_{22}+\sqrt{a_{11}^2+4a_{12}^2-2a_{11}a_{22}+a_{22}^2}\right)}{2a_{12}}; v - \frac{u\left(-a_{11}+a_{22}+\sqrt{a_{11}^2+4a_{12}^2-2a_{11}a_{22}+a_{22}^2}\right)}{2a_{12}}$$

For stability, division by zero is prevented. The eigenvectors are multiplied with a_{12}
$\text{e1} = \text{Simplify}\,[\text{e1} * a_{12}]$
$\text{e2} = \text{Simplify}\,[\text{e2} * a_{12}]$

$$\frac{1}{2}\left(ua_{11} + 2va_{12} + u\left(-a_{22} + \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right)\right)$$

$$\frac{1}{2}\left(ua_{11} + 2va_{12} - u\left(a_{22} + \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right)\right)$$

These are the eigenvectors in object space. We recognize eigenvalues $\kappa_1$and κ_2 in the eigenvectors, so we can simplify.
$\text{f1} = \{u, v, w\}.\{\text{k1} - a_{22}, a_{12}, 0\};$
$\text{f2} = \{u, v, w\}.\{\text{k2} - a_{22}, a_{12}, 0\};$
Equal[Simplify[e1], Simplify[f1]]&&Equal[Simplify[e2], Simplify[f2]]
True

We can write this in one formula:
$\text{g1} = (\text{k1} - a_{22}) * u + a_{12} * v;$
$\text{g2} = (\text{k2} - a_{22}) * u + a_{12} * v;$
Simplify[g1]
Simplify[g2]

$$\frac{1}{2}\left(ua_{11} + 2va_{12} + u\left(-a_{22} + \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right)\right)$$

$$\frac{1}{2}\left(ua_{11} + 2va_{12} - u\left(a_{22} + \sqrt{a_{11}^2 + 4a_{12}^2 - 2a_{11}a_{22} + a_{22}^2}\right)\right)$$

This equals the result presented by C. Sigg in his dissertation [31]. However, the (u,v) basis is swapped.

$\text{h1} = a_{12} * u + (\text{k1} - a_{11}) * v;$

$\text{h2} = a_{12} * u + (\text{k2} - a_{11}) * v;$

Program 5: Derivation of curvature computation