Painterly Stylization of Real-time Volume Rendering

by

Carlos Jorge da Cruz Ramalhão, BSc. Computer Science

Dissertation

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

Master of Science in Computer Science

University of Dublin, Trinity College

September 2010

Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Carlos Jorge da Cruz Ramalhão

September 13, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Carlos Jorge da Cruz Ramalhão

September 13, 2010

Acknowledgments

I would like to thank my dissertation supervisor, John Dingliana, for his availability and trust as well as all the assistance and guidance he provided. I would also like to thank my colleagues in the MSc. Interactive Entertainment Technologies for all their help and comradery, Sameer Baroova from PopCap Games, Andrew Corcoran and Niall Redmond from the Graphics Vision and Visualisation Group and all my dear friends for their patience and support.

CARLOS JORGE DA CRUZ RAMALHÃO

University of Dublin, Trinity College
September 2010

Painterly Stylization of Real-time Volume Rendering

Carlos Jorge da Cruz Ramalhão
University of Dublin, Trinity College, 2010

Supervisor: Dr. John Dingliana

Interest in the field of non-photorealistic rendering (NPR) has grown significantly within graphics research and development. NPR techniques are regarded as increasingly important tools to provide artists and designers with novel ways of achieving artistic expressiveness from visual information. Furthermore, NPR abstracts the detail of a given set of information such that resulting images are simplified into more comprehensible representations. For this reason, research in this field has also been driven by science domains such as medicine and physics, especially for the visualization of 3D volume information.

We shall concentrate on the research of painterly rendering techniques. That is, the representation of a scene in a way that it would mimic the visual appearance of a hand-made painting and the effects achieved from the materials used, such as oil or acrylic paints.

We propose a real-time interactive painterly rendering pipeline for the visualization of volume data on the GPU. A certain iso-surface, a region in the volume data of a certain iso-value, is retrieved through a raycasting algorithm, all the necessary information (*namely iso-surface intersections, gradients and flow*) is calculated on-the-fly and used to directly influence the final image. The result is a real-time rendering that takes inspiration from traditional paintings. The volume's colour, details and surface topology are conveyed using brush stroke colour, size, density and orientation. The brush properties and texture can be defined by an artist to achieve the desired result.

Contents

List of Figures

Chapter 1

Introduction

"Photorealism, like pornography, leaves nothing to the imagination."
- Cassidy Curtis [5].

Photorealism has always dominated research in real-time computer graphics. With increasingly less barriers between the current state-of-the-art and real-time photorealism though, a greater amount of research has been dedicated towards rendering simplified scenes where visual and creative expressiveness are vital. The field of non-photorealistic rendering (NPR) attempts to model representations that resemble artistic media, with or without the help of creative input by a user. NPR techniques are considered a key tool for the development of unique visual identities on real-time interactive applications. Some of the oldest and most common NPR techniques are recurringly implemented in the entertainment industry. Among these the most famous would include cel-shading (also known as cartoon shading), shape contours and hatching. Another important feature from NPR techniques is the ability to abstract renderings in such a way that unnecessary information is removed from the scene while relevant information is highlighted. This, of course, greatly impacts the understandability of an image by the end user. For such a reason, an area that has invested a good deal of research into NPR is that of volume rendering, especially for medical imagery.

Volume rendering refers to the visual interpretation of 3D discretely sampled fields of data. The representation of these values is relevant not only as a computer graphics technique but as a means of interpreting the sampled data in a way that would expose certain properties of the information, the later is often the primaty objective of volume rendering of scientific data. Use of volumetric data-sets is widespread among the fields of medicine and science, which led to a great amount of research in this area. Recently, it has also caught the interest of the electronic entertainment industry. New hardware allows representation of large quantities of volume data in real-time, thus relieving artists from previous constraints with art asset budgets, such as limited polygon counts.

Our proposal will concentrate on a particular set of NPR techniques called painterly rendering. Painterly rendering refers to stylization techniques that represent a scene in a way that emulates the visual appearance of a hand-made painting and the effects achieved from the materials used, such

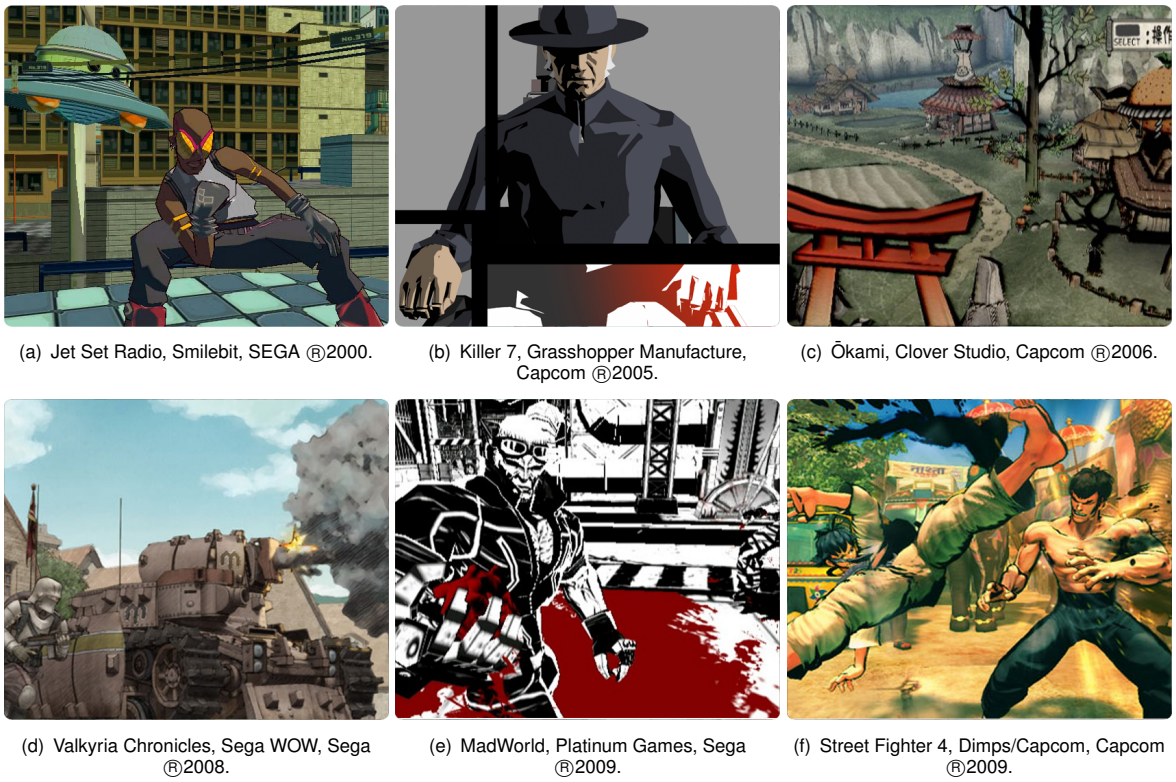
as oil or acrylic paints. When painting a scene, the artist relies on a refined range of techniques to achieve expressiveness and creativity, relying heavily on representation and abstraction to convey the essence of the scene. The artist can define forms, rhythm and energy through the variation of brush stroke texture, size and direction, in order to direct the viewer's eye. Larger brushes can be used to represent surfaces with little detail, uniform textures or to move the attention of the viewer away from objects of little interest, while thinner strokes can depict areas with more detail or highlight objects of greater significance. NPR (Non-photorealistic rendering) research has explored many techniques that achieve painterly rendering stylization by emulating the many effects of traditional painting through distributions of brush strokes of varying size, density and orientation. Significant amount of research pursued optimizations on arrangements and paths of said strokes. Some model-based techniques depend on the parametric surface to generate a series of particles that would be used to emulate a stroke. Other image-based techniques would use the gradient magnitudes and directions of an image to decide where to place strokes, where these should be coarser strokes or fine details and even extending them along the gradient normals. Most modern painterly rendering techniques operate under the assumption that artists initially use larger brushes for their rough sketch and progressively add in detail with increasingly smaller brushes in areas of higher magnitudes or of greater interest, where the artist uses finer strokes in order attract the viewer's eye to the finer detail.

Our proposal aims to combine state-of-the-art rendering techniques from both NPR and volume rendering into a solid real-time painterly stylization of volumetric data. We will thus discuss the implementation of a real-time painterly rendering pipeline specific to volumetric data-sets. The desired result should not only resemble a traditional hand-made painting but also express surface properties such as colour, detail and topology through visual cues like brush stroke colour, size, orientation and density.

1.1 Motivation

Non-photorealistic rendering (NPR) research attracts attention from many creative areas since it concentrates in providing great expressive capabilities to its users. In fields such as medicine and science, NPR has been highly sought for its capability to hide unnecessary information from the user and highlight relevant information instead, increasing comprehension of a given set of data. This is especially true when dealing with areas such as medical imagery.

Volume rendering is widespread among the fields of medicine and science, which led to a great amount of research in this area. Many researchers examine how to extract visual information from these 3D fields of data it order to correctly convey the relevant surface information like contours, shape, ridges, valleys and so on. These techniques can then be applied in order to enhance understanding of medical imagery resultant from non-invasive medical imaging techniques like Computed Tomography (CT) and Magnetic Resonance Imaging (MRI). Recently, it has caught the interest of the electronic entertainment industry as new hardware allows representation of large quantities of volume data in real-time. This technology relieves artists of common constraints with art asset budgets,



(a) Jet Set Radio, Smilebit, SEGA ®2000.

(b) Killer 7, Grasshopper Manufacture, Capcom ®2005.

(c) Ōkami, Clover Studio, Capcom ®2006.

(d) Valkyria Chronicles, Sega WOW, Sega ®2008.

(e) MadWorld, Platinum Games, Sega ®2009.

(f) Street Fighter 4, Dimps/Capcom, Capcom ®2009.

Figure 1.1: Non-photorealistic rendering techniques in Games.

such as limited polygon counts, providing incredible amounts of detail.

The relevance of non-photorealistic rendering in volume data is clear, but the additional investments by the entertainment industry in both fields further justify research in this area.

1.1.1 Non-Photorealistic Rendering in the Electronic Entertainment Industry

In the electronic entertainment industry, an increasing number of titles explore NPR techniques in search of a unique visual identity that would separate them from the competition. These techniques allow artists and designers to create novel interactive visual styles that often become a key characteristic of the products.

Titles such as *Jet Set Radio* (Smilebit, SEGA ®2000), *The Legend of Zelda: The Wind Waker* (Nintendo ®2002) or *No More Heroes* (Grasshopper Manufacture, Marvelous Entertainment, 2008) make heavy use of techniques such as cel-shading and contours and are renown for their unique visual styles. Titles like *Valkyria Chronicles* (SEGA WOW, SEGA ®2008) or *Ōkami* (Clover Studio, Capcom ®2006) for example, make use of watercolor and stroke-based sketching to achieve some amazing visual renderings. *Borderlands* (Gearbox Software, 2K Games ®2009) shifted their visual direction to include comic-book style illustration and contours, a shift that is generally regarded as

a good decision and that provided them with a unique visual identity that set them apart from the competition. Capcom's *Street Fighter 4* (Dimps/Capcom, Capcom ®2009) is highly regarded for its amazing visuals and includes some of the most extensive implementations of non-photorealistic techniques such as cel-shading, hatching, ink contours, watercolor and posterization.

As one can see, non-photorealistic rendering has become an increasingly important and popular tool with much to offer to industries such as electronic entertainment or animation. We find greatest motivation in our research from the rise of interest of such techniques and the impact it may have on the creative and innovative projects that make use of them.

1.2 Dissertation Layout

The contents of this dissertation have been structured as follows:

Chapter 2 (*Background and Related Work*) presents a literature review of the domains of interest for the work described in this dissertation. In it, you will find not only general introductory information on certain areas, but also reviews of state-of-the-art research of relevance to our work. Specifically, you will find information on both non-photorealistic and volume rendering domains, with a look at various different techniques and optimization algorithms, as well as relevant research in the areas.

Chapter 3 (*Design*) exposes the requirements and expectations of our work, the research proposal and objectives we set upon ourselves. This chapter also explains how the proposed system was designed and what influenced such decisions, both for the general pipeline and for each main component that makes up the final system.

Chapter 4 (*Implementation*) gives you an in-depth look at how the pipeline works and details the implementation of each component. In this section you can find a description of the pipeline, components and algorithms used, their implementation details and optimizations.

Chapter 5 (*Evaluation*) reviews and analyzes the implemented system. In this chapter we discuss how successful we actually were at reaching our goals and how well the results from our system met the requirements we had set. Requirements can be divided into visual output, visible properties and performance.

Chapter 6 (*Conclusions*) quickly reviews the proposal, challenges in its implementation and how successful we believe the system is at performing its tasks. In this chapter we not only review the proposed system but discuss future work to be done in this area that, although relevant, was not part of this dissertation because it was either out of the scope of our research or beyond our possibilities due to time constraints.

Chapter 2

Background and Related Work

In this chapter we will go into the background research from which this work was inspired and the theoretical background the reader should become acquainted with before we move onto the design and implementation of the proposed system.

2.1 Volume Rendering Techniques

The visual representation of 3D fields of data, usually referred to as volume rendering, has been an important research area over the past few decades. Although originally driven by scientific visualization such as fluid simulation and 3D medical imaging, volume rendering has also become an important field in other graphics disciplines such as real time graphics and their applications to computer and video games. Good examples of relevant work that targets applications in real-time graphics and entertainment are found in the work of Laine et al. [22] on efficient voxel representations (volumetric data) as an alternative to triangle-based geometry, or that of Harris [18] et al. on the simulation of cloud dynamics on graphics hardware.

Volume rendering covers a series of techniques used to interpret the contributions of each value, commonly referred to as a *voxel*, in a discrete 3D array of data, referred to as a *volume*, into images. A good example would be the interpretation of a medical scan of a human part and the respective 2D rendering of the volume in a way that would allow users to examine the properties of certain areas of the scan, such as bone or arteries.

Several techniques have been developed and optimized over the past few decades that enable us to visualize volumetric information, offline or in real-time, whether in increasingly photo-realistic or illustrative fashions.

Volume rendering techniques can be divided into two separate categories, direct and indirect volume rendering. In the following sections we will take a brief look at some of the more common approaches to volume rendering in real-time.

2.2 Indirect Volume Rendering (IVR)

Indirect Volume Rendering (IVR) techniques describe the representation of volumetric data by explicitly extracting geometric structures, often as polygonal representations, in order to render the desired information. Common indirect volume rendering techniques include the classic marching cubes algorithm proposed by Lorensen [26] et al. in which a polygonal mesh is constructed from an iso-surface (a surface described by values contained within a specified threshold) by examining a cube of values (eight neighboring voxels) and determining the correct polygonal representation for that configuration from a precalculated array of 256 possible polygon configurations. For further work involving surface extraction using indirect volume rendering, please consult the work of Levoy [25] et al.

2.3 Direct Volume Rendering (DVR)

Direct Volume Rendering (DVR) techniques display volumetric data by directly rendering the desired information without the need of an intermediate conversion into an auxiliary geometric structure. These techniques are able to handle large data-sets, complex information layering and dynamic or animated information with greater ease. Examples of popular direct volume techniques include shear-warp, 3D texture mapping, splatting and raycasting, all of which will be explained in more detail shortly.

2.3.1 Shear-Warp

One of the fastest CPU volume rendering algorithms available was first proposed by Lacroute [21] et al. in 1994. In this approach, volume information is split into slices and optimally traversed from slice to slice. The render is not performed directly into a final image but to an intermediate image we'll refer to as the base plane. The volume's slices are sheared and resampled so the viewing ray directions are perpendicular to the base plane. Then, the information is sampled onto the intermediate image which will need to be warped into the original projection. The ideal performances for the algorithm are possible due to optimizations, such as run-length encoding, that require non-uniform access to memory and thus are not feasible for implementation on the GPU.

2.3.2 3D Texture Mapping

One common approach to rendering 3D data arrays, popularized by the work of Cabral [3] et al. in 1994, bases itself on current hardware's capability to perform trilinear interpolations of 3D textures. In it, volume information is interpreted as a 3D texture where the parametric texture domain is sliced into multiple planes all of which are parallel to the current viewing plane, meaning they need to be updated whenever changes occur to the view. The respective index into the volume will be retrieved by the interpolated texture coordinate values of the slice. The value for that pixel can be retrieved through

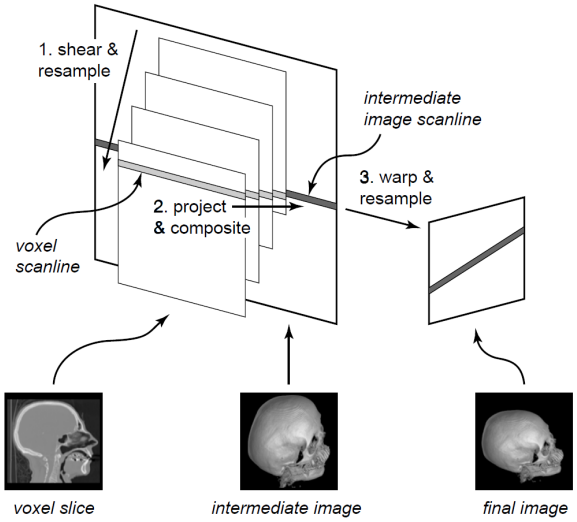


Figure 2.1: Shear-Warp algorithm, courtesy of Lacroute [21] et al.

hardware texture mapping, which allows slices to be placed anywhere in 3D space. The sampling rate for all viewing rays (*rays that originate from the viewer and intersect the viewport at a given point*) is identical for orthogonal projection but differ when using perspective projection, although this difference should only be noticeable with rather large fields of view. This provides a very fast hardware supported approach where quality and performance depend directly on the number of slices used to represent the volume, where too few slices result in greater sampling artifacts.

2.3.3 Splatting

Splatting, proposed by Westover[52] in 1990, is performed by projecting 3D reconstruction kernels onto the image plane. These kernels are integrated into 2D images called footprints. The final render may be composed using superposition of footprints weighed by the volumetric data values. This technique traverses the volume in object space and the results are projected onto image space. This traversal is quite flexible as long as the spacial order is maintained (*i.e. through spacial sorting*) to guarantee a correct final image composition. Recent techniques, referred to as image-aligned

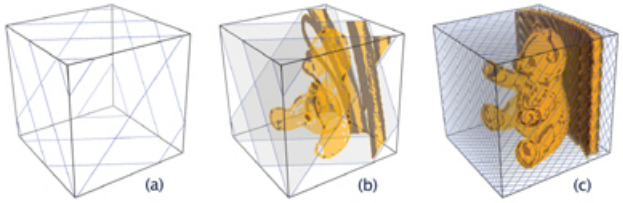


Figure 2.2: 3D Texture-based Volume Renderer, courtesy of Ikits [12] et al.

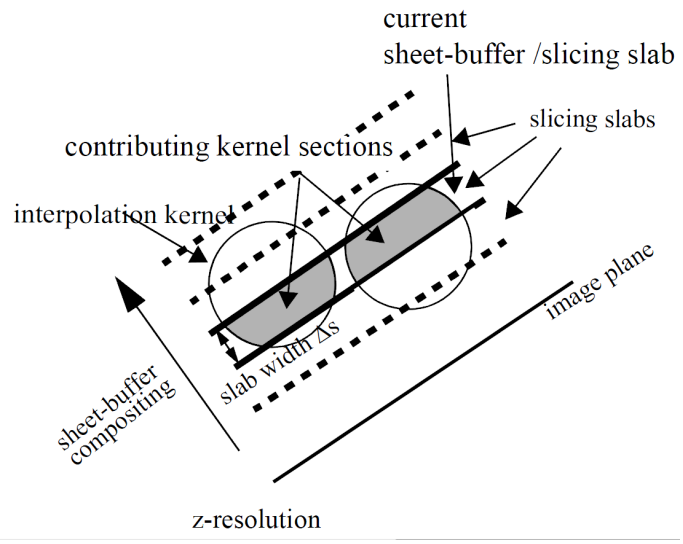


Figure 2.3: Image-Aligned Sheet-Based Splatting Algorithm, courtesy of Neophytou [35] et al. Δs refers to the sampling interval between image-aligned planes.

sheet-based splatting, traverse the volume information along sheets (*also called slabs, as referred by Mueller et al.* [33]) parallel to the image plane to eliminate drawbacks such as bleeding or brightness variations (popping artifacts).

2.3.4 GPU-Based Raycasting

Raycasting is a well known high-quality rendering technique (especially CPU rendering applications dating back to the 1980s). Raycasting refers to an image-order algorithm for computing the colour of a pixel from a 3D scene to a 2D screen. A ray of sight is computed from the eye of the observer and through the image plane, the 2D plane into which the users view of the 3D world will be projected

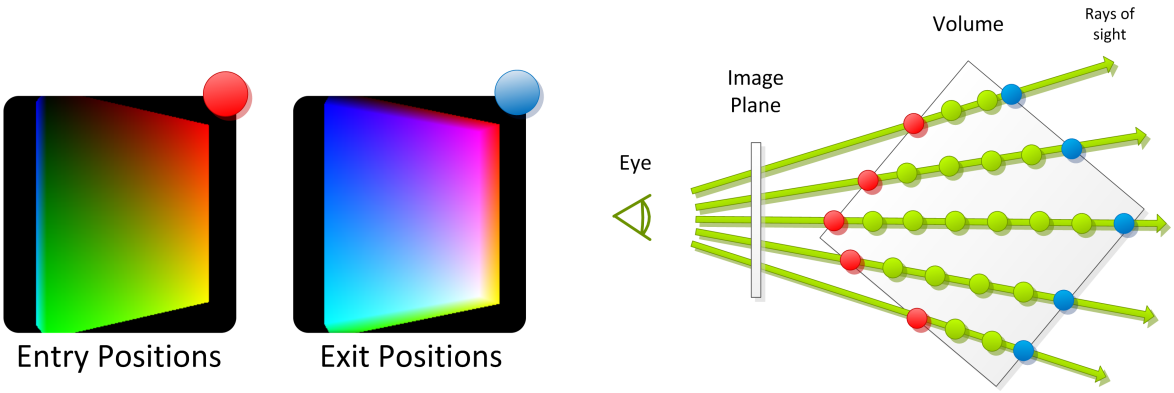


Figure 2.4: GPU-Based Raycasting Algorithm.

Listing 2.1: Fragment shader pseudocode for volume raycaster. Courtesy of Stegmaier [49] et al.

```
Compute volume entry position
Compute ray of sight direction
While in volume
    Lookup data value at ray position
    Accumulate colour and opacity
    Advance along ray
```

onto. Samples of contributions are taken along the defined ray until it intersects a surface, or in the case of volume raycasting, until it leaves the volume or already contains the final contribution for that ray. Unlike raytracing, this algorithm does not compute a new ray bouncing off of the intersection between the current ray and the surface. This algorithm, however, can be quite costly and has for a long time been considered not to be suitable for interactive rendering.

The graphics processing unit has grown into an incredibly powerful computational tool for parallelizable tasks such as volume rendering. Even more exciting is the recent evolution of these processing units to include more complex functionality such as branching and dynamic looping. These advances allow for a technique as simple and powerful as raycasting to be a viable option for rendering at interactive rates.

GPU raycasting provides a viable real-time alternative to slice-based methods such as 3D texture mapping. Despite their performance, these techniques suffer from several limitations that affect the quality and complexity of the final image. These implementations are inherently rasterization-limited and algorithmic optimizations are scarcely possible. They often produce artifacts due to integration step size variations caused by perspective projection. Complex effects (i.e. light distortion effects such as reflection and refraction) can be incompatible or require a great amount of effort to integrate with slice-based techniques. Finally, these approaches are not suitable for large data-sets since the number and position of the slices is dependant on the volume data-set.

In contrast, raycasting is a simple, flexible and robust technique that provides accurate, high-quality results. It is not constrained by the same issues as an inflexible approach such as slice-based rendering. The method is intuitive, fits neatly into current generation GPUs for parallel stream processing and will easily accommodate advanced rendering algorithms and effects. For these reasons, raycast is considered a future-proof alternative.

A GPU-based raycasting algorithm of interest was proposed by Stegmaier [49] et al. in 2005. A single-pass pixel shader is sufficient to implement the algorithm. For each pixel of the final image, each instance of the shader will cast a single ray from the camera, through the image plane and trace it along the volume. The volume information is sampled along the ray and the values of each sample are accumulated as contributions to the final chromaticity and opacity of the pixel in question. In order to generate the viewing rays and their limits for a certain pixel, one needs the entry and exit intersections of the bounding box that tightly encapsules the volume. For a pseudocode algorithm for this technique, please refer to listing 2.1.

Many optimization algorithms are available to improve the performance of raycasting. We'll quickly go through a few of these techniques. One of the easiest optimizations possible is the implementation of *early ray termination*, a very simple principle that takes advantage of the GPUs current branching capabilities in order to terminate calculations once all the visible contributions for a pixel have been calculated. For static geometry, it is possible to store information on the spacial properties of the volume and only evaluate rays intersecting areas that actually contain visible data with the support of spacial division structures, such as octrees. This technique is often referred to as *empty-space skipping*.

The most relevant technique for this work is called *Iso-Surface Extraction*. Iso-surface extraction involves extracting surface information instead of evaluating contributions along a ray. Each voxel within a volume is classified as belonging to a surface if its value is contained within a specified threshold. These surfaces are commonly designated as *iso-surfaces*, while the threshold is referred to as the *iso-surface value* or *iso-value*. Iso-surfaces can also be extracted using alternative algorithms such as the classic marching cubes algorithm. Although we extract surfaces directly onto the screen (or an equivalent off-screen buffer), these extracted surfaces can also be represented using intermediate data-sets such as polygonal surfaces. If you wish to know more about iso-surface extraction, please refer to the work of Lorensen [26] et al.

For more information on these and other techniques, please refer to the work of Krüger [20] et al. and Engel [11] et al. on volume rendering optimization techniques for the GPU.

2.4 Non-Photorealistic Rendering

In the following sections we will go into various research done in the field of NPR, especially when applied to volume information. Although the greatest focus is put on painterly rendering techniques, various different approaches are investigated as most of these techniques share source information, algorithms and objectives that are relevant to our research.

2.4.1 Cel-shading and Contours

Cel-shading, also known as *cartoon shading*, is a critical simplification technique for animation when scenes need to be (often manually) generated 24 frames per second of film. Because of this, artists where forced to budget their efforts and transmit the optimal amount of information to the user (emphasizing or omitting certain features). This algorithm is based on an animation technique where artists filled in existing line drawings in acetate cels with areas of solid colour.

This approach is now one of the most popular and widespread non-photorealistic techniques in the entertainment industry due to its simplicity and attractive visual results. Lake [23] et al. proposed a *hard-shading* (abrupt changes in shading) algorithm similar to a cel animator's procedure of painting an inked cel. Instead of calculating colour per vertex, Lake et al. generate a texture map of discretized colours (usually 2 or 3 different levels representing areas that are in shadow, illuminated

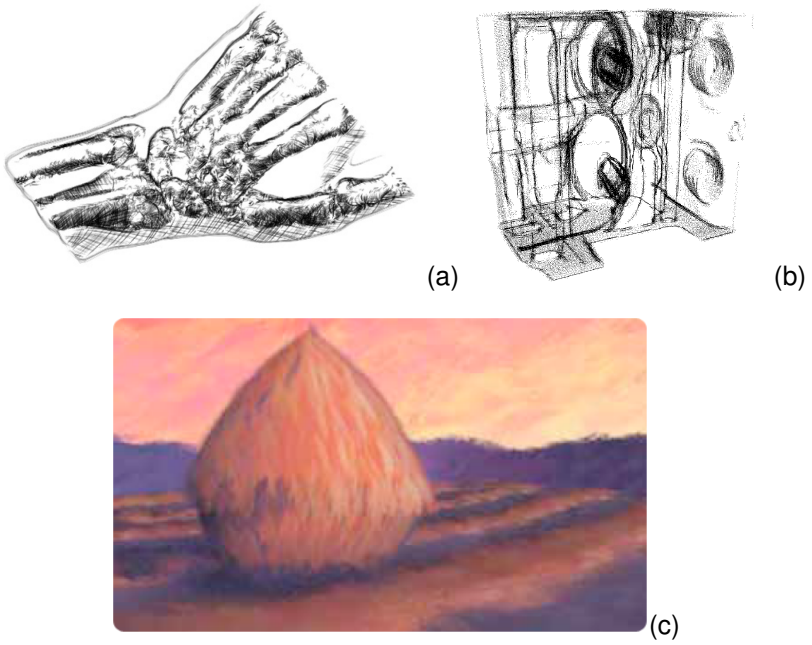


Figure 2.5: Non-photorealistic renderings. (a) Hatching representation of the volumetric dataset of a human hand, courtesy of Moritz Gerl [13]. (b) Engine block volume rendered as stipple drawing, courtesy of Lu [27] et al. (c) Painterly rendering of 3D geometry, courtesy of Meier [32] et al.

or highlighted). The illumination value is then used to sample the appropriate colour level from the texture map.

This technique is often accompanied by some sort of edge rendering technique. We consider edges to be either silhouettes, boundaries and creases. A surface point p is defined as a silhouette if $V.N = 0$ (V being the direction vector to the viewer and N the normal of the surface at point p, or alternatively a silhouette can also be defined as an edge that is shared by both a front and back facing polygon). Boundaries appear in non-closed models as edges that are not shared with any other polygon Creases are regions of a surface where the surface normals change abruptly.

Edge detection techniques can be divided into object-space and image-space. Saito and Takahashi [44] proposed an image-space edge detection algorithm where they use a depth render of the scene in relation to the viewpoint. They define profile edges as first-order derivatives of the depth values and internal edges as second-order differentials of the same values. For this they propose derivation through the use of a Sobel filter. Decaudin [6] et al. improved on this algorithm by taking normals into account, thus detecting creases as well as boundaries and silhouettes. Object-space algorithms include a brute force approach where all edges are iterated and tested if they are being shared by front and back facing polygons, which can become quite computationally expensive, specially for detailed animated models. Markosian [31] et al. proposed a probabilistic testing algorithm where they use rapid probabilistic identification of silhouettes, trading accuracy for speed.

2.4.2 Hatching

Hatching is a technique used in drawing wherein an artist expresses shapes using closely spaced parallel lines that follow the curvature of the forms they are used to describe. The quantity and thickness of the lines is used to represent lighting intensity and consequently describe the volume of the shapes. To express darker areas, cross-hatching is often used by drawing sets of parallel lines at different angles.

Hatching can convey the lighting, material properties and/or shape of a given scene. This technique can be used by itself, or with complementary techniques, not just to express shape and lighting but also to achieve interesting visual styles.

Winkenbach and Salesin [53] developed a pen & ink hatching algorithm from parametrized smooth surfaces using NURBS patches (Non-uniform rational B-splines, mathematical models commonly used to represent curves and surfaces). They used hatching density patterns to express complex texture and illumination effects. Elber [10] developed an hatching algorithm where lines where drawn based on the principal curvature of a surface. Using curvature allows us to capture important geometric features without depending on parametrization. This approach though still suffers from problems, especially in flat areas where curvature is not well defined, leading to noisy results. Saito and Takahashi [44] developed a 2D image processing algorithm that renders uniform density hatched images. They thin the number of lines drawn in areas where the gradient of the image is large and the contour lines density increases. In areas where contour lines density decreases, new contour lines are introduced in between existing lines.

The research of hatching on volume data-sets, sometimes referred to as volumetric hatching, aims to simulate traditional hatching techniques. One reason behind this research interest is the automatic generation of images similar to the ones used in scientific illustrations automatically onto relevant data, such as medical scans. A good example of this is the research done by Feng [8] et al. in the use of NPR in medical data.

2.4.3 Stippling

Stippling refers to the technique used to represent the volume of a shape, or other properties when used in scientific illustration, by varying the density of a pattern of small dots. Darker (or lower value) areas are represented by denser distributions of points while lighter areas will have lower density distributions. As a manual process, stippling can be very time consuming and has a very steep learning curve for beginners. It relies on even, but random, distributions of dots to express shape and texture of an object. A common approach to define dot density on the surface of the object is to directly relate it to the height of the surface from the current point of view, where height $h \in [0, 1]$. Pastor [38] et al. proposed a technique for real-time animated stippled renditions that produce view-dependant, frame-coherent animations based on 3D models. They use the concept of particle systems where each vertex is a particle that indicates the position of a potential stipple. Lu [27] et al. applied stippling render techniques by emphasizing features like silhouettes, surface

details and interior details though point distribution. For this they needed to extract from each voxel the following information: number of points, gradient, voxel data value, point size, list of points.

2.4.4 Painterly Rendering

Painterly rendering is the term coined for the representation of a scene in a way that would mimic the visual appearance of a hand-made painting and the effects achieved with the materials used (such as oil, acrylic paints and so on). We will go into detail on painterly rendering techniques since our work specializes on painterly stylization as a form of non-photorealistic rendering.

Meier [32] et al. implemented painterly rendering through a set of particles with a computed colour, orientation (dependant on the normal of the surface projected onto two dimensions along the view vector), and size (user-specified) where each particle is applied a certain brush texture to generate the final image. This approach was then implemented in real-time by Sperl [48] et al.

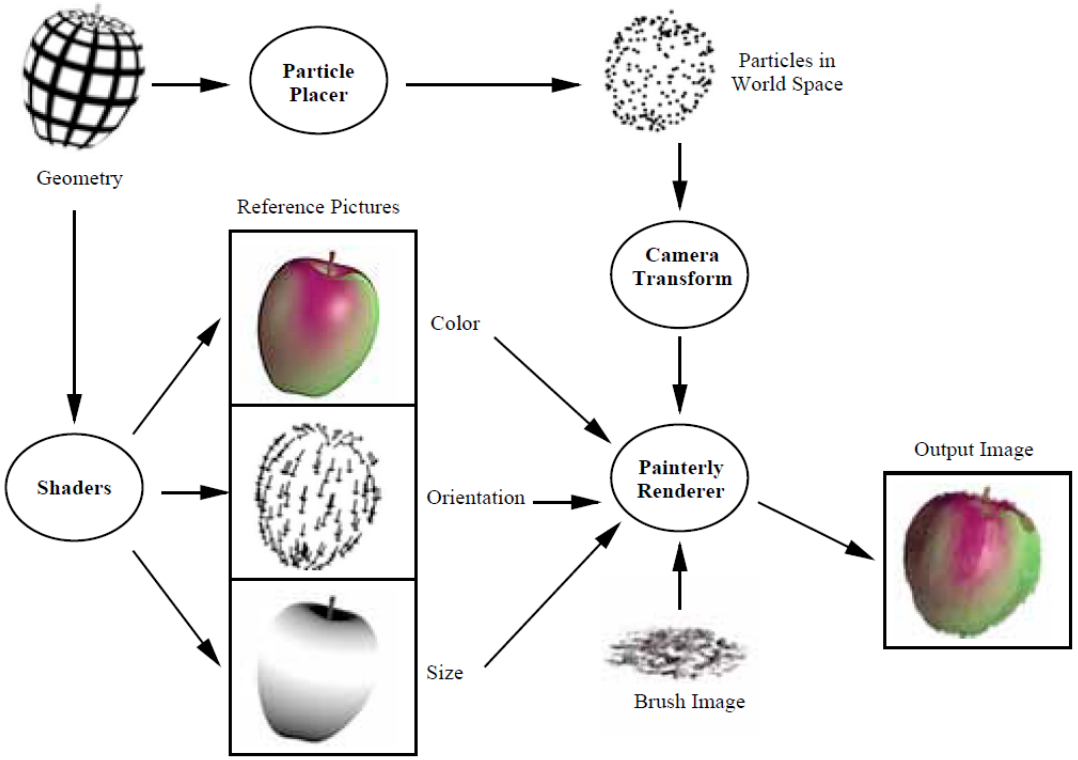


Figure 2.6: Painterly Rendering Pipeline proposed by Meier [32] et al.

Hertzmann [19] et al. built a painterly rendering model using curved brush strokes by assuming that the visual emphasis in the painting corresponded roughly to the spacial energy in the source image. Curved brush strokes are represented by cubic B-Splines, aligned to the normals of the image gradients. Their algorithm also allowed different size brush strokes since they took into account that

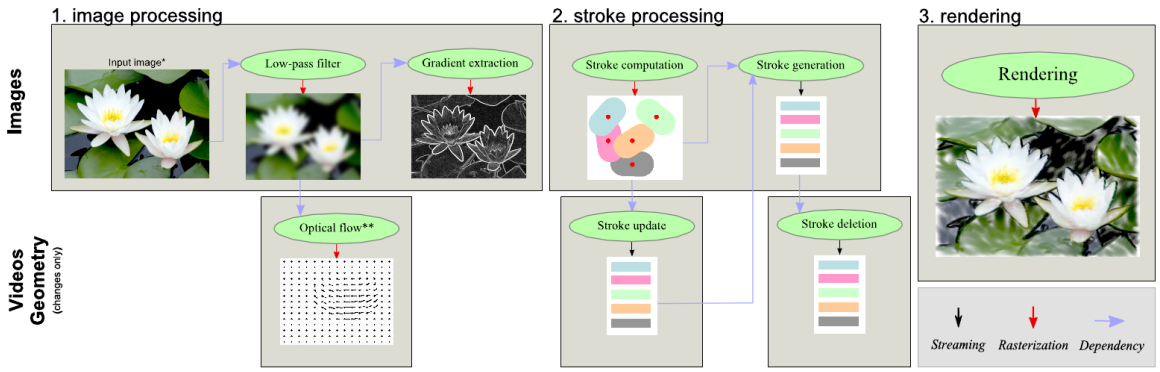


Figure 2.7: Painterly Rendering Pipeline proposed by Lu [28] et al.

artists usually start with rough sketches and add detail afterwards with increasingly smaller brushes, applying fine strokes to draw the attention of the viewer to fine detail.

Vanderhaeghe [51] et al. proposed a dynamic drawing algorithm for interactive painterly rendering based on the work of Meier [32] et al. that uses the surface's principal curvature to orient thick strokes in order to better represent surface shapes using painterly rendering.

The most important algorithm for the purpose of this paper was proposed by Lu [28] et al., who improved on Hertzmann's [19] work by implementing a parallel image-based painterly rendering algorithm applicable to hybrid scenes with image, video and 3D geometry. They divided an image into several layers with different levels of detail, in most examples three layers were sufficient. These were categorized as coarse, medium and fine detail layers. Strokes are stored in a 2D texture where each texel represents at most one stroke, with information about its orientation, center location and short and long radius. These strokes are then calculated based on a localized stochastic process where the probability of placing a stroke is defined by user and context, for example, probability of placement is inversely proportional to the number of strokes per texels. The coarseness of a stroke is determined by a set of probabilities for each layer, these are user-defined and style-specific but, since one would want edges (higher gradient magnitudes) to have a higher number of strokes in order to correctly delineate object boundaries, the following preset ranges usually achieve the best results:

The values p_c,p_m and p_f will represent the probabilities for placing strokes on the coarse, medium and fine layer respectively. It should be noted that it is not necessary to follow these guidelines for all rendering styles. The algorithm depends on two computer vision algorithms, one of which can be replaced by geometric information. Gradient information is extracted through a Sobel filter applied to a blurred image (3x3 box filter) in order to retrieve the gradient magnitude and direction for each pixel. This gradient information is necessary because artists commonly apply brush strokes following the boundaries between different shading levels, hence the natural orientation for a single stroke is perpendicular to the intensity gradient direction. The other algorithm is an optical flow algorithm they apply to video and image data. For geometry, they resolve optical flow through geometry reprojection,

that is, they track the position of a pixel p in the current and previous frame similar to the approach implemented by Nehab [34] et al. Stroke placement is composed of three steps:

1. Image processing: A low-pass filter is applied and the image gradient is extracted.
2. Stroke processing: The stochastic process generates new strokes and outputs them to stroke texture M. A vertex buffer of strokes is streamed to a geometry shader.
3. Rendering: The geometry shader reads the stroke information and generates point sprites for each. The scene is then rasterized.

On each update the stroke particles are moved according to the optical flow vectors. To reduce temporal artifacts, the following properties are restrained on update: fixed size and gradual update on orientation (1 degree per iteration, expect for fine details), colour is updated to location, opacity is changed so the particle fades out while new particles fade in to replace it. Particles overlapping in some areas are removed while new particles are generated in areas with insufficient stroke density. The authors also mention how failure in optical flow may generate artifacts, while gradient magnitudes for coarse-layers are not well defined resulting in poorly oriented strokes.

Chapter 3

Design

In this chapter, we expose our initial objectives and describe the design, inspirations and key features of the proposed work. Finally, we also take an in-depth look at the full rendering pipeline we propose in this chapter.

3.1 Goals

The aim of this work is to provide an implementation of a real-time painterly rendering pipeline specific to volumetric data-sets. The desired result should not only resemble a traditional hand-made painting but also express surface properties such as colour, detail and topology through visual cues like brush stroke colour, size, orientation and density.

3.2 Requirements

Because the pipeline in question is specialized towards volume graphics, all input information must be retrieved from the volumetric set itself in such a way that relevant intrinsic properties of volumes are explored and visible in the resulting images. Also, as the pipeline aims to apply painterly stylization to the input volume information, the result must not just reflect the volume's visual properties but also simulate the look of hand-made paintings.

3.3 Pipeline

In order to implement real time painterly rendering for volumes, the pipeline can be split into two main areas. The volume rendering component of the pipeline is responsible for sampling and calculating the necessary information from the volume in order to produce the final render. We use an implementation of real time raycasting on the GPU to retrieve the significant iso-surfaces from

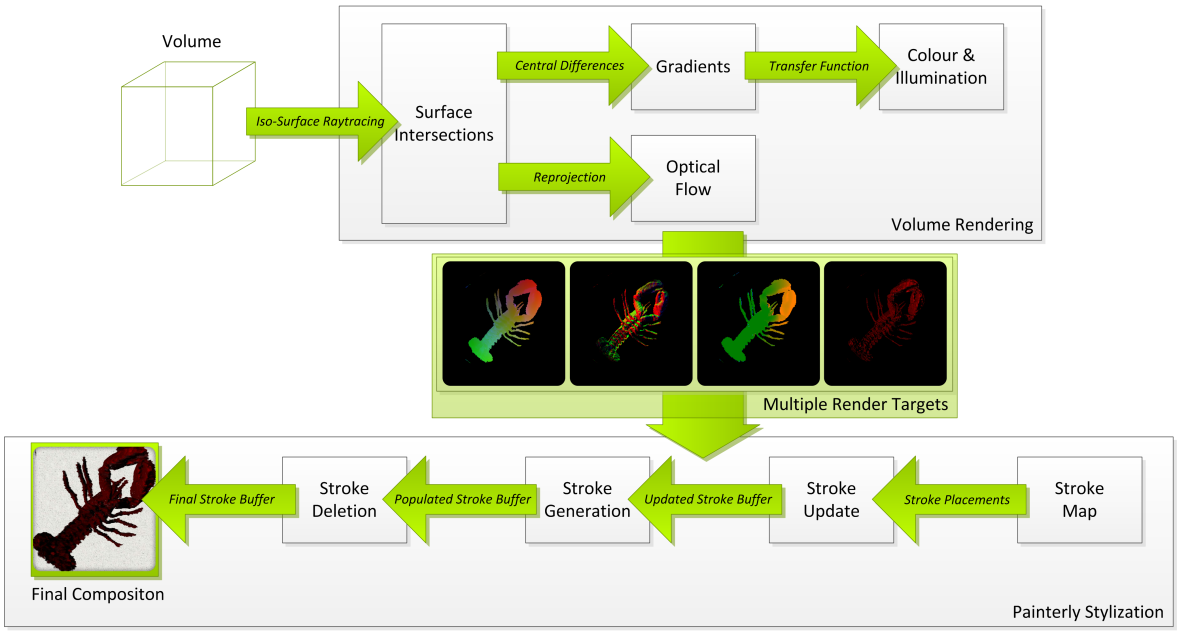


Figure 3.1: Illustration of Proposed Painterly Rendering Pipeline.

the data-set. We calculate the first order partial derivatives for the surface which we use to retrieve the gradient direction and magnitude. We then sample the colour values of the surface from a one-dimensional transfer function and apply a Phong illumination model (*Phong is an empirical model of local illumination that describes the interactions between light and a surface as a combination of ambient, diffuse and specular reflectivity components*). Finally, we calculate the optical flow of the data through reprojection and compare previous and current depth values to detect occlusion. This information is computed simultaneously and stored in multiple off-screen render targets.

The painterly stylization component of the pipeline is responsible for the generation of the final image given the data passed on by the volume rendering function. First, we threshold the magnitudes into separate levels, in this case three, that are used to indicate the areas of the screen that will be painted using coarse, medium and fine strokes. Using a GPU-based pseudo-random number generator we populate the screen using a stochastic placement function to achieve different stroke densities. We store all stroke information in stroke-map textures, where the texel coordinates specify the screen-space position of the brush stroke on screen and its channels store information on the orientation, size, aspect ratio and visibility. The size of each stroke will depend on which layer it belongs to while its orientation is directly dependant on the gradient directions of the surfaces. A buffer of strokes is generated from this information and updated according to the optical flow of the image, this may cause areas of the image to become either overly or insufficiently dense. To address this, we generate new strokes onto the stroke buffer using the previous stochastic placement function in areas where stroke density is lower than a specified threshold. Also, in areas with excessive congregation of strokes, strokes are removed from the buffer using a similar stochastic function as

the one used for placement. Finally, the stroke buffer is streamed through a geometry shader where each stroke is expanded into a screen-space quad and rendered with the appropriate stroke texture. A screen-space quad is a simple rectangular render primitive composed of two adjacent triangles to whom position coordinates do not refer to world space coordinates but directly to the position on the 2D screen surface.

We will go into more detail on each step of the pipeline in the following sections. For further information on the implementation of the pipeline please refer to chapter 4.

3.4 Volume Rendering: Realtime GPU-Based Raycasting

3.4.1 GPU Volume Raycaster

In order to retrieve all the necessary visual information off of the volume data-set we use a GPU raycasting algorithm based on the work of Stegmaier [49] et al. Stegmaier used a single-pass pixel shader to project rays in a volume and sample along the ray all the contributions for the final pixel. For more information on GPU volume raycasting algorithm consult chapter 2, section 2.3.4.

3.4.2 Iso-Surface Extraction

Our approach only extracts iso-surfaces directly from the volume and performs all further calculations directly on the resulting surfaces, as proposed by Krüger et al. [20] and implemented in the work of Hadwiger [16] on the advanced shading of discrete iso-surfaces. In iso-surface rendering, the only contribution to a given pixel is the first value contained within a certain threshold. The surface defined by the algorithm is called an iso-surface and the threshold applied to the sampled data is called an iso-value. For more information on iso-surfaces, please consult chapter 2, section 2.3.4.

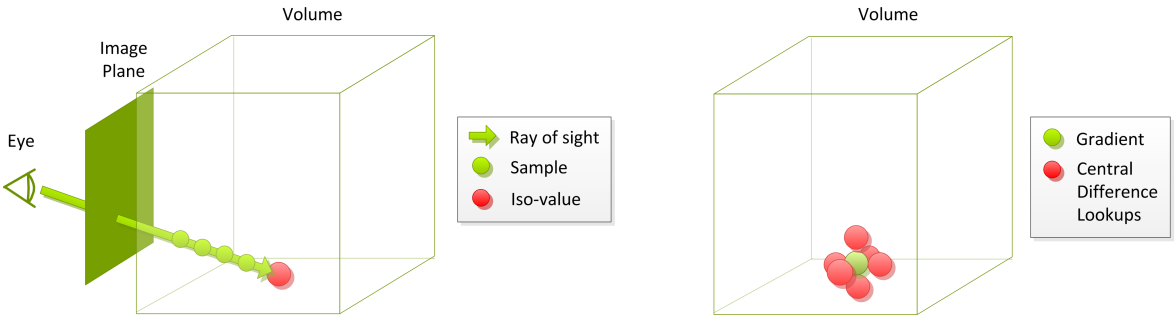


Figure 3.2: Illustration of Iso-Surface Raycaster and On-The-Fly Gradient Calculation.

3.4.3 On-The-Fly Gradients

As implemented by Hadwiger [16] et al., the gradient directions and magnitudes are extracted on-the-fly, that is, in real-time from the first-order partial derivatives of the values of the surface. These gradient calculations are limited to the area of the extracted iso-surface.

On-the-fly gradients incur an extra cost to the rendering pipeline, but the benefits outweigh the drawbacks since the extra-cost for on-the-fly gradient calculation is negligible on current hardware implementations and this real time approach supports the use of large, dynamic and/or animated data-sets as well as dynamic transfer functions.

3.4.4 Multiple Targets

The output of the volume rendering component of the pipeline includes surface intersection points, colour and illumination (using a Phong illumination model, refer to section 3.3 for further detail), gradient direction and magnitude, optical flow of the scene and occlusion detection. The actual rendering to screen is performed in a deferred step on the non-photorealistic rendering component of the pipeline. To optimize the pipeline interaction between components, we output all the necessary information into off-screen textures (render targets). All the output is calculated and returned in one single pass, simultaneously rendering all the results into multiple render targets to take advantage of current hardware support for parallel output.

3.5 Non-Photorealistic Rendering: Painterly Stylization

The second major component in the pipeline is the painterly stylization and render of volume data-sets. This component is responsible for taking the above-mentioned information and generating a set of on-screen brush strokes that will not only represent the volume but reflect its colour, details and surface topology correctly. For this, we were heavily inspired by the works of Meier [32] et al. and Sperl [48] in painterly rendering techniques, but the greatest inspiration came from the pipeline proposed and implemented by Lu [28] et al. on painterly stylization of images, videos and 3D animations. For more information on their work, please refer to chapter 2, section 2.4.4.

3.5.1 Deferred Rendering

Deferred rendering refers to the deferring of render steps such as shading or lighting further down the pipeline, where calculations are done at a per-pixel basis. With new hardware support for the parallel render of several targets, deferred rendering is now a very attractive option since scene complexity will not affect any calculations done in the deferred stages of the pipeline, where more computationally intensive techniques are often performed. In our pipeline, with the exception of the phong illumination model, which has negligible complexity, all complex calculations performed in order to generate, delete or update strokes, as well as their expansion to screen-space quads and

correct placement, are performed in one or more separate steps further down the pipeline. There is always an option to apply a custom illumination model in the deferred rendering stages.

3.5.2 3-tier Layered Brush Strokes

To simulate the process an artist would take in order to represent a certain image as a painting, the brush stroke creation was split into separate layers. One layer would represent the first pass of coarse strokes for the general shape. A set of medium sized strokes would specify further detail while a finer brush would finalize the image's highly detailed areas with denser, smaller strokes. These layers are retrieved from the surface's gradient magnitudes by thresholding three different, user-specified intervals.

3.5.3 Stochastic Placement

In order to place a stroke on a certain region of the screen, we use an independent stochastic function to decide whether or not a stroke should exist on a given pixel. By specifying different probabilities to the stochastic placement function on different layers, one can achieve different distributions of strokes. The stochastic function itself simply tests if the result of a random number generator is greater than the probability value for a certain distribution (please refer to appendix A for details on the implementation of the GPU-based pseudo-random number generator).

3.5.4 Stroke Persistency

One common problem with painterly rendering techniques is the inconsistent final result, caused by the random nature of stroke placement. If strokes are being randomly distributed per frame, this randomness introduces visual noise to the animation. If, on the other hand, the strokes are constant over the animation we introduce a problem commonly referred to as the *shower door* effect. The *shower door* effect is described by Meier [32] et al. as a result of brush-strokes having constant positions on screen instead of constant positions on the surface. The resulting effect is the distortion of the underlying shape with a certain static texture, *"as if it were being viewed through textured glass"*. To address this problem, existing strokes are preserved as much as possible by incrementally updating their properties, namely position, orientation and colour to reflect the properties of surface area they were originally assigned to. These strokes are also translated along the screen according to the optical flow of the image. When a given stroke disappears, is occluded or exchanges layers then that stroke is deleted instead of being updated.

3.5.5 Stroke Control and Repopulation

The attempt to keep existing strokes and update them along time causes stroke density to decrease in areas where strokes have originated and increase in areas where strokes may accumulate due to the optical flow of the image. In order to keep the general stroke density in the image uniform, a

density map is taken after the stroke update stage. If the density for a given area of the image is insufficient, then new strokes are inserted using the aforementioned stochastic placement function. On the other hand, if the density for a given area exceeds a set maximum value then strokes will be arbitrarily removed using a variation of the mentioned stochastic function.

3.5.6 Artistic Control

Because the aim of this work is not to automatically generate art, but to provide artists and designers with tools that will allow them to create meaningful visual expressions of the source data, several parameters are accessible in order to customize the result of the pipeline. The first and most important properties of interest to artists are the volume transfer function and stroke texture. The transfer function maps the volume data's values to visual output, namely colour, directly affecting the final colours of the image. The stroke texture is the visual representation of a single stroke to be used in the final image. Other values include layer thresholds based on gradient magnitudes, the size of the strokes per layer, the probabilities per layer used in the stochastic placement function to control distributions, the minimum and maximum stroke density and so on. Controlling these values can result in several drastically different results using the same source information and can be used to configure a certain painterly style, such as impressionism, pointillism and so on (for examples of different visual styles created with these properties, please refer to section 5.1.3).

Chapter 4

Implementation

In this chapter, we describe in detail how the rendering pipeline designed in chapter 3 was implemented and which decisions and optimizations were taken and why.

4.1 Iso-Surface GPU Raycaster

The iso-surface extraction algorithm implemented for the propose of this work bases itself on the GPU raycasting algorithm proposed by Stegmaier [49] et al. where rays of sight are computed from the entry and exit positions of the volume's bounding box and traversed within the volume using a lookup to a 3D texture on a single-pass pixel shader program. Our implementation, though, is only interested in the significant iso-surfaces, so instead of accumulating contributions we only return the intersection values from the view to the desired surface by testing a sampled, and classified, value in the volume against a set iso-value.

In order to retrieve the rays of sight for the volume area, we render the front and back faces of the bounding box surrounding the volume in local space $[0, 1]^3$. Then, retrieving a certain ray of sight is simply a matter of getting the ray position from the sampled front face value for a given pixel $Entry(x, y)$, calculating the ray direction as the sampled back face value for the same pixel as the ray starting position using $RoS(x, y) = normalize(Exit(x, y) - Entry(x, y))$. Both renders are stored in off-screen textures and calculated in separate render calls.

After retrieving a ray of sight we iterate through it with a given step value. The resulting coordinates (x, y, z) will be used to lookup the 3D texture containing the volume data in order to retrieve the data value for that position ($Value = Volume(x, y, z)$). This value is then classified, that is, it is used to look up a transfer function, in our case one-dimensional, and the classified value for that position is returned. In the current implementation, the transfer function expresses the final colour and opacity of each value in the volume and the opacity component of the classified value is tested against an iso-value ($Colour = Transfer(Value)$). To put it simply, only values with an opacity component superior or equal to the given iso-value are calculated and rendered ($Colour.a >= Isovalue$).

Listing 4.1: Fragment shader pseudocode for an iso-surface GPU raycaster.

```
For pixel coordinate x,y
Sample entry position from texture
Sample exit position from texture
Retrieve ray of sight start position
Compute ray of sight direction
While in volume
    Lookup data value at current ray position
    Classify value using 1D transfer function
    Test for intersection with iso-surface
    If intersection is detected
        Return intersection position
    Advance along ray
```

We'll now go into the calculated output from the extracted iso-surface information. This information is stored in a series of off-screen textures using multiple render-targets in order to efficiently render and store all the necessary information in a single pass.

4.1.1 Transfer Function

In order to map the values of a volumetric data-set to visual properties, we need some kind of lookup table to perform such conversions. That conversion table is what we refer to as the transfer function. A transfer function maps one or more input values into visual information such as colour. The values that serve as input to the transfer function need not be the original values of the data. One can use calculated variables as indices to the resulting value. For example, curvature-based transfer functions may use curvature magnitudes to index to a certain colour map that would visually represent ridges and valleys on the volume.

A transfer function may contain more than one dimension in order to achieve more accurate or complex data visualizations. For our purposes, a one dimensional transfer function will suffice. The transfer function will directly map sampled values from the 3D data field into colour values.

Our one dimensional transfer function is implemented on the GPU as a simple texture lookup to a user-defined texture, where each value is represented by the horizontal texture coordinate $u \in [0, 1]$.

4.1.2 Surface Intersections

Surface intersections are calculated directly from the iso-surface extraction algorithm mentioned above. This information is the starting point to all other calculations.

4.1.3 Gradient Directions & Magnitude

As mentioned in section 3.4.3, our implementation does not perform a pre-pass for calculating gradients offline. Instead, all gradients are calculated real-time on-the-fly, only for the areas of direct interest. This allows us to not only express large data-sets but also dynamic or animated volume data or transfer functions.

In order to calculate the gradient directions and magnitude, we need the surface intersection points and the 3D texture with the necessary volume information. Given that information, we lookup the surface intersection for each texel of the respective texture. We can then use the retrieved intersection points to index the volume data-set directly by sampling the respective 3D texture at the correct coordinates.

We decided to calculate all gradients using first order partial derivatives using a finite differences scheme. Central differences, described in equation 4.1, provide us with a fast and efficient way of estimating the gradients of a surface with only six texture lookups, two in every axis.

The gradient direction is taken from the normalized derivative calculated as described above, while the gradient magnitude is taken from the length of the unnormalized derivative. This information is packed into a 4-channel RGBA 32-bit texture where the colour channels store the gradient directions and the opacity channel stores the gradient magnitude.

$$\delta_h[f](x) = f(x + \frac{1}{2}h) - f(x - \frac{1}{2}h) \tag{4.1}$$

where:
$\delta_h[f](x)$ is the derivative of f at x.
h is the difference quotient (the distance from which to sample neighbor values).

4.1.4 Colour & Illumination

With the opacity used as an iso-value to determine visibility, iso-surface colours are retrieved directly from the colour components of the one dimensional transfer function.

For volume illumination, we opted for a gradient-based approach, using the extracted gradient directions discussed in section 4.1.3. To illuminate visible surfaces we apply a *Phong* illumination model (by Bui Tuong Phong [40], 1973) as demonstrated in equation (4.2).

$$I_p = k_a i_a + \sum (k_d(L.N)i_d + k_s(R.V)^\alpha i_s). \tag{4.2}$$

where:
I_p is the light contribution for a given surface point.
k_a, k_d and k_s are the ambient, diffuse and specular reflection constants, respectively.
i_a, i_d and i_s are the ambient, diffuse and specular light intensities (often described as colour values).
α is a shininess constant for this material.

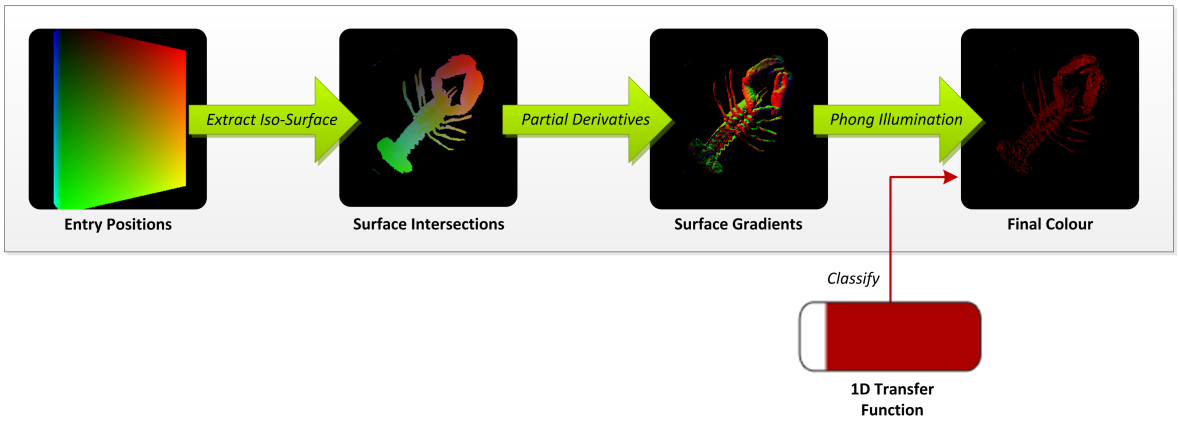


Figure 4.1: Illustration of Colour & Illumination Component Process

L is a direction vector from the surface point to the light source.
N is the normal vector at the surface point.
R is the direction of a perfectly reflected ray L off the surface point in question.
V is the direction from the surface point to the viewer.

The colour and illumination contributions are stored on an off-screen texture as a final combined colour value. This step is optional though, as illumination may be deferred to a different step further along the pipeline.

4.1.5 Optical Flow

Optical flow represents the patterns of motion of all the objects in the scene. With it, we are able to track a point on a surface across time. This enables us to tackle the randomness and *"shower door"* effects mentioned in section 3.5.4.

We take the optical flow of an image through a technique called reprojection. In reprojection we make use of the information available about the spacial transformations applied to objects at different times to calculate the motion of their respective pixels on screen. The position on screen of a point belonging to an object can be calculated by simply multiplying it by its respective world, view and projection transformations, in order to transform the position's coordinates from local-space to screen-space. Knowing the transformations applied to an object at time t and the same calculations applied at time $t - h$ one can easily know the motion vector of point p for the period of time h using $g(p) = f(p, t) - f(p, t - h)$.

For further information, we refer to Nehab [34] et al. and their work in reverse reprojection for real-time caching.

In order to store information on the scene's optical flow, we used a 4-channel 32-bit RGBA off-screen texture where the values of the pixel motions along the screen's x and y axis. These values must be packed into two 8-bit channels with range $[0, 1]$. Remaining channels are used for occlusion

detection as described in section 4.1.6. For further information on how optical flow is used in the pipeline please refer to sections 3.5.4 and 4.2.3.

4.1.6 Occlusion

In order to detect if a certain existing stroke represents an area now occluded and thus should be deleted, we require the detection of not just optical flow but the occlusion of pixels between frames. To do this we take the previous pixel's post-projection z coordinate $z(t-h)$, compare them to the current post-projection z coordinate $z(t)$. The post-projection z coordinate should match regardless of surface movement. If values do not match it means the surface point under evaluation is no longer visible, so we consider there to be an occlusion.

Detected occlusions are flagged into an off-screen texture, in our particular case using the leftover channels from the optical flow output.

For more details on how occlusion information affects the rendering pipeline, please refer to section 4.2.3.

4.1.7 Downscaling

Because only a small set of pixels are actually represented by strokes, it is possible to use low resolution renders for the most computationally expensive stages of the pipeline, such as raycasting and gradient estimation. This dramatically increases the system performance without any noticeable impact to the final result. To implement this optimization, all texture renders are performed on low resolution targets (i.e. 480x300 render targets) while the final render of the screen-space quad stroke representations may be done at full resolution (i.e. 1280x800) with little or no impact on the performance of the system.

4.2 Painterly Rendering

4.2.1 Brush Stroke Maps

The next step in the rendering pipeline is to separate visual information into layers. Lower layers would be filled with coarser strokes to describe uniform areas of the image. The higher the layer, the more detailed the strokes and higher stroke density, in order to be able to represent greater detail. In our particular case, we limit our use to three different layers: coarse, medium and fine. This configuration of layers seems ideal in both quality and performance.

To achieve this effect, we segment the image into three levels of gradient magnitudes. Higher gradient magnitudes represent areas with sharper difference between values and thus greater need to represent detail, while areas of lower magnitude have less need to represent details and thus can be represented with fewer, larger strokes. Segmentation is performed by simply applying the stochastic

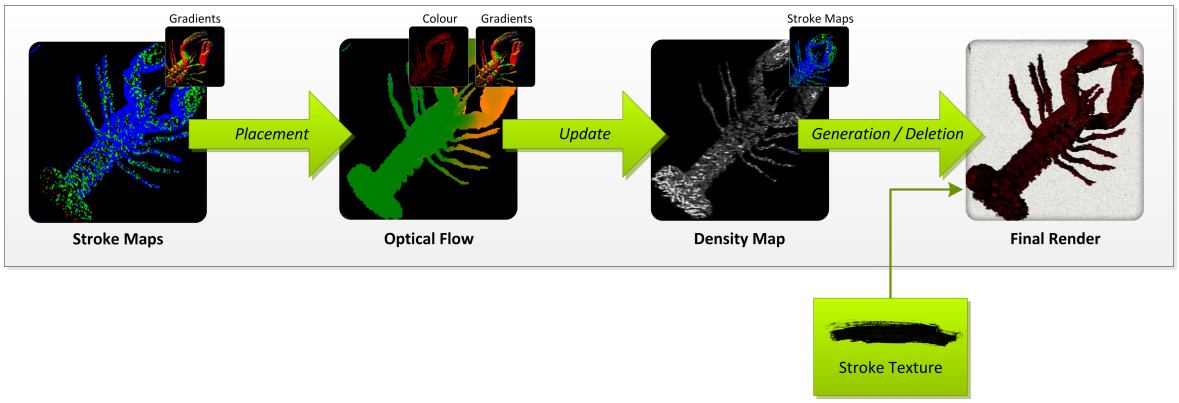


Figure 4.2: Illustration of Painterly Rendering Stages.

placement function, which we will discuss below, only when gradients are within the defined threshold for the level in question.

The stochastic function used to independently perform the placement of strokes in screen-space is based on a GPU pseudo-random number generator (for a detailed explanation of how the random number generator works, please refer to appendix A). This random number generator is used to return random values that will be tested against a certain user-defined probability. So, for a distribution rate of 10% (one stroke per every ten pixels) one would define the probability for that area to be $p = 0.1$ and by only placing strokes when $random \leq p$ we achieve the desired average distribution. This approach is advantageous because it runs independently per pixel, making it a perfect candidate for a GPU-based single-pass pixel shader program.

Once the placement of a certain stroke has been decided for a given pixel on screen, we need to specify the remaining stroke properties, namely scale and orientation. The size of strokes will be defined directly by the layer of strokes where it belongs to. So all coarse layer strokes will have the same size, which should be applied to all strokes in that level.

The orientation is calculated by using the previously retrieved gradient directions. Gradient directions are projected into screen-space and then normalized. The angle between the origin rotation (horizontal brush stroke) and a direction perpendicular to the screen-space gradient direction is then calculated and applied as the orientation of the brush. An approach similar to the one proposed by Meier [32] et al. for brush stroke orientation.

With all properties of the brush strokes already calculated, they are finally stored in an off-screen 32-bit RGBA texture called a '*Stroke Map*' (please refer to Lu [28] et al. for details on the original definition of stroke map). The position in the texture defines the position on screen and the channels contain the size, orientation, aspect ratio and a flag to indicate whether or not the stroke exists at that pixel. Different stroke layers are rendered to different textures to avoid the use of conditional instructions when accessing the stroke information. Also, current hardware allows us to process and output all layers in parallel and in a single pass, using multiple render targets.

One obvious limitation with this approach is that you can never have a higher number of strokes

per image then the map pixel resolutions. This, though, does not present a problem as it is rarely the case that the number of strokes will surpass or equal the number of pixels available on the stroke map textures.

4.2.2 Stroke Buffer

A stroke buffer is a linear structure that contains the description of all existing strokes in the scene. Each component of the buffer contains the position, texture coordinate, colour, size, orientation and aspect ratio of the stroke. The position specifies the screen-position of a stroke. The associated texture coordinate indicates where in the several texture maps that hold relevant scene information the stroke is located. The colour specifies the tint with which the stroke texture will be modulated. The size, orientation and aspect ratio define how to convert a stroke from a point to an oriented screen-space quad.

The stroke buffer is generated by iterating through the generated stroke maps (for further detail on stroke maps, see section 4.2.1) and generating a new entry for each stroke placement found. The stroke buffer is also passed through the stroke update, generation and deletion step in order to add, remove or modify stroke entries as needed.

In order to take advantage of hardware support, we use point-list vertex buffers to hold stroke information, and pass it through a series of geometry shaders for data manipulation. Geometry shaders were introduced into the programmable graphics pipeline with *Shader Model 4.0*. These shaders can receive output from vertex shaders and emit zero or more primitives, passing them along towards the pixel shader for rendering and/or streaming them back into memory. These properties make geometry shaders the ideal tool for stroke manipulation, generation and deletion, as well as for the generation in hardware of a final screen-space quad system from a list of points. To add new strokes to a buffer, one simply has to append a given stroke input to the output stream. For the deletion of strokes, the data is ignored and no stroke information is output for that point.

4.2.3 Stroke Update

The stroke update step aims to reduce temporal incoherency from the pipeline through the use of persistent strokes. Common problems with the lack of temporal coherency in painterly rendering include noisy randomness or the *shower door* effect. Please refer to section 3.5.4 for more information on the objectives of this step.

In order to update the strokes correctly in screen-space we need to track the motion of several pixels across the screen. For that we need the optical flow of the scene. For details on how optical flow works and/or how it is calculated please refer to section 4.1.5.

Given the optical flow of the scene we know exactly where each pixel in the screen moved to from the last frame. Now, updating strokes becomes simply the process of looking up what was the offset of the pixel the stroke is associated with and update its position and texture coordinates.

With a new position and texture coordinates, we'll need to reevaluate the stroke's colour, opacity and orientation based on the information from the colour and gradient direction of the newly associated texture coordinates. For details on how colour, opacity and orientation are calculated from the respective texture inputs, please refer to section 4.2.1. Also, to reinforce temporal coherency, colour, opacity and orientation may only change a limited amount per frame. In the current implementation, strokes only take 5% of the new orientation, one third of the new colour and 90% of the new opacity.

To smoothen transitions and alterations on the image, any new stroke is created with nearly zero opacity and any destroyed stroke is set to target transparency zero. This provides a seamless fade in/out system for all strokes on screen.

Besides the optical flow information available to us, we also have information on pixels occluded between frames. This allows us to identify which existing strokes have been occluded by new overlapping strokes and delete them appropriately.

When a given stroke moves out of its respective layer, the stroke is faded out and destroyed. New strokes should cover the empty areas during the stroke generation step, refer to section 4.2.4.

This step is implemented in hardware as a geometry shader that receives a stroke buffer (for more information on stroke buffers, see section 4.2.2) as input, where each point in the buffer represents a stroke. Each stroke is processed, updated and copied over to an output stroke buffer. If a stroke has been marked for deletion during the update process, the program will simply disregard the stroke and output zero primitives for the stroke in question.

4.2.4 Stroke Generation

Strokes in this pipeline are animated in such a way that they follow the location on the surface they represent (for more information on how strokes are updated see 4.2.3). This may cause the distribution of strokes to shift with time, leaving many areas with insufficient strokes to represent the necessary surface information. To counter this problem we analyze the density of strokes across the surface of the visual data and repopulate areas that do not contain sufficient stroke density to correctly represent the surface.

The current stroke buffer is rendered to screen as a quad system (as described by section 4.2.6). The opacity of each final textured stroke is accumulated in an off-screen texture by additively rendering the resulting quads onto a render-target. The contribution of each stroke's opacity value is reduced so the resulting values $[0, 1]$ can represent areas with several overlapping strokes.

The stroke generation step will then iterate through the whole density map. For areas of the surface where density values are below a specified minimum, this stage will generate and place new strokes to fill in the gaps. The minimum density is user-specified and designates how many overlapping strokes any surface area should have as a bare minimum. The generation of new strokes is done through the same stochastic placement process described in section 4.2.1 by looking up the respective stroke map for each pixel. Because the stroke map for that frame already has a set of possible stroke placements with a given distribution, this step will ensure the correct density of new strokes for areas with a low population of strokes.

4.2.5 Stroke Deletion

As strokes follow the optical flow of the areas on the surface to which they belong, they may drift with time and cause a conglomeration of strokes in certain areas, such as edges. To address the excessive density of brush-strokes in certain areas, we perform another pass where we evaluate the density of strokes across the surface and stochastically remove strokes from overpopulated areas.

The current stroke buffer is rendered additively to a texture in order to represent the density of strokes across the surface. For more details on the density map and how it works, please refer to section 4.2.4.

This stage will then iterate through all strokes it receives as input and delete strokes in areas where the density of strokes exceeds a specified maximum threshold. On such areas, strokes will be deleted using a stochastic process similar to the ones described in sections 4.2.1 and 4.2.4. This way, just enough strokes are deleted to keep the desired distribution. Deletion is performed by simply omitting the stroke in its output stream.

4.2.6 Expansion into Screen-space Quads

The final step of the painterly rendering pipeline is responsible for taking a buffer of strokes in the form of a list of points and generating a system of screen-space quads to be rendered to the screen with the correct position, colour, size, texture and so on. A screen-space quad is a simple rectangular render primitive composed of two adjacent triangles to whom position coordinates do not refer to world space coordinates but directly to the position on the 2D screen surface.

For each stroke received by this stage, we calculate a quad centered around the stroke position, with the respective size and aspect ratio and we apply a rotation to the quad's vertices in order to achieve the final orientation. To take advantage of hardware support for these operations, these steps are implemented as a geometry shader that receives a stroke buffer (for more information on geometry shaders and the stroke buffer, see section 4.2.2) and outputs a list of triangle strip primitives, the screen-space quads that will represent the final strokes.

The geometry output by the geometry shader is passed to a pixel shader program that finally applies the correct texture and colours it appropriately.

Chapter 5

Evaluation

In this chapter, we will evaluate how successful we were in our implementation and examine how well the outcome matches the expected results. The objective of the proposed pipeline is to establish an implementation of a real-time painterly rendering pipeline specific to volumetric data-sets. The expected result should not only resemble a traditional hand-made painting but also express surface properties such as colour, detail and topology through visual cues like brush stroke colour, size, orientation and density.

5.1 Results

To evaluate the implemented system, we will examine the final visual outcome and how well it meets all the objectives. We will also evaluate the performance of the implemented pipeline, and whether or not it is capable of real-time visualization.

5.1.1 Visual Results

We can see from the outcome of the discussed pipeline a system of brush strokes that not only consistently covers the visible surface with strokes that reflect its visual properties, but also reduce temporal incoherency by moving the strokes along with their corresponding original surface area.

The result is a smooth visually attractive, painterly rendering of visual volumetric information. We can see little sudden change and random noise in the outcome. Areas of higher detail are represented by a large quantity of small brush strokes, and thus given greater emphasis, while low detail areas are covered by larger, simpler strokes. The general shape of the volume is defined with large, coarse strokes in the background, while the detail is filled in using finer strokes, mimicking a common practice in artistic rendering. Also, one can see the stroke colour correctly conveys the colour of the original rendering and, more importantly, the strokes follow the general shape of the volume, thus expressing natural cues of the topology of the volume's surface.

Figure 5.1: Temporal Coherency through Optical Flow based Stroke Update. Strokes are following the objects surface as it rotates to the right.

Temporal Coherency

From an animation point of view, we set off with the objective of reducing temporal incoherency in the proposed pipeline by updating the strokes along the surface area they were created to represent. We achieved this by calculating the optical flow of the final animation through a technique called reprojection (please refer to section 4.1.5 for more detail). In figure 5.1, we see a sequence of images representing the evolution of strokes with time. In this scene, generation or deletion of strokes based on density has been switched off and we can easily see how the strokes are being updated as the object in question rotates to the right. As we can see, the strokes correctly stick to the areas they were assigned. One issue visible from this progression, though, is the deletion of strokes due to overly sensitive occlusion detection or gradient noise causing strokes to switch between layers.

Surface Topology through Strokes

As for the objectives we set for any static image (whether or not it is part of an animation), two are worth further discussion. The first is how well an image expresses surface information such as colour and surface topology.

For better visual results and to provide visual cues that can express the shape of the object, we oriented the strokes in our pipeline according to the gradient directions extracted from the volume's surfaces. In figure 5.2, we can see the gradient directions overlaid onto the final image. As we can, see, especially from the back of the shape in question, the gradients correctly follow the sides of the lobster, and down along the shape of the legs. We can also see that in areas with low gradient magnitudes, such as the tail or claws, the gradient direction is not well defined and may introduce noise into the animation.

Figure 5.2: Brush Stroke Colour and Orientation. The orientation vector field was overlaid on the final image in order to examine if stroke directions correctly follow surface gradients.

In order to truly evaluate the impact correct stroke orientation has in the final image, we compare two identical renderings in figure 5.4. The image on the left has been rendered with strokes oriented along the gradient directions of the surface. The image on the right had been rendered with static stroke orientation. As we can see, detail on the gradient oriented strokes render is much greater and obvious than on the statically oriented render, resulting in an image that is overall visually pleasing and easier to understand.

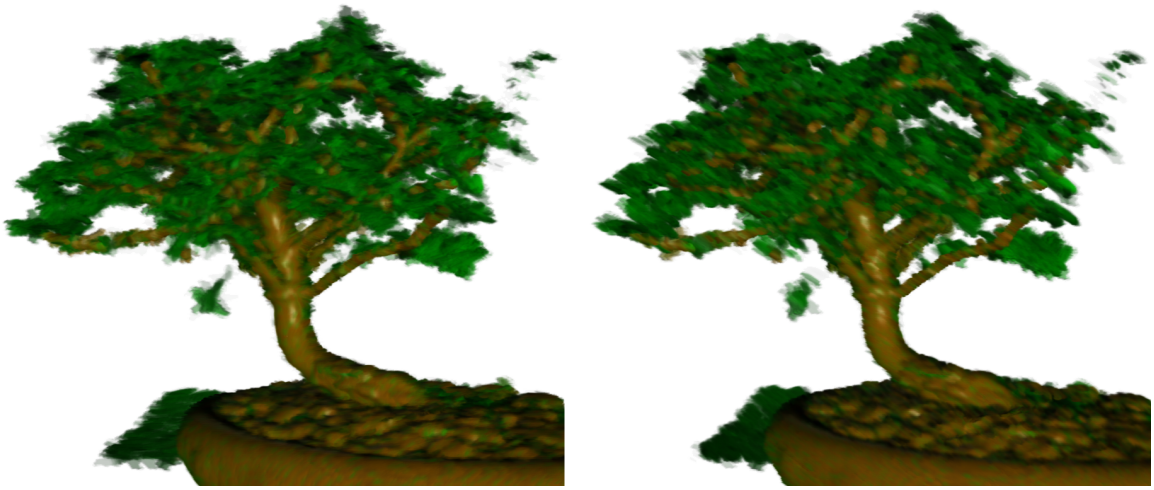


Figure 5.3: Comparison between gradient oriented strokes and statically oriented strokes.

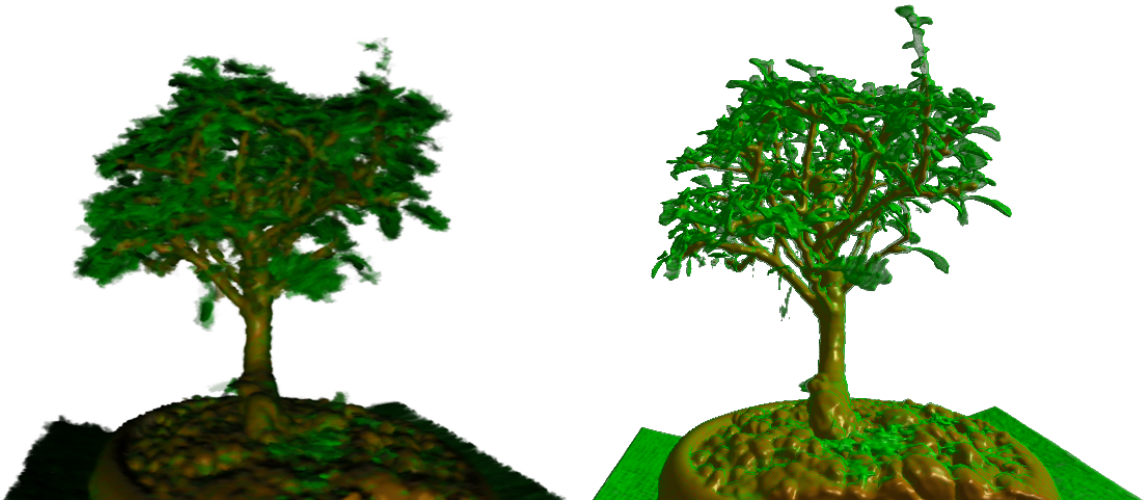


Figure 5.4: Comparison between the same volume rendered by our pipeline (left) and the Voreen Engine (right).

Comparison of Painterly Rendering with Standard Volume Raycasting

In figure 5.4, we can see the result of our painterly rendering pipeline (left) side by side with a the volume raycasting rendering from one of the main volume rendering applications available, the *Voreen* Volume Rendering Engine (right). Both renderings share the exact same volume data-set and transfer function. As we can see, our pipeline not only correctly displays all essential information from the original render, it also abstracts any unnecessary information by representing lower magnitude gradients in less detail. The result is a visually attractive painterly rendition of the original volume data-set, even when relying on a fairly basic set of information and simple 1D transfer function.

5.1.2 Performance

The work discussed and implemented here is suitable for real-time interactive applications that desire painterly stylization of volume information. The system ran at interactive, real-time frame rates on current generation hardware. Namely, the software rendered a 256x256x256 8-bit volume data-set at a final resolution of 1280x800 on an Intel Core 2 Duo E6300 CPU at 3.15GHz with 4GB RAM and an NVIDIA Geforce GTX470 using Microsoft DirectX 10 rendering an average of 30.000̃50.000 strokes on screen at an average frame rate greater than 60 frames per second.

5.1.3 User-defined Values

Because the system we have discussed is meant as a tool with which artists and designers may express themselves, several parameters were made accessible that let the users customize the result of the pipeline as they wish. The developed pipeline allows the user to tweak the values and sources as they see fit in order to generate truly creative and impressive visual styles.

Three styles were developed to showcase the power and flexibility of the developed pipeline. Each styles achieves a separate and unique result that exemplifies the creative possibilities of the proposed work. For the detailed parameters of each style please refer to table 5.1. Each style is demonstrated with the render (and close-up) of two different volumetric data-sets, an orange and a bonsai tree (please refer to figure 5.5).

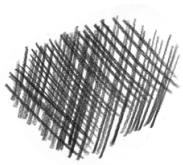
	Style (a)	Style (b)	Style (c)
Number of Layers:	3	3	3
Threshold (Layer 1):	0.1	0.1	0.1
Threshold (Layer 2):	0.3	0.3	0.3
Threshold (Layer 3):	1.0	1.0	1.0
Probability (Layer 1):	0.01	0.01	0.01
Probability (Layer 2):	0.05	0.02	0.02
Probability (Layer 3):	0.05	0.05	0.05
Size (Layer 1):	100	150	200
Size (Layer 2):	100	100	80
Size (Layer 3):	40	50	30
Maximum Density (Layer 1):	0.3	0.8	0.8
Maximum Density (Layer 2):	0.1	0.03	0.025
Maximum Density (Layer 3):	0.1	0.03	0.02
Minimum Density:	0.1	0.01	0.01
			

Table 5.1: Table of user-defined values for the three different showcased styles.

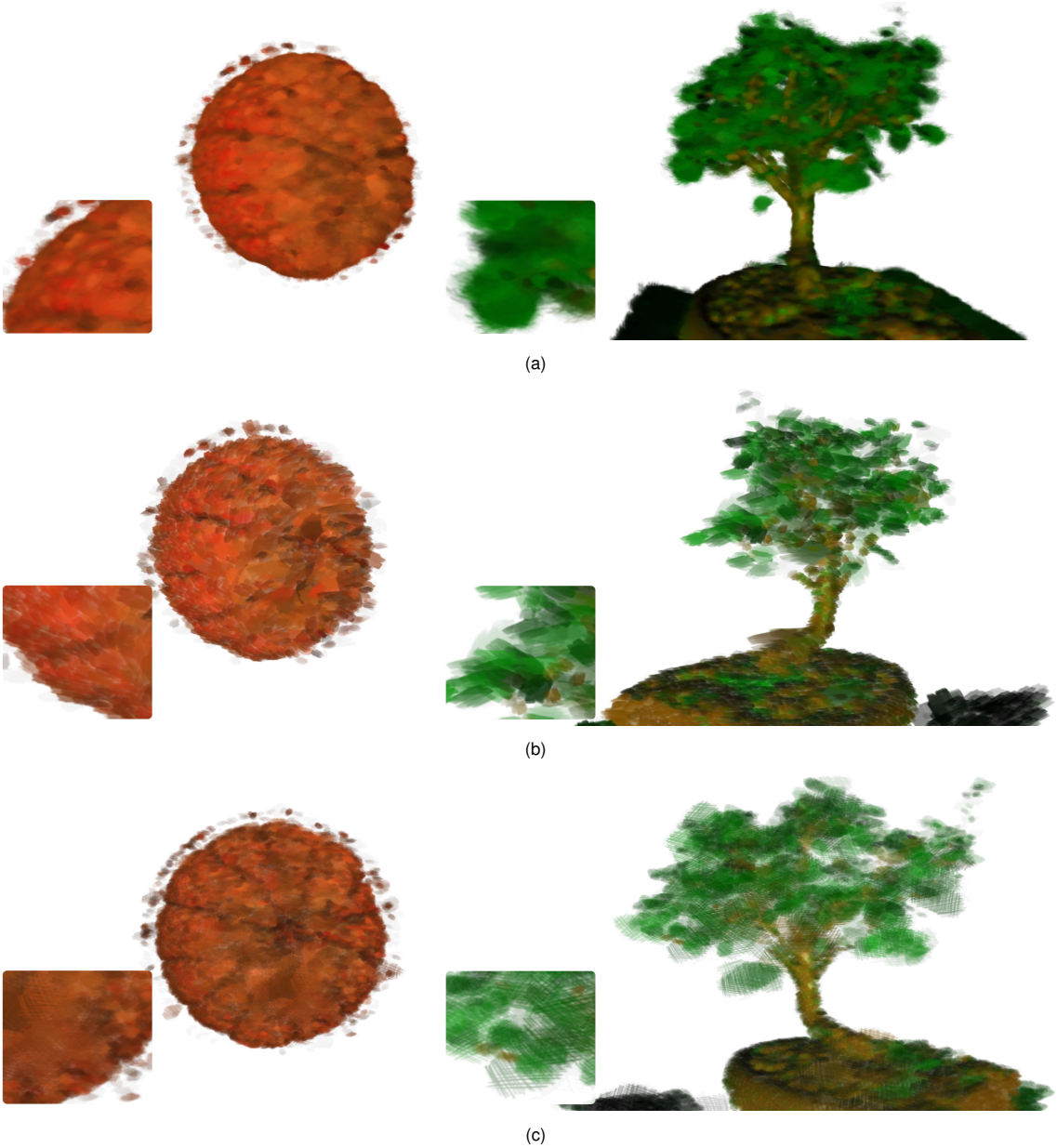


Figure 5.5: Examples of styles achieved with used-defined values and resources.

Chapter 6

Conclusions

We proposed a real-time painterly rendering pipeline for volumetric data-sets. The main goal was to improve upon the work of Lu [28] et al. and Sperl [48] by implementing real-time stylization of volume renderings using surface properties, as proposed by Meier [32] et al. In order to extract the necessary information from the volume, we implemented a real-time GPU iso-surface raycaster similar to the one discussed by Hadwiger [16] et al. The extracted information is passed down the pipeline where it is used to generate strokes and their properties such as position, colour, size and orientation. For this, we need information on the iso-surface positions, gradients, optical flow and so on.

The main challenge for the development of this pipeline was the implementation of recent technology such as the one described by Lu [28] et al. based on a high quality yet notoriously expensive technique such as raycasting while maintaining an interactive frame rate.

Having finished implementing this pipeline, we have found that we were successful not only at achieving visually satisfactory output that mimics hand-made paintings but also at keeping the whole system real-time. From the resulting images, we can notice that its not only colour that is expressed as information on the volume but also surface topology through stroke orientation and size. Seeing the pipeline in action, we can see that temporal incoherency problems are reduced since strokes are persistent along time, yet not static on the screen, avoiding what is often referred to as the *shower door* effect (see section 3.5.4 for details).

The implemented pipeline runs smoothly on current generation hardware. The examples shown ran at an average of over 60 frames per second, rendering a 256x256x256 8-bit volume data-set at a final resolution of 1280x800 on Intel Core 2 Duo E6300 CPU at 3.15GHz with 4GB RAM and an NVIDIA Geforce GTX470 using Microsoft DirectX 10 with an average of 30.000̃50.000 strokes on screen, although the system supports a much higher number of strokes.

We are pleased with the stylization achieved using our pipeline and believe this is a valuable tool for artists and designers to express their creativity for emergent visual data structures such as volumes (for relevant work on volume information applied to the entertainment industry, see Laine [22] et al.), making use of the available user-definable information that controls everything from colours to stroke distribution and size.

6.1 Future Work

Still, we believe that further work can be done in this area. Such work has not been currently implemented either because of time constraints or because the subject might lay outside the scope of the project. The following sections describe the different areas we consider of higher importance for further research and/or implementation.

6.1.1 Optimizing GPU Raycaster

Examining the implementation of the proposed project as a whole, it is clear that the greatest bottleneck of the pipeline currently is the volume raycasting stage.

Although care was taken to optimize the GPU raycaster in order for the proposed system to run in real-time, such as *early ray termination* or iso-surface extraction, further optimizations would lie outside the scope of the project and thus were not considered during implementation.

Optimizations such as *empty-space skipping* or *swizzling* are just two examples of the many possible to implement in order to increase the performance of the system even further. For more information on optimization techniques for GPU raycasters, please refer to section 2.3.4 and/or to the work of Krüger [20] et al. and Engel [11] et al. on volume rendering optimization techniques for the GPU.

6.1.2 Colour mixing using Volume Contributions

While the current implementation of the system only works on extracted iso-surfaces, many volume representations are desirable due to their capability to show several layers of information at the same time and specify which properties are of interest and which are not.

Taking into account all valid contributions from a raycast raises several interesting challenges such as stroke limitations. The number of strokes necessary to represent all data would increase exponentially, not to mention stroke overlapping, that is, how to deal with strokes from different layers in the volume that share the same area on screen.

We would like to research further into painterly rendering using the various contributions of the volume data-set to define a final pixel. A system could receive all contributions of the volume for a given pixel instead of a certain surface value. These contributions, and their order, could then be simulated as mixing or overlaying paint in order to calculate the final colour for a given region of the volume on screen.

6.1.3 Curvature-based strokes using image-space flow advection

The most interesting area of research that did not make it into this project is the study of the contribution of curvature information to stroke orientation, in order to optimize how stroke orientations would map the surface. Previous research in the use of curvature for volume rendering include real-time curvature calculation, visualization and contribution to non-photorealistic rendering techniques.

(a) Curvature Colour Mapping. (b) Curvature Ridge and Valley Lines. (c) Curvature Flow.

Figure 6.1: Curvature-based shading, courtesy of Hadwiger [16] et al.

Hadwiger [16] et al. proposed curvature-based shading of iso-surfaces extracted from volume data. In their work, they approach rendering optimizations, gradient calculation and more importantly, efficient curvature calculation (based on their previous work [47]) as well as advanced curvature-based shading effects.

Curvature colour mapping allows control of shading based on principal, mean and/or gaussian curvature through a one or two dimensional transfer function. An example of this would the to use a 2D transfer function that highlights ridge and valley structures on the surface (see figure 6.1(b)).

Sigg [47] et al. proposed a technique for fast third-order texture filtering where Sigg and Hadwiger calculate the first and second order partial derivatives in real-time on the GPU.

The first order partial derivatives are approximated using cubic B-splines, that is, the original data is convolved with the derivative of the proposed filter kernel in order to reconstruct the cubic B-spline's derivative. The derived values, also known as gradients (see equation 6.1), are used as surface normals for illumination.

$$g = \nabla f = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)^T . \tag{6.1}$$

In order to efficiently compute the second order partial derivative on the GPU, they create the respective three dimensional Hessian matrix (see equation 6.2) along with the first order derivative computations.

$$H = \nabla g = \begin{pmatrix} \frac{\partial^2 f}{\partial x^2} & \frac{\partial^2 f}{\partial x \partial y} & \frac{\partial^2 f}{\partial x \partial z} \\ \frac{\partial^2 f}{\partial y \partial x} & \frac{\partial^2 f}{\partial y^2} & \frac{\partial^2 f}{\partial y \partial z} \\ \frac{\partial^2 f}{\partial z \partial x} & \frac{\partial^2 f}{\partial z \partial y} & \frac{\partial^2 f}{\partial z^2} \end{pmatrix} . \tag{6.2}$$

Surface curvature information can be computed with the calculated gradients and Hessian matrix by evaluating a tricubic convolution filter for each of the nine required components (three values for gradients and six for the Hessian matrix, due to symmetry). Hadwiger [16] et al. go into more detail on how to extract principal curvature magnitudes and directions. The principal curvature magnitudes are extracted from the two eigenvalues of the tangent space projection of the normalized Hessian matrix (for more information on how to extract curvature magnitudes and directions please refer to Hadwiger's [16] chapter on differential surface properties).

In order to visually display the curvature information extracted from the surface, Hadwiger et al. render the curvature directions based on image-based flow visualization by computing the advection of each pixel according to the corresponding vector field of principal curvature directions.

Other approaches use curvature information to improve visual cues for expressing surface topology in non-photorealistic rendering techniques. Vanderhaeghe [51] et al. proposed a dynamic drawing algorithm for interactive painterly rendering based on the work of Meier [32] et al. that uses the surface's principal curvature to orient thick strokes in order to better represent surface shapes using painterly rendering.

We would like to experiment improving our representation of surface topology and brush stroke direction by taking principal curvature information into account. We believe principal curvature direction may provide a powerful and intuitive way of visualizing surface shape and the aforementioned techniques provide us with feasible alternatives to perform these computations while remaining at interactive frame rates.

6.1.4 User Perception Tests

As further work, we would like to carry out experiments in order to test how well our results match the user's expectations for a human-made painterly representation of the same shapes. We also intend to test how fast and with how much accuracy users can identify specified surface properties on the result based on the available visual cues such as stroke size, density and orientation.

Finally, and more importantly, we intend to perform tests on a certain set of participants in order to determine how our render style affects the recognition speeds of the users, as proposed and performed by Niall Redmond and John Dingliana [41]. In their experiment, Niall et al. studied the gaze behaviour (using eye-tracking) and reaction times of twelve participants when instructed to click upon a particular textured sphere as quickly as possible. This experiment used one hundred scenes each of which contained one hundred and fifty randomly generated spheres. Twenty of the scenes used normal 3D local illumination rendering while the other eighty used four different types of non-photorealistic abstraction. User's reaction times were averaged for each different render style and an ANOVA (Analysis of variance between groups) was performed on the results. These results found that the average recognition speeds for abstracted scenes (1.45s) were faster than normally rendered images (1.64s), as well as which of the chosen styles performed the best.

Appendix A

Pseudo-random Number Generator on the GPU

A pseudorandom number generator, or random bit generator, is an algorithm that generates sequences of seemingly random numbers. These sequences though are entirely deterministic.

In order to implement a pseudorandom number generator on the GPU we populate a texture offline with precalculated pseudorandom values for each available texel. This texture is passed onto the necessary shaders, and applied to a texture sampler that wraps the texture coordinates around their domain $t \in [0, 1]$. When sampling for random numbers in the texture, the appropriate texture coordinates are generated as a function of both the screen-space coordinates where the lookup is being performed and the current time. The result from these lookups should give us a random sequence of numbers long enough to accommodate the number of queries demanded per frame.

Appendix B

Glossary

ANOVA (Analysis of variance between groups): a collection of statistical models, and their associated procedures, in which the observed variance is partitioned into components due to different sources of variation.
Cel-shading: a type of non-photorealistic rendering designed to mimic the style of traditional 2D animation.
CPU (Central processing unit): the component of a computer system that carries out the instructions of a computer program; the primary element carrying out the computer's functions.
CT (Computer Tomography): medical imaging method that processes geometry to generate a three-dimensional image of the inside of an object from a large series of two-dimensional X-ray images taken around a single axis of rotation.
Geometry Shader: a graphics processing function that receives output from vertex shaders and emit zero or more primitives, passing them along towards the pixel shader for rendering and/or streaming them back into memory.
GPU (Graphics Processing Unit): a specialized microprocessor that offloads and accelerates 3D or 2D graphics rendering from the CPU.
Gradient: a vector field which points in the direction of the greatest rate of change of the scalar field.
Hatching: a technique used in drawing wherein an artist expresses shapes using closely spaced parallel lines that follow the curvature of the forms they are used to describe.
Image-space: object coordinates are defined in the image coordinate system.
Iso-surface: a set of voxels that describe a surface whose value is contained within a specified threshold.
Iso-value: a value that specifies voxels belonging to a certain iso-surface.
MRI (Magnetic resonance imaging): noninvasive medical imaging technique used in radiology to visualize detailed internal structure and limited function of the body.
Octree: a common 3D space partitioning structure; a tree data structure in which each internal node has exactly eight children.
Painterly Rendering: a stylization technique that mimics the visual appearance of a hand-made

paintings.

Phong: an empirical model of local illumination that describes the interactions between light and a surface as a combination of ambient, diffuse and specular reflectivity components.

Pipeline: a set of data processing elements connected in series, so that the output of one element is the input of the next one.

Pixel Shader: a graphics processing function that computes the final output (usually colour) and other attributes of each pixel.

Polygon: a closed planar path composed of a finite number of sequential line segments (often triangles) used as a primitive entity for rendering of more complex objects.

Posterization: conversion of a continuous gradation of tone to several regions of fewer tones, with abrupt changes from one tone to another.

Pseudocode: a compact and informal high-level description of an algorithm intended for human reading.

Render call: a full execution of the graphical rendering pipeline.

Reprojection: tracking the position of a pixel in the current and previous frame through its current and previous transformations.

Shower door effect: constant brush-strokes positions on screen result in the distortion of the underlying shape with a certain static texture as if it were being viewed through textured glass.

Sobel filter: a discrete differentiation operator, computing an approximation of the gradient of the image intensity function (used on edge detection algorithms).

Texel: a single data value in a texture with a respective texture coordinate.

Texture: an image held in memory that can contain colour (or other) information and can be applied to geometry using texture mapping.

Transfer Function: a function that maps one or more input values into visual information such as colour.

Vertex: a point which describes the corners of polygon primitives.

Vertex Shader: a graphics processing function that manipulates each vertex's 3D properties and positions them from virtual space to the 2D coordinate at which it appears on the screen.

Viewport: the 2D rectangle used to project the 3D scene to the position of a virtual camera.

Volume (Volumetric Data): 3D discretely sampled fields of data.

Voxel: a single data value on a 3D data field (volume).

Watercolor: the medium in which the paints are made of pigments suspended in a water soluble vehicle.

Appendix C

Painterly Renderings

Figure C.1: Bonsai Tree (256x256x256 8-bit)

Figure C.2: Orange (256x256x64 8-bit)

Figure C.3: Lobster (301x324x56 8-bit)

Bibliography

[1] Shiben Bhattacharjee and P. J. Narayanan. Real-time painterly rendering of terrains. In *ICVGIP '08: Proceedings of the 2008 Sixth Indian Conference on Computer Vision, Graphics & Image Processing*, pages 568–575, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Adrien Bousseau, Matt Kaplan, Joëlle Thollot, and François X. Sillion. Interactive watercolor rendering with temporal coherence and abstraction. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 141–149, New York, NY, USA, 2006. ACM.

[3] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *VVS '94: Proceedings of the 1994 symposium on Volume visualization*, pages 91–98, New York, NY, USA, 1994. ACM.

[4] Andrew Corcoran, Niall Redmond, and John Dingliana. Technical section: Perceptual enhancement of two-level volume rendering. *Comput. Graph.*, 34(4):388–397, 2010.

[5] Cassidy J. Curtis. Loose and sketchy animation. In *SIGGRAPH '98: ACM SIGGRAPH 98 Electronic art and animation catalog*, page 145, New York, NY, USA, 1998. ACM.

[6] Philippe Decaudin. Cartoon looking rendering of 3D scenes. Research Report 2919, INRIA, June 1996.

[7] Oliver Deussen and Thomas Strothotte. Computer-generated pen-and-ink illustration of trees. In *SIGGRAPH '00: Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 13–18, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co.

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

[9] David Ebert and Penny Rheingans. Volume illustration: non-photorealistic rendering of volume models. In *VIS '00: Proceedings of the conference on Visualization '00*, pages 195–202, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press.

[10] Gershon Elber. Line art illustrations of parametric and implicit forms. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):71–81, 1998.

[11] 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.

[12] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[13] Moritz Gerl. *Volume Hatching for Illustrative Visualization*. VDM Verlag, Saarbrücken, Germany, Germany, 2008.

[14] Bruce Gooch, Greg Coombe, and Peter Shirley. Artistic vision: Painterly rendering using computer vision techniques. pages 83–90. ACM Press, 2000.

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

[16] C. Scharsach H. Bhler K. Hadwiger, M. Sigg and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. In *In Proceedings of Eurographics 2005*, pages 303–312, 2005.

[17] Paul Haeberli. Paint by numbers: abstract image representations. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 207–214, New York, NY, USA, 1990. ACM.

[18] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[19] Aaron Hertzmann. Painterly rendering with curved brush strokes of multiple sizes. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 453–460, New York, NY, USA, 1998. ACM.

[20] J. Krüger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *VIS '03: Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, Washington, DC, USA, 2003. IEEE Computer Society.

[21] Philippe Lacroute and Marc Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 451–458, New York, NY, USA, 1994. ACM.

[22] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 55–63, New York, NY, USA, 2010. ACM.

[23] Adam Lake, Carl Marshall, Mark Harris, and Marc Blackstein. Stylized rendering techniques for scalable real-time 3d animation. In *NPAR '00: Proceedings of the 1st international symposium on Non-photorealistic animation and rendering*, pages 13–20, New York, NY, USA, 2000. ACM.

[24] Hyunjun Lee, Sungtae Kwon, and Seungyong Lee. Real-time pencil rendering. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 37–45, New York, NY, USA, 2006. ACM.

[25] M. S. Levoy. *Display of surfaces from volume data*. PhD thesis, Chapel Hill, NC, USA, 1989.

[26] 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.

[27] Aidong Lu, Christopher J. Morris, David S. Ebert, Penny Rheingans, and Charles Hansen. Non-photorealistic volume rendering using stippling techniques. In *VIS '02: Proceedings of the conference on Visualization '02*, pages 211–218, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Jingwan Lu, Pedro V. Sander, and Adam Finkelstein. Interactive painterly stylization of images, videos and 3d animations. In *I3D '10: Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*, pages 127–134, New York, NY, USA, 2010. ACM.

[29] Thomas Luft and Oliver Deussen. Real-time watercolor illustrations of plants using a blurred depth test. In *NPAR '06: Proceedings of the 4th international symposium on Non-photorealistic animation and rendering*, pages 11–20, New York, NY, USA, 2006. ACM.

[30] Eric B. Lum and Kwan-Liu Ma. Non-photorealistic rendering using watercolor inspired textures and illumination. In *PG '01: Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, page 322, Washington, DC, USA, 2001. IEEE Computer Society.

[31] Lee Markosian, Michael A. Kowalski, Daniel Goldstein, Samuel J. Trychin, John F. Hughes, and Lubomir D. Bourdev. Real-time nonphotorealistic rendering. In *SIGGRAPH '97: Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 415–420, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[32] Barbara J. Meier. Painterly rendering for animation. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 477–484, New York, NY, USA, 1996. ACM.

[33] Klaus Mueller and Roger Crawfis. Eliminating popping artifacts in sheet buffer-based splatting. In *VIS '98: Proceedings of the conference on Visualization '98*, pages 239–245, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.

[34] Diego Nehab, Pedro V. Sander, Jason Lawrence, Natalya Tatarchuk, and John R. Isidoro. Accelerating real-time shading with reverse reprojection caching. In *Graphics Hardware*, August 2007.

[35] N. Neophytou and K. Mueller. Gpu accelerated image aligned splatting. pages 197 – 242, jun. 2005.

[36] Csebfalvi B. König A. Neumann, L. and E. Gröller. Gradient estimation in volume data using 4d linear regression. Technical Report TR-186-2-00-03, Institute of Computer Graphics and Algorithms, Vienna University of Technology, Favoritenstrasse 9-11/186, A-1040 Vienna, Austria, February 2000.

[37] Hubert Nguyen. *GPU Gems 3*. Addison-Wesley Professional, 2007.

[38] Oscar Meruvia Pastor, Bert Freudenberg, and Thomas Strothotte. Real-time animated stippling. *IEEE Comput. Graph. Appl.*, 23(4):62–68, 2003.

[39] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005.

[40] Bui Tuong Phong. *Illumination for computer-generated images.* PhD thesis, 1973.

[41] Niall Redmond and John Dingliana. Evaluation of non-photorealistic abstraction techniques in influencing user behaviour. In *APGV '08: Proceedings of the 5th symposium on Applied perception in graphics and visualization*, pages 202–202, New York, NY, USA, 2008. ACM.

[42] Niall Redmond and John Dingliana. A hybrid technique for creating meaningful abstractions of dynamic 3d scenes in real-time. In *WSCG '08: The 16th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2008*, pages 477–484. WSCG, 2008.

[43] Niall Redmond and John Dingliana. Investigating the effect of real-time stylisation techniques on user task performance. In *APGV '09: Proceedings of the 6th Symposium on Applied Perception in Graphics and Visualization*, pages 121–124, New York, NY, USA, 2009. ACM.

[44] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 197–206, New York, NY, USA, 1990. ACM.

[45] Michael P. Salisbury, Sean E. Anderson, Ronen Barzel, and David H. Salesin. Interactive pen-and-ink illustration. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 101–108, New York, NY, USA, 1994. ACM.

[46] Rezwan Sayeed and Toby Howard. State of the art non-photorealistic rendering (npr) techniques. In *Theory and Practice of Computer Graphics 2006*, pages 89–98, 2006.

[47] C. Sigg and M. Hadwiger. Fast third-order texture filtering. In *GPU Gems II*, pages 313–329. Addison Wesley, 2005.

[48] Daniel Sperl. Künstlerisches Rendering für Echtzeit-Applikationen. Master's thesis, Fachhochschule Hagenberg, 2003. In German.

[49] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. pages 187 – 241, jun. 2005.

[50] Thomas Strothotte and Stefan Schlechtweg. *Non-photorealistic computer graphics: modeling, rendering, and animation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[51] David Vanderhaeghe, Pascal Barla, Joëlle Thollot, and François X. Sillion. A dynamic drawing algorithm for interactive painterly rendering. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Sketches*, page 100, New York, NY, USA, 2006. ACM.

[52] Lee Westover. Footprint evaluation for volume rendering. In *SIGGRAPH '90: Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, New York, NY, USA, 1990. ACM.

[53] Georges Winkenbach and David H. Salesin. Rendering parametric surfaces in pen and ink. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 469–476, New York, NY, USA, 1996. ACM.

[54] J. Zhou and K. Tönnies. State of the art for volume rendering. Technical Report TR-ISGBV-03-02, Institute for Simulation and Graphics, University of Magdeburg, 2004.