UNIVERSITÄT KOBLENZ · LANDAU
Fachbereich 4: Informatik

Fraunhofer SINGAPORE

Accelerating Volume Rendering for Virtual Reality Applications

Masterarbeit

zur Erlangung des Grades Master of Science (M.Sc.)
im Studiengang Computervisualistik

vorgelegt von
Arend Buchacher

Erstgutachter: Prof. Dr.-Ing. Stefan Müller
(Institut für Computervisualistik, AG Computergraphik)

Zweitgutachter: Dr.-Ing. Marius Erdt
(Fraunhofer Singapore, Visual and Medical Computing)

Koblenz, im Juli 2017

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Institut für Computervisualistik
AG Computergraphik
Prof. Dr. Stefan Müller
Postfach 20 16 02
56 016 Koblenz
Tel.: 0261-287-2727
Fax: 0261-287-2735
E-Mail: stefanm@uni-koblenz.de

UNIVERSITÄT KOBLENZ · LANDAU

Fachbereich 4: Informatik

Aufgabenstellung für die Masterarbeit
Arend Buchacher (Matr.-Nr. 211 100 073)

Thema: **Beschleunigungs- und Bildsyntheseverfahren zur effizienten Darstellung medizinischer Volumendatensätze in Virtual Reality**

In keinem Bereich der Informatik hat sich die Hardware so rasant entwickelt wie im Bereich der Computergraphik. In den unterschiedlichsten Gebieten erfahren Virtual Reality Anwendungen durch die hohe Rechenkraft aktueller GPUs in Verbindung mit innovativen Endnutzergeräten wie der HTC Vive großes Interesse. Im medizinischen Kontext bestehen Ansätze, Virtual Reality unterstützend zur Diagnose oder zur Eingriffsplanung heranzuziehen. Insbesondere in Virtual Reality Anwendungen ist das Aufrechterhalten von sehr hohen Bildwiederholungsraten zur Vorbeugung von „Cyber Sickness“ unerlässlich.
Die Herausforderung besteht in der Untersuchung, wie rechenintensive Graphik-Verfahren (wie hier das Ray-Casting) sich mit Hilfe der neuen Möglichkeiten umsetzen lassen und wo die Möglichkeiten und Grenzen der Technik liegen.

Ziel dieser Arbeit ist es, verschiedene Verfahren zur Beschleunigung von GPU-Ray-Casting sowie Bildsyntheseverfahren zum Erzielen hoher Bildwiederholungsraten zu recherchieren und zu analysieren. Darauf aufbauend sollen ausgewählte Verfahren implementiert und miteinander verknüpft werden. Optional ist eine Erweiterung der Anwendung durch prototypische Interaktionsmöglichkeiten zur Exploration medizinischer Volumendaten angedacht, um die potentielle Praxistauglichkeit zu verdeutlichen. Ebenfalls optional ist die Integration in die Visualisierungs-Bibliothek VTK angedacht.

Schwerpunkte dieser Arbeit sind:

1. Einarbeitung in Virtual Reality- und VTK-Programmierung und Recherche bisheriger Ansätze
2. Konzeption und Implementierung der Verfahren
3. Optionale Erweiterungen
4. Demonstration und Bewertung der Ergebnisse
5. Dokumentation

Koblenz, den 9.1.2017

- Arend Buchacher -

- Prof. Dr. Stefan Müller-

Zusammenfassung

Mit dem Aufkommen von Head-Mounted Displays (HMDs) der aktuellen Generation erlangt Virtual Reality (VR) wieder großes Interesse im Feld von medizinischer Bildgebung und Diagnose. Exploration von CT oder MRT Daten in raumfüllender Virtual Reality stellt eine intuitive Anwendung dar. Allerdings gilt in Virtual Reality, dass das Aufrechterhalten einer hohen Bildwiederholungsrate noch wichtiger ist als bei konventioneller Benutzerinteraktion, die sitzend vor einem Bildschirm erfolgt. Es existieren starke wissenschaftliche Hinweise, die nahelegen, dass geringe Bildwiederholungsraten und hohe Latenzzeit einen starken Einfluss auf das Auftreten von Cybersickness besitzen. Diese Abschlussarbeit untersucht zwei praktische Ansätze, um den hohen Rechenaufwand von Volumenrendering zu überkommen. Einer liegt in der Ausnutzung von Kohärenzeigenschaften des besonders aufwändigen stereoskopischen Rendering Set-ups. Der Hauptbeitrag ist die Entwicklung und Auswertung einer neuartigen Beschleunigungstechnik für stereoskopisches GPU Raycasting. Zudem wird ein asynchroner Renderingansatz verfolgt, um das Ausmaß von Latenz im System zu minimieren. Eine Auswahl von Image-Warping Techniken wurden implementiert und systematisch evaluiert, um die Tauglichkeit für VR Volumenrendering zu bewerten.

Abstract

With the emergence of current generation head-mounted displays (HMDs), virtual reality (VR) is regaining much interest in the field of medical imaging and diagnosis. Room-scale exploration of CT or MRI data in virtual reality feels like an intuitive application. However in VR retaining a high frame rate is more critical than for conventional user interaction seated in front of a screen. There is strong scientific evidence suggesting that low frame rates and high latency have a strong influence on the appearance of cybersickness. This thesis explores two practical approaches to overcome the high computational cost of volume rendering for virtual reality. One lies within the exploitation of coherency properties of the especially costly stereoscopic rendering setup. The main contribution is the development and evaluation of a novel acceleration technique for stereoscopic GPU ray casting. Additionally, an asynchronous rendering approach is pursued to minimize the amount of latency in the system. A selection of image warping techniques has been implemented and evaluated methodically, assessing the applicability for VR volume rendering.

Publications and Conference Attendance

The work of this thesis has been presented at the *28th Eurographics Symposium on Rendering* and has subsequently been published in the proceedings.

Arend Buchacher and Marius Erdt. Single-Pass Stereoscopic GPU Ray Casting Using Re-Projection Layers. In Matthias Zwicker and Pedro Sander, editors, *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*. The Eurographics Association, 2017

Acknowledgments

This thesis could not have been brought into being without the incredible support I received throughout my studies from all the peers and mentors, friends and family, respected and beloved. In particular, I would like to thank Prof. Dr. Stefan Müller for his generous supervision and attentive teachings that have played an important part at making my studies a gratifying experience. I would also like to thank Dr. Marius Erdt and Fraunhofer Singapore for giving me the opportunity to experience a remarkable country, culture and place of work.

Most of all, I must express my deepest gratitude to my parents, my brother and my significant other Lisa who for years always built confidence in my intentions, who always boosted my morale during wearing phases, and who preservered and bonded together even stronger through lengthy periods of seperation. Thank you!

Contents

1 Introduction

1.1 Motivation

In recent years, efficient stereoscopic rendering has become a field of interest again. Various new consumer products of head-mounted displays (HMDs) for virtual reality (VR) applications are driving this interest. Many VR applications are aimed at entertainment or gaming based interaction and common acceleration methods for polygonal meshes are implemented. In the medical field VR is regaining interest where in many cases, volumetric medical data needs to be displayed [31, 2]. Volumes are commonly rendered directly using GPU accelerated image-based ray casting techniques in which the volume is sampled along rays shot from the eye's center. Intuitively to produce a stereo pair of images the volume is rendered once from the left and once from the right eye, effectively doubling the rendering time. Volume rendering is computationally very expensive as it is. In VR applications requirements for constant and high frame rate is stronger than ever. Exposure to low frame rate and high latency might have direct negative physiological effects on the user. The condition, also called virtual reality sickness or cybersickness, causes symptoms like headache and nausea. Best practices for volume rendering in virtual reality have yet to be established, therefore this property is often overlooked. This thesis is motivated by the eagerness to expedite the potential of VR volume rendering with the determination to leverage the risk of cybersickness therein.

1.2 Goals

The driving question of this thesis is: How is it possible to do interactive volume rendering in virtual reality without exposing the user to low frame rates and latency? The hypothetical use-case this thesis aims for is the inspection of medical volume data in a room-scale VR environment. The goals to the identified problems in this context are:

- Find ways to maintain a constant, head-tracked image stream of (optimally) 90 frames per second.
- Find ways to accelerate ray casting while maintaining overall interactivity of the system.
- Find ways to accelerate the stereoscopic image generation process.

To meet the first goal of a constant low-latency image stream, this thesis evaluates the implementation of image warping. With this a low frame rate is compensated for by generating in-between frames from the last rendered image. To meet the second goal of fast, yet interactive rendering a selection of acceleration methods that do not rely on pre-computations and elaborate data-structures is implemented. The third goal arised from the very specific setup imposed by rendering for a virtual reality HMD. That is rendering two images per frame, one for each eye. More

often in stereoscopic rendering some sort of coherency between the views allows for accelerated rendering. To this end, this thesis proposes a novel technique to speed up stereoscopic GPU ray casting.

1.3 Outline

This thesis is structured as follows. In Section 2 the fundamental principles and methods to this work are established. It establishes terminology that is tightly linked to the field of virtual reality and recapitulates fundamental aspects of volume rendering. Section 3 gives an overview of current advancements and related work concerning virtual reality in a medical environment, volume rendering acceleration, image warping and stereoscopic rendering techniques. Additionally, a brief introduction to the condition of cybersickness and current research on the topic is given. Following this, Section 4 introduces the novel stereoscopic GPU ray casting technique. The proposed method is thoroughly described, also discussing its limitations. Section 5 describes a system design for volume rendering in virtual reality that employs the knowledge gathered before. The concept describes an asynchronous rendering system which on one hand consists of an accelerated volume rendering component and on the other hand of a programmable display layer that handles the image warping. Section 6 evaluates implementations of both the stereo method and the described VR volume rendering system. Finally, Section 7 concludes the thesis, first discussing areas for future work and lastly summarizing the work.

2 Fundamentals

2.1 Virtual Reality Terminology

Ivan Sutherland [61] described the idea of using advanced display technology to immerse users in virtual worlds. However, the current field of virtual reality (VR) is broad and finding a universal definition is still difficult. Adapting the definitions by Rebenitsch [53], *virtual reality* (VR) refers to a simulated environment whose visual content is entirely created by a computer and the appearance of the environment is altered by the participant's real-world actions. Furthermore, only the usage of a position-tracked *head-mounted display* (HMD), a visor display that places a small screen that renders two seperate images of a stereoscopic view before the eyes, is considered. *Latency*, in the context of this thesis, refers to the *motion-to-photons latency*, which is the time it takes for a user motion to become visible on screen. This latency comprises the time until the hardware reports the updated tracking information, the time until the rendering component finishes using this information and the time until the display finally lights up (emitting photons) to show the image. For current generation HMDs that have a very high update rate for tracking information, the largest portion of latency is likely to arise from the rendering component.

HMD	OR DK1	OR DK2	OR CV	HTC Vive
Resolution (eye)	640×800	960×1080	1080×1200	1080×1200
Update rate	60 Hz	75 Hz	90 Hz	90 Hz
Field of view	$\sim 90^\circ$	100°	110°	110°
Tracking	Rotation	Position	Room-scale	Room-scale
Year	2013	2014	2016	2015

Table 1: *Summary of technical specifications of recent consumer product HMDs.*

VR hardware historically consisted of complicated setups with expensive equipment. However, state-of-the-art VR HMDs are now affordable enough for the mass market. The current consumer products vary in functionality, but the overall trend shows increasing display resolution per eye, update rate, horizontal field of view and positional tracking capabilities. Table 1 summarizes these for a selection of popular, recent HMD products: the successive Oculus Rift (OR) development kits (DK1 and DK2) and consumer version (CV) and a competing product, the HTC Vive. The trend is suggesting that future HMDs will aim for *room-scale* positional tracking with six degrees of freedom. In this type of VR setup, the user is able to walk freely in a room-sized area while the head movement is tracked and mapped to the virtual camera. This thesis uses the exemplary HMD specifications of the HTC Vive, however the presented techniques are generally agnostic of the actual HMD employed.

2.2 Volume Rendering Fundamentals

Volume rendering is a broad and generally well-researched field. Over the years, works of reference have been compiled [36, 26, 28, 21] time and again.
Generally, volume data is simply scalar values arranged in a 3D grid. Cells of the grid are refered to as *voxels*. Slicewise inspection by mapping the scalar values to a grayscale is often possible, but unintuitive and may require special training and experience. With indirect volume rendering, the volume is processed to generate an intermediate surface representation of the volume that can be visualized using common polygonal rendering methods [28]. A notable example of this is the marching-cubes algorithm by Lorensen and Cline [42]. With direct volume rendering, much more of the information contained within the data can be visualized, by evaluating an optical model for the data [21].

2.2.1 Emission-Absorption Model

The most common optical model for direct volume rendering is the *emission-absorption model* by Kajiya and Von Herzen [27] (cf. Figure 1). It aims at solving the *volume rendering integral* (Equation 5), a physically-based optical model. In this model, the volume is assumed to consist of particles that emit light, and absorb incoming light. However, there is no scattering of incoming light.

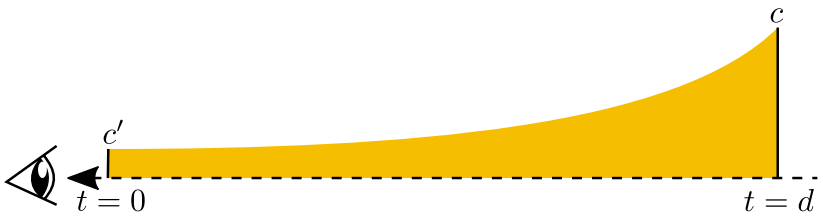


Figure 1: *Partial absorption of emitted energy c along the distance d.*

Let $\vec{x}(t)$ denote a ray that is parametrized by the distance t from the eye. The scalar value that corresponds to a position along the ray is denoted by $s(\vec{x}(t))$. In the emission-absorption model, *absorption coefficients* $\kappa(s)$ and *emissive colors* $c(s)$ are integrated. For simplicity, coefficients are denoted as functions of the eye distance t:

$$c(t) := c(s(\vec{x}(t))) \tag{1}$$

$$\kappa(t) := \kappa(s(\vec{x}(t))) \tag{2}$$

The absorption along a ray is modeled such that once emitted energy c is continously absorbed such that only a remainder c' reaches the eye. For a constant absorption parameter κ, this amounts to the following equation where d denotes the distance the emitted light travelled through this medium.

$$c' = c\mathrm{e}^{-\kappa d} \tag{3}$$

Taking into account that absorption may change along the ray, it is calculated as the integral over the absorption coefficients in the interval of d, also called *optical depth*.

$$\tau(d_1, d_2) = \int_{d_1}^{d_2} \kappa(\hat{t})\partial\hat{t} \tag{4}$$

Thus, the total amount of radiant energy C that reaches the eye from this direction resolves to the following.

$$C = \int_0^\infty c(t)\mathrm{e}^{-\tau(0,t)}\partial t \tag{5}$$

In practice, this integral is solved numerically, to which ray casting is one of the most direct and straight forward methods.

2.2.2 Volume Ray Casting

The idea of ray casting [38] is to substitute the integral of Equation 5 by a Riemann sum to evaluate it numerically. To do so, the optical depth (Equation 4), is approximated by

$$\tau(0,t) \approx \tilde{\tau}(0,t) = \sum_{i=0}^{\left\lfloor \frac{t}{\Delta t} \right\rfloor} \kappa(i\Delta t)\Delta t, \tag{6}$$

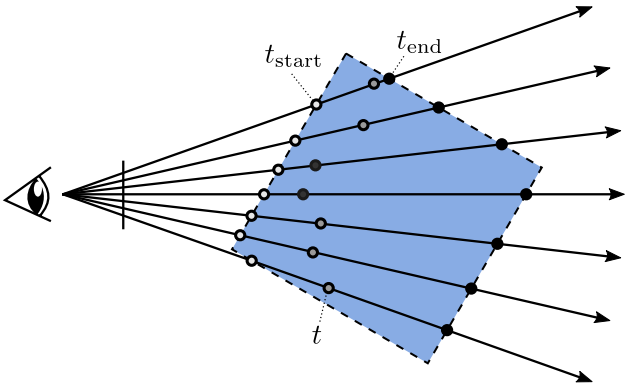


Figure 2: *Ray casting scheme. Rays are traversed in parallel between t_{start} and t_{end} and may be terminated early (dark gray circles).*

where Δt denotes the distance between successive samples taken along the ray. In presented algorithms $\vec{X}_i = \vec{x}(i\Delta t)$ denotes the position of sample i.

$$A_i = 1 - \mathrm{e}^{-\kappa(i\Delta t)\Delta t} \tag{7}$$

$$C_i = c_i(i\Delta t)\Delta t \tag{8}$$

With Equation 7 defining the *opacity* and Equation 8 defining the color of a sample, the volume rendering integral can be approximated as follows, where n denotes the total number of samples.

$$\tilde{C} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1}(1 - A_i) \tag{9}$$

This equation can be evaluated iteratively by *alpha blending*, using A_i as the respective α-value. In front-to-back order, an iterative formulation that evaluates Equation 9 by stepping i from 1 to n is given by

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{10}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i. \tag{11}$$

At each iteration, the new values of C'_i and A'_i are calculated from the color C_i and opacity A_i at the current location i, and the composited color C'_{i-1} and opacity A'_{i-1} (cf. [21, Chap. 1.2.4]). Additionally, it is $C'_0 = 0$ and $A'_0 = 0$, and the colors C_i are pre-multiplied with their associated opacity A_i. For the sake of brevity, in presented algorithms equations 10 and 11 are implemented by the function *blend(C', C)* which returns the updated, composited color, while the opacity is assumed to be included as a color component C_α. The big advantage of front-to-back compositing over the alternative back-to-front compositing is the potential to implement the *early ray termination* optimization. With this, the traversal along a ray is terminated when the cumulative α-value reaches 1, allowing to skip the remaining samples that can't influence the final color anymore. Conversely, the *space leaping* optimization skips empty space between the eye and the first non-transparent position along the ray.

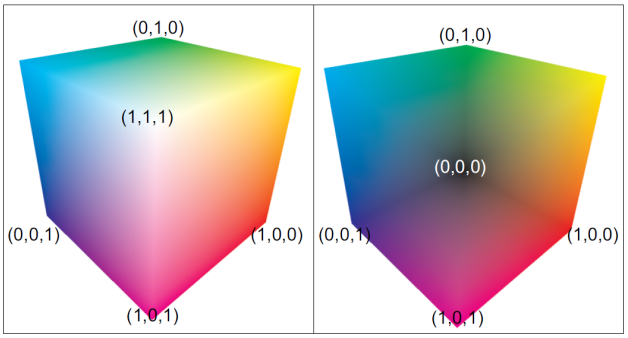


Figure 3: *Texture coordinates of front faces (left) and back faces (right) of the volume bounding box determine the entry and exit points of each ray. [36, Fig. 4].*

For hardware-accelerated volume rendering, usually the volume data is loaded into a 3D texture and fragments are generated by rasterizing a *proxy geometry* with interpolated texture coordinates. For image-order volume rendering, like ray casting, this may simply be the volume's bounding box. GPU implementations of ray casting as proposed by Roettger et al. [54] and Krüger and Westermann [36] use the fragment shader to evaluate the colors of many rays in parallel. For each fragment, a ray is traversed through the volume, iteratively evaluating the equations above (cf. Figure 2). The basic approach that is used throughout this thesis, is split into two passes.

1. **Entry and exit point determination**: The volume's bounding box is rendered to write the 3D texture coordinates of the front and back faces to seperate 2D RGBA textures (cf. Figure 3).

2. **Fragment shader ray casting**: The proxy geometry is rasterized again and entry and exit points of the rays are retrieved, to determine the ray direction and number of samples. These are fetched from the textures of the front faces (entry point) and back faces (exit point). Iteratively evaluating equations 10 and 11, the rays are traversed.

2.2.3 Transfer Function

A transfer function assigns optical properties to the abstract data values of a volume. Generally, the purpose of a transfer function is to help identify regions of interest, classify features and distinguish between different regions or matter. it is a function that maps the domain of the input data, e.g. Hounsfield scale of CT data, to the range of RGBA colors. The input can be of arbitrary dimensionality [32], however 1D or 2D transfer functions are often most manageable. Furthermore, a transfer function is commonly implemented as a simple lookup table.

$$f : [s_{\min}, s_{\max}] \mapsto [0,1]^4 \tag{12}$$

Throughout this thesis, a 1D transfer function is used that maps the data range to a range of colors, as defined by Equation 12. Here, $[s_{\min}, s_{\max}] \subset \mathbb{R}$ describes the dynamic range of the volume data. In the implementation, the transfer function is represented by a 1D RGBA texture and data values are mapped linearily to the texture coordinate range $[0,1]$. Then, the color is retrieved by a texture lookup, making use of hardware texture filtering. This approach is also referred to as *post-classification*, as data is first interpolated, then optical properties are assigned. Additionally, *pre-integration* [16] can be used to further improve the accuracy, however post-classification has been sufficient for the purposes of this thesis.

2.2.4 Shading Techniques

The transfer function provides a way to assign optical properties to abstract data of a volume, but doesn't take light conditions into account. Basic local illumination models like the *Phong illumination model* [52] require the calculation of the gradient vector of a given position in the volume. This vector is used as the surface normal to calculate the diffuse and specular terms of the illumination model. Considering light transport through the volume adding shadows can further improve the visual quality of the rendering. Volumetric shadows can be added to the emission-absorption model (cf. Section 2.2.1) by interpreting the emission coefficient as an isotropic reflection term that is evaluated with the incoming radiation of a light source.

$$C = \int_0^\infty c(t) \mathrm{e}^{-\tau(0,t)} \cdot c_l \mathrm{e}^{-\tau(t,\vec{\omega}_l,l_l)} \, \partial t \tag{13}$$

In Equation 13 the function $\tau(t,\vec{\omega},l)$ evaluates the optical depth of length l for a ray beginning at point $\vec{x}(t)$ in direction $\vec{\omega}$, otherwise analogously to Equation 4. Here, $\vec{\omega}_l$ represents the direction towards the light source with color c_l at distance l_l. In the implementation, for each sample point an additional ray towards the light source is cast to evaluate $\tau(t,\vec{\omega},d)$ analogously to Equation 6. A constant $\vec{\omega}_l$ simulates a collimated light source and l is set to a short distance to reduce the computational costs per sample. Other advanced shading techniques involve more efficient ways to simulate volumetric shadows, such as *deep shadow maps* [22] or the usage of pre-computed data structures for efficient dynamic illumination [35].

2.3 Fast Stereo Volume Rendering

The work by Adelson and Hansen [1] and the extension by He and Kaufman [23] describe an efficient method to generate stereoscopic views of a volume using software ray casting. It lays the foundation of the novel GPU approach presented in Section 4.

The method produces a stereo pair of images by rendering the left view using conventional ray casting from the left eye's center. At the same time, the right view is produced by re-projecting each sample point along the left view's rays to the

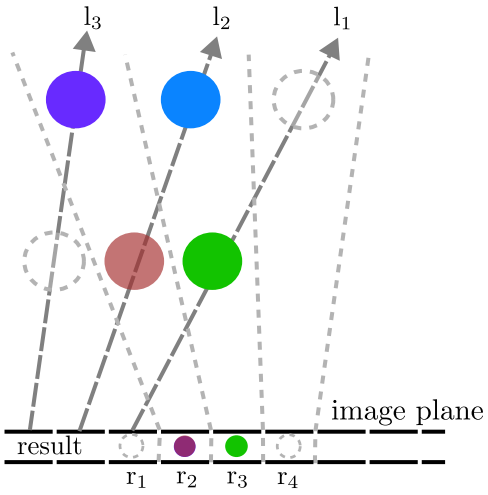


Figure 4: *Scan-line order re-projection of samples along rays (based on [1, Fig. 2]).* l_i*: left view's ray, colored circles: sample colors,* r_j*: right image's pixel. The projective pixel boundaries are indicated by the light gray dashed lines. [9, Fig. 2]*

right view's image and accumulating the color. The right image is thus produced in a fraction of the time, since the computationally expensive sampling and shading part is only performed once. Given that the angle between two corresponding rays is generally small, multiple consecutive samples are re-projected to the same right view's image pixel. The *segment composition* scheme [23] first accumulates a segment of a left view's ray which corresponds to the range of one right view's image pixel before re-projecting its color.

The key observation is illustrated in Figure 4. If rays are evaluated in the correct order, the right view's rays will accumulate the values in the same order. If the left view's rays are processed right to left and each sampled front to back, then the right view's rays will inherently accumulate the values from front to back aswell [1]. This geometric property is also true if perspective projection is used and the image planes are chosen to be coplanar and parallel.

Figure 5 illustrates the most relevant stereoscopic perspective projection geometry properties using the pinhole camera model. A segment between two points $\mathrm{P}(x,y,z)$ and $\mathrm{P}'(x',y',z')$ with $h \leq z' \leq z$ on a ray that is emitted from the left eye E_l is re-projected to a segment of at most length d on the epipolar line. This length is equivalent to the distance between the re-projected start and end points $\mathrm{P}_\mathrm{r}(x_r,y_r,h)$ and $\mathrm{P}_\mathrm{r}{}'(x_r',y_r,h)$ with

$$d = \frac{eh}{z} - \frac{eh}{z'}. \tag{14}$$

Equation 14 is derived from the observation that for any point on a left view's ray,

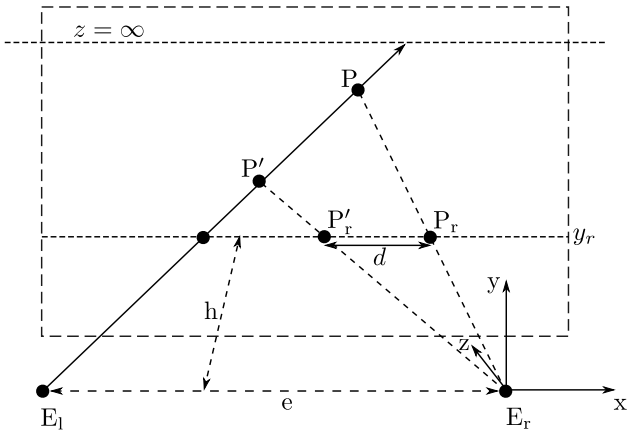


Figure 5: *Stereoscopic perspective projection geometry. Dashed rectangle: projection plane,* E_l*: left eye,* E_r *right eye,* h*: focal length,* e*: eye distance,* P,P′*: sample points on left view's ray,* P_r,P_r'*: re-projected sample points,* y_r*: height of epipolar line, d: distance between re-projected points. [9, Fig. 3]*

for its re-projected x-coordinate x_r it yields

$$x_r = \frac{hx}{z} \tag{15}$$

$$= \frac{h(z\tan\alpha - e)}{z}$$

$$= h\tan\alpha - \frac{eh}{z} \tag{16}$$

where α is defined by the horizontal angle of the ray to the z-axis. The views are setup such that both images lie on the same projection plane and are only offset along the x-axis by eye distance e. If the distance e between the two eyes is small enough, the images can overlap. In any case, for every left view's ray the corresponding epipolar line runs parallel to the x-axis at height y_r on the image plane.

3 Related Work

3.1 Virtual Reality in Medical Applications

The applications for virtual reality in medicine are manifold [55], with use-cases including medical data inspection [19, 31], medical training and teaching [49], surgical planning [10] and others. King et al. [31] proposed a concept for a VR application using an HMD, in which a virtual radiology room is simulated to inspect multiple CT volumes at a time. The volumes are visualized as slices arranged in a semicircle around the user, floating in virtual space. In this setup, typical radiology room tasks, such as finding areas of lesion, are achievable at a fraction of the cost and accessibility of a real radiology room. Gallo et al. [19] use a Nintendo Wi-

imote controller to present interaction methods for virtual 3D volumes. The user-friendly 3D interaction allows to extract volumes-of-interest by intuitively manipulating the cropping box of the volume. The VR setup does not use an HMD, but a semi-immersive stereoscopic projection setup. In this thesis, the fictitious VR application aims at intuitive medical data inspection using an HMD and handheld controllers for 3D interaction, but using 3D volume rendering techniques.

3.2 Volume Rendering Acceleration Techniques

The current state-of-the-art method to visualize volumetric data in 3D at interactive frame rates is GPU ray casting (cf. Section 2.2.2). Many acceleration techniques have been presented, generally addressing different aspects of the rendering process. Additionally, techniques also differ in the requirements on the rendering method and volumetric data at hand. Factors such as whether voxels have a predefined color and transparency mapping or to which extent ray traversal can be altered also influence their applicability.

One of the most advanced data structures for large uncompressed volume data sets is the *GPU voxel database structure* by Hoetzlein [24]. It aims at "combining features for efficient simulation, dynamic topological changes, sparse compression, fast raytracing and very large addressable spaces" [24]. The underlying data structure is based on a sparse hierarchical octree representation of the volume that is closely linked to the *Gigavoxels* data structure by Crassin et al. [12] that also powers the *Voxel Cone Tracing* [13] algorithm. Visualization of large volume data is an active research field for which an overview is given by Beyer et al. [6].

Another popular way to accelerate ray casting is to adaptively reduce the sampling rate based on the volume's density distribution. To some extent, this includes *empty space skipping* and *early ray termination* as the sampling rate is reduced to zero for fully transparent and fully occluded segments of the ray. A more sophisticated technique was presented by Suwelack et al. [62]. Applying a Fourier analysis to the volume, its local frequencies can be used to adaptively change the sampling distance along the ray. At high-frequency changes in the volume, i.e. borders, the sampling distance must be short and at low-frequency changes in the volume, i.e. homogeneous areas, the sampling distance can be longer.

The work by Wang et al. [67] aims at incorporating the notion of GPU hardware to increase memory access times by factoring in low-level texture caches. Due to the fact that memory on a GPU is not layed out in 3D, but nearby memory access on 2D textures is optimized, some sampling directions utilize the GPUs cache better than others. The proposed sampling strategy that requires a GPGPU ray casting implementation is called *3D warp marching*. It adapts the ray traversal such that a warp of shader invocations reads samples for rays that are most favorable concidering the memory layout.

Another promising approach is *foveated rendering* [20] which is based on the idea that rendering quality can be distributed over the image by adapting to the user's gaze. The work by von Mammen et al. [65] provides an adaptive ray tracing

algorithm that decreases quality driven by the limitations of human perception. Using an eye tracker, only pixels that lie within a 10° field of view around the center of view are rendered at high quality. Pixels outside a 20° field of view are rendered at lower quality.

In general, acceleration techniques for volume rendering have different prerequisites, make different assumptions and generate memory or computational overhead of their own. While the asynchronous rendering system proposed in Section 5 reduces the importance of a fast volume renderer, it is still highly advantageous. For this reason, the ray caster implemented for this thesis incorporates two selected acceleration techniques that are easy to implement, provide on-the-fly acceleration and have very lenient and common prerequisites. One is the *space leaping* technique by Mensmann et al. [46], the other is the *level-of-detail* (LOD) technique based on the proposed algorithm by Weiler et al. [69]. LOD rendering techniques are useful to trade visual quality for performance in accordance to some importance factor, like the proximity to the camera. The work by Weiler et al. [69] proposes to generate multiple levels of the 3D texture of a volume. This hierarchical approximation of the volume data allows to increase the sampling step size with each incremental level. This allows to dynamically adjust the sampling step size and sampled texture level based on the ray's sampling position. In the implementation, the hierarchy construction is handled by hardware 3D texture mipmapping. As volume data often consists of an object of interest within space that would be mapped to transparent matter like air, enclosing the volume as tightly as possible with a proxy-geometry other than the volume's bounding box is often useful. Using the proxy-geometry to define the start and end points of the rays, big areas of samples that are known to be uninfluential are skipped. Mensmann et al. [46] present a method that uses the geometry shader to generate a proxy-geometry on the fly. The proxy-geometry called *occlusion frustums* is generated view-dependently from the preceding frame's *first-hit map* which saves the texture space position at which each ray first hit a non-transparent voxel, similar to a depth map. A geometry shader then generates grid of view-aligned frusta that tightly enclose these positions and rasterizes them from the new view configuration. Thus, the ray starting points are advanced to the fragment's corresponding texture positions.

3.3 Image Warping and Temporal Upsampling

Image warping in the context of HMDs is a technique that can be used to reduce the perceived latency. More generally, image warping can be used for (spatio-)temporal upsampling of image streams of rendered content [15, 70, 8, 41]. The frame rate is improved at very low computational costs by extrapolating information from the last frame. Specifically designed as an image warping VR-architecture, Smit et al. [58] presented the *programmable display layer* (PDL) architecture. The multi-GPU-based system decouples the rendering and displaying tasks, such that rendering runs on one GPU while the other warps the last result to generate intermediate display frames. The technique was further advanced by Smit et al. [57] and adapted

to a single-GPU implementation. The paper by Peek et al. [50] incorporates the same architectural design in a simplified and very efficient image warping technique to improve the appearance of head tracking on HMDs. In subsequent work [51] the technique has been adapted such that it runs the image warping component on an integrated GPU, concurrent to the rendering running on the dedicated GPU. As this form of warping is used to compensate for latency between rendering and displaying the term *asynchronous time warp* may also be used. The paper by van Waveren [63] discusses "the various challenges and different trade-offs that need to be considered when implementing an asynchronous time warp on consumer hardware". One posed challenge is that by the time of writing there has been lacking support for context priorities by OpenGL. Thus, it is difficult to preempt one graphics task (rendering) to run the other (warping). Context priority support has been added to drivers for a small subset of available graphics hardware with exactly this use-case in mind [47]. By now, some of the popular consumer HMD companies have incorporated these notions into their respective HMD driver software in one way or another [4, 37, 5]. However, currently the functionalities are not fully exposed and are instead designed to compensate automatically for frames that are missed by the main application. The asynchronous rendering system described in Section 5 poses a solution to this problem, while exploring multiple suitable warping techniques in the context of volume rendering.

3.4 Stereoscopic Rendering Techniques

One way to efficiently generate alterative views from existing data is image-based rendering. A notable technique commonly used with polygonal rendering are *layered depth images* (LDI) that were first introduced by Shade et al. [56]. For volume rendering where continious transparency attenuation is a key factor to the result image layered surface representations are not as expedient. An LDI adaptation for volumes is *volumetric depth images* (VDI) presented by Frey et al. [18]. The view is subdivided into a sliced representation based on depth. Subsequent frames are produced by rendering frusta that approximate the volume's color and opacity between slices. The frusta must be sorted before rendering which poses the need to perform GPU-CPU synchronization. The recent work by Lochmann et al. [40] yields a similar approach in which the old view is encoded in a piece-wise analytic representation of emission and absorption coefficients. Subsequent frames are produced by sampling this representation along the new rays. In principle, the above methods may be classified as *gathering* approaches in which the resulting image is produced by gathering colors from an intermediate representation of the first view. In contrast, the fast stereo volume rendering technique (cf. Section 2.3) may be classified as a *scattering* approach in which the resulting image is produced by scattering colors to the target image. Additionally, stereo rendering poses just one of many use-cases for the above methods. They address more generally the efficient generation of frames from arbitrary views in a common single-view setup. The work by Hübner and Pajarola [25] addresses the particular case of multi-view

rendering for auto-stereoscopic displays. They refrain from adopting the approach by He and Kaufman [23] as significantly more and stronger artifacts would be expected for N views. Also, the volume is rendered using a texture-based method using viewport-aligned quadrilaterals. The work by Wan et al. [66] implements ideas of Shade et al. [56] and He and Kaufman [23], while extending them to a stereoscopic perspective projection model. However, the volumetric environment is rendered opaquely resulting in a well-defined depth value per pixel.

3.5 Order-Independent Transparency

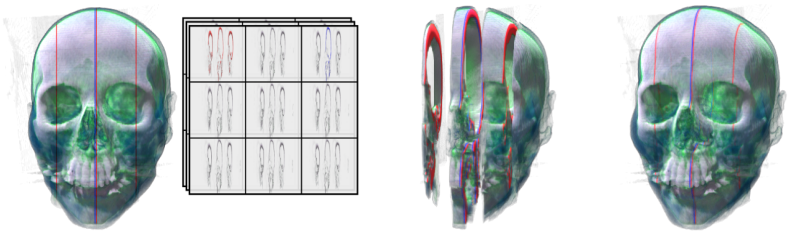
Direct volume rendering acquires pixel colors by tracing paths through matter of varying optical properties and integrating the changes as they appear along the ray. There is also the research field of order-independent transparency (OIT), refering to the order in which surfaces are rasterized. Rendering surfaces in the correct front-to-back or back-to-front order to achieve the correct blend-result is the key challenge. Seasoned techniques like *Depth Peeling* [17] aim at first buffering the color of each intersected surface along a viewing ray. The original method required multiple rendering iterations, with each acquiring the surfaces that lie one position behind the ones acquired by the preceding iteration. Since then, methods to buffer the surface colors in a single-pass have been developed [3, 64] without knowing their final blending-position up-front. The values can later be retrieved, sorted and composed on a per-pixel basis. However, usually only a limited number of the front-most surface colors is available due to memory restrictions. Also, racing conditions between fragments are an emminent issue to this kind of parallel memory access. Alternatively, other work focusses on defining blending operators that are independent of order, but look plausible for phenomena like smoke or non-refractive glass [45]. In a sense, ray casting is feasible because it is a streaming process that does not need to memorize and sort the intersected matters at all. Nevertheless, Section 4 will seize upon many mutual ideas and issues, such as buffering colors to off-screen memory and composing them correctly in a subsequent step.

3.6 Cybersickness

A prevailing problem of virtual reality usage is the so called *cybersickness* [44] which is related to simulator sickness [34], however has been considered to be unique to virtual reality applications and possibly more severe [60]. These terms and some others, including visually induced motion sickness (VIMS), virtual simulation sickness or virtual reality-induced symptoms and effects are often used interchangably. The condition that may occur during and after experiencing virtual environments through an HMD shows similar symptoms to motion sickness, including nausea, headaches and dizziness [53]. An established method to quantify the condition is the *simulator sickness questionnaire* [30], in which probands assess the severity of the known symptoms after using a VR application. The causes of cybersickness are versatile, including inexact alignment of the virtual camera

with the eyes, lengthy duration of exposure, increased field of view and sensory mismatch between the visual input and the vestibular system [53, 39]. The last of which is related to the amount of *latency* in the rendering system. Visual lag (latency) and frame rate were identified as factors of simulator sickness by Kolasinski et al. [34]. The research field on latency reduction for HMDs had already been established [48], however more recent work has been specifically motivated by the link to cybersickness [39, 50]. Additionally, Chen et al. [11] concluded that head-controlled 6-DoF navigation in virtual environments is better than using a joystick. It improves user performance, the sense of immersion and yields a lower occurence of cybersickness. This finding is also reinforced by the results of [39]. There is also other conspicuous research effort, such as the work of von Mammen et al. [65] which deliberately amplifies cybersickness inducing effects in a user study, finding that the users considered their experience in VR to outweigh the negative effects of cybersickness. In the context of this thesis, cybersickness is considered one of the driving motivations why tight latency and frame rate constraints must be upheld with diligence.

4 Single-Pass Stereoscopic GPU Ray Casting



(a) *Left View and Re-projection Texture Array* **(b)** *Image-based Compositing* **(c)** *Composed Right View*

Figure 6: *Principle of the single-pass stereoscopic GPU ray casting method. The left view is rendered via ray casting. (a) Vertical scan-lines of rays re-project ray segments to specific layers of a texture array. (b) Subsequently, the layers are blended in an image-based compositing pass. (c) This results in the right view. [9, Fig. 1]*

This section includes content that has also been published in Buchacher and Erdt [9], which has been compiled in unison with this thesis.

4.1 Proposed Method Using Re-Projection Layers

The proposed method is split into two phases, as illustrated in Figure 6. First, the left view is rendered using regular ray casting during which accumulated colors of ray segments are re-projected and stored in layers of a texture array buffer. Second, the textures are blended in the correct per-pixel order such that the right

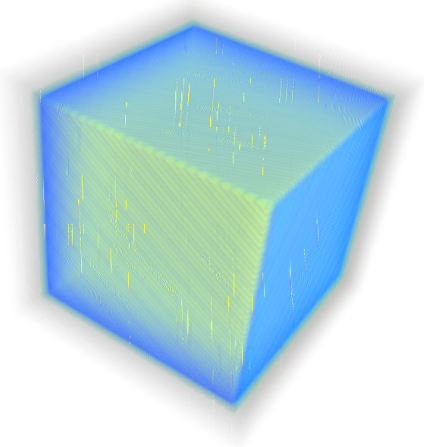


Figure 7: *Fragment order related artifacts for GPGPU ray caster using the fast stereo method by He and Kaufman [23].*

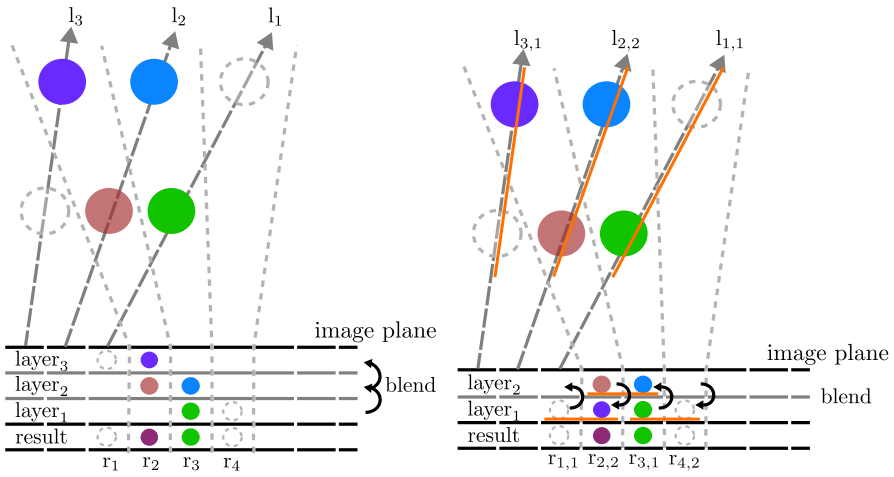
view is composed. The approach builds upon the fast stereo method outlined in Section 2.3. Following the previous approach, ray casting is only executed for the left view and the right view is essentially produced by means of ray segment re-projection. However, a GPU adaptation of the method is not trivial, as the parallelization of ray traversal is opposed to the requirement of sequential processing of rays.

Attempting to control the order of fragment shader instances using the vertex shader or altering the shape of the blocks to resemble scan-lines using a GPGPU ray caster is not universally applicable. One such attempt is to reduce the rasterizer's influence on the fragment order by arranging vertices in scan-line order in the vertex buffer and rendering each as a point. Another idea is to use a GPGPU implementation of a ray caster and to invoke scan-lines of pixels as local work groups. In both cases some hardware might in fact process the rays in scan-line order, whereas other processes multiple scan-lines or multiple pixels in parallel. As a result flickering artifacts occur, as the order of writes to the right image pixels is arbitrary (cf. Figure 7).

The solution presented here lies within decoupling the compositing from the acquisition of segment colors by introducting a texture array buffer. Instead of reading, compositing and writing the segment colors to the right image directly, each scan-line of rays is assigned a layer of a texture array instead. These layers are referred to as *re-projection layers*. Following the ray casting phase, the layers are composited to obtain the final right view's image. Thus, a pixel's final color is now independent of the order in which horizontally adjacent rays are processed. Instead it depends on the order in which the layers are blended which is fixed. The re-projection layers can be filled in parallel.

The approach is illustrated in Figure 8a. Based on the horizontal image coor-

dinate, each ray is assigned a layer of the texture array. The re-projection to the right image coordinates is performed as before. The segment color is written to the corresponding position in the layer texture. When all rays have been processed the final image is composited by blending the layers.



(a) *Full Array Length [9, Fig. 4]* **(b)** *Reduced Array Length [9, Fig. 5]*

Figure 8: *Re-projection of samples along rays to layers of a texture array.* layer_k*: texture array layer. The blend order is front-to-back.*

The naive implementation would require as many textures as the image is wide in pixels, i.e. rays. To reduce the number of layer textures it is possible to share them between multiple rays.

The key observation is that each ray has a start and end point and thus only writes to a bounded horizontal segment of the texture. The space outside of this segment could be used by another ray which writes the corresponding image coordinates, but initially on a different layer. In turn, this ray also only writes to a bounded segment of the texture. Following this pattern, the same texture could be assigned to rays in equal regular steps. The required step size can be estimated using the epipolar geometric properties between the left and right view and the rays.

When re-using layer textures for multiple rays the compositing phase must to be modified. Before, a layer's index indicated its spatial relationship to the other layers and thus the blend order. The fully transparent areas in each texture simply corresponded with the empty space that is not covered by the rays. Now, for each pixel the index of the texture must be identified in which the color of the outmost sample can be found. The problem originates from the re-usage of textures for multiple rays. Texture space that would otherwise be free is now filled with colors from a ray that is actually further in the back regarding the re-projection. The correct entry layer to the compositing loop is the one that potentially yields the front-most re-projected color-sample. Details on a possible implementation are

given in Section 4.3.

The described structure is illustrated in Figure 8b. The segment each ray traverses is indicated in orange. In this example each segment is approximately two pixels wide. The second index of each left view's ray $l_{i,k}$ identifies the layer it writes to. The second index of each right view's pixel $r_{j,k}$ identifies the entry layer at which the front-most color is to be found. From this layer, as many layers are blended as the epipolar line segments are wide. A modulo operation using the number of layers is applied to the current index each time it is incremented.

4.2 Estimating Texture Array Buffer Size

Depending on which parameters can be set to a constant, the suitable number of layers or rendering parameters may be calculated. For example, using one of the current consumer VR headsets, such as the HTC Vive, some of the properties can be fixed. It has a display resolution with a width of $w = 1080$ pixels per eye, a recommended field of view of $2\alpha = 110$ degrees and a default eye distance of $e = 0.065$ metres.

It is possible to estimate the minimal number of textures that still ensures that no two rays will interfere with each other. This is achieved by estimating the maximal pixel range d^{p} that any ray could write to. For quick reference, the following equations will be used in the subsequent descriptions. They comprise the pixel space transformation (Eq. 17, 18), finding the maximal segment lengths for infinite ray length (Eq. 19), for finite ray length (Eq. 20) and for known bounding sphere radius (Eq. 21) and finding the minimal focal length given the other parameters (Eq. 22).

$$s = \frac{w}{2h\tan\alpha} \tag{17}$$

$$d^{\mathrm{p}} = ds \tag{18}$$

$$d_{\infty} = e \tag{19}$$

$$d_z = e - \frac{eh}{z} \tag{20}$$

$$d_r = e - \frac{eh}{h+2r} \tag{21}$$

$$h = -r + \sqrt{r^2 + \frac{edw}{2d^{\mathrm{p}}\tan\alpha}} \tag{22}$$

It is sufficient to first estimate the maximal epipolar line segment length d. The subsequent transformation to pixel space is achieved by a scalation by s using the constants, following equations 17 and 18.

The optimal, i.e minimal, value of d^{p} can be approximated through different approaches. One approach is to assume that any given ray ranges from the left camera's near plane all the way to infinity. Given that both view directions are parallel, any point at the horizon is re-projected to the same position as in the left

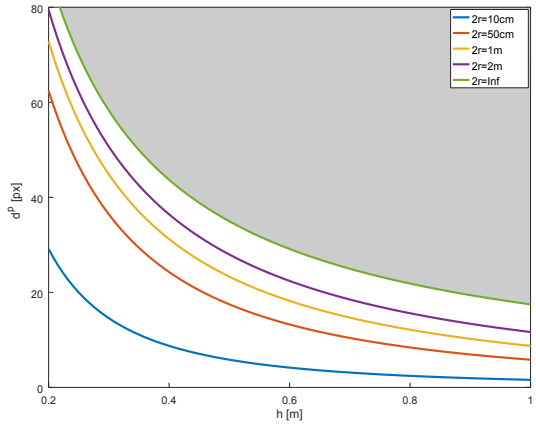


Figure 9: *Pixel range d^{p} plotted against focal length h for different object sizes $2r$ (colored lines). d^{p} calculated with $e = 0.065m$, $2\alpha = 110°$, $w = 768px$. [9, Fig. 6]*

image. A point on the near plane will be re-projected to a point which lies one eye distance e away on the image plane. Thus, the segment on the epipolar line from start to end will have length $d_\infty = e$ (cf. Equation 19).

The distance to the far plane or outmost exit point of all rays can also be taken into account. Equation 20 holds true for a ray that begins at the near plane at distance h and ends at distance z, resulting in the estimate d_z. This is suitable for the case in which a volume might enclose the entire view frustum, for example a volumetric terrain.

Considering the case in which the inspected volume has a fixed or maximal size, the range can be computed by finding the longest possible ray that enters the volume at the image plane. Commonly, volumes are represented by a bounding box, thus the longest distance between two points on the bounding box is defined by its diagonal with length $2r$. The end point of the longest possible ray then lies at the distance $z = h + 2r$, which corresponds to the ray that is emitted through the center of the view. Thus, the estimate d_r is found following Equation 21.

In this case the re-projected segment on the epipolar line changes with the distance to the volume. Refering to Figure 5, imagine the volume stretches between P' and P. As the entry and exit points move away from the near plane the re-projected points P'_r and P_r drift to the right while d becomes shorter. As stated above the entry layer to each right view's pixel needs to be identified to ensure the correct blend order. The corresponding left view's ray lies at an offset of $\frac{eh}{z'} - e$ to P'_r. The index is determined by calculating the corresponding left view's image coordinate. Furthermore, the compositing loop can be stopped after fewer iterations as d is also shorter.

Lastly, the minimal focal length h, given all the other parameters including the maximal pixel distance d^{p} can be found following Equation 22. The plot in Figure 9 exemplifies how the focal length maps to the number of layers at different object

sizes. The upper bound is given by an object infinite in size, which relates to the case of a ray cast to infinity.

4.3 Implementation

In this section, notable details to a possible OpenGL implementation are highlighted. First of all, the OpenGL implementation of the technique requires functionality of at least version 4.3. Most notably, the *image load store* functionality [7], for arbitrary access to texture memory. The texture memory can be initialized as a floating point texture array, with the user defining the texture resolution and number of layers. As a consequence, the near plane distance h should be calculated following Equation 22. One straight-forward way to assign fragments of the ray caster to a texture layer index is to use a modulo operation on the x-coordinate. The shader function listed in Algorithm 1 assigns indices incrementally from right to left.

Algorithm 1 Assignment of Fragment $\vec{x}$ to Layer Index k

procedure GETLAYERIDX($\vec{x}$)
    $x \leftarrow \text{getFragCoordX}(\vec{x})$
    $k \leftarrow d^{\text{p}} - (x \bmod d^{\text{p}}) - 1$ ▷ d^{p} : total number of layers
    return k
end procedure

Algorithm 2 Ray Casting with Ray Segment Re-Projection

procedure RAYCASTANDREPROJECT($\vec{x}$)
    $k \leftarrow \text{getLayerIdx}(\vec{x})$
    $x_r \leftarrow \text{reprojectCoords}(\vec{x}(t_{\text{start}}))$
    $S' \leftarrow \vec{0}$ ▷ S'_r: composite segment color
    for $t \in [t_{\text{start}}, t_{\text{end}}]$ **do**
        $x'_r \leftarrow \text{reprojectCoords}(\vec{x}(t))$
        if $x_r \neq x'_r$ **then**
            for $x \in [x_r \,..\, x'_r - 1]$ **do**
                $L_{k,x} \leftarrow S'$ ▷ $L_{k,x}$: color at position x in layer k
            end for
            $S' \leftarrow \vec{0}$ ▷ reset segment color
        end if
        $C' \leftarrow \text{blend}(C', C_i)$
        $S' \leftarrow \text{blend}(S', C_i)$
        $x_r \leftarrow x'_r$
    end for
    $L_{k,x_r} \leftarrow S'$
end procedure

Algorithm 2 depicts the extended ray casting procedure to be implemented in a fragment shader. First the target texture index for the fragment is determined, then the running variables for the segment color and texture coordinates are initialized (lines 2 to 4). During the ray traversal loop (lines 5 to 16), the current position is re-projected to the right view and the pixel coordinate is updated (Line 6). If a change is detected, the color of the traversed ray segment is stored at the covered texture positions and the segment color is reset (lines 7 to 12). At the same time, the standard ray casting procedure accumulates the fragment color (Line 13) and the segment color (Line 14). Lines 9 and 17 are implemented using arbitrary memory access functionality, such as the OpenGL *imageStore* function, given the texture array is available as an *image2DArray* uniform variable.

```
1  // view space (v) start (n) and end (f) points
2  // for left (l) and right (r) view
3  vec4 vn_l = vec4(0.0, 0.0, zStart, 1.0);
4  vec4 vf_l = vec4(0.0, 0.0, zEnd, 1.0);
5  vec4 vn_r = viewMatrix_r * inverse(viewMatrix_l) * vn_l;
6  vec4 vf_r = viewMatrix_r * inverse(viewMatrix_l) * vf_l;
7
8  // transform to respective projective space (p)
9  vec4 pn_l = perspectiveMatrix_l * vn_l;
10 vec4 pn_r = perspectiveMatrix_r * vn_r;
11 pn_l /= pn_l.w;
12 pn_r /= pn_r.w;
13 vec4 pf_l = perspectiveMatrix_l * vf_l;
14 vec4 pf_r = perspectiveMatrix_r * vf_r;
15 pf_l /= pf_l.w;
16 pf_r /= pf_r.w;
17
18 // transform to respective pixel space (s)
19 vec4 resolution = vec4(fWidth, fHeight, 1.0, 1.0);
20 vec4 sn_l = ((pn_l * 0.5) + 0.5) * resolution;
21 vec4 sn_r = ((pn_r * 0.5) + 0.5) * resolution;
22 vec4 sf_l = ((pf_l * 0.5) + 0.5) * resolution;
23 vec4 sf_r = ((pf_r * 0.5) + 0.5) * resolution;
24
25 // pixel coordinates offset (d)
26 vec4 sdn = (sn_r - sn_l);
27 vec4 sdf = (sf_r - sf_l);
28 float pixelOffsetNear = abs(sdn.x);
29 float pixelOffsetFar  = abs(sdf.x);
```

Source 1: *Offset Calculation for Entry Layer Determination*

Concerning the compositing phase, the entry layer to each right view's pixel needs to be identified when working with Equation 21. That is, ray segments are considered to be confined to the volume's bounding sphere and the volume's distance may change dynamically. Working with current generation HMDs it may be the case that exposed projection matrices are not symmetrical for the stereoscopic views. Thus, essential assumptions about the two pixel coordinate systems might not hold. To preclude possible problems space transformations can simply be made

using the exposed matrices instead. Source 1 shows the offset calculation using the HMD matrices. The values for *zStart* and *zEnd* are calculated from the current distance to the front and back side of the bounding sphere of the volume. The resulting values *pixelOffsetNear* (d^{n}) and *pixelOffsetFar* (d^{f}) are used to calculate the pixel position of the left view's ray that intersected the front-most position of a right view's ray. The offsets need to be computed only once whenever the view changes and can be provided to the compositing shader as a uniform variable.

Finally, the compositing shader is a simple blend shader that composes the reprojection layers in a front-to-back manner. Algorithm 3 depicts the compositing procedure to be implemented as a fragment shader. The layer index is retrieved, that corresponds to the calculated left view's pixel (lines 3 and 4). This determines the initial layer index, potentially containing the first color along the virtual right view's ray. In Line 5, the number of consecutive layers to blend is calculated.

Algorithm 3 Second View Compositing of Fragment $\vec{x}$

```
1:  procedure ComposeView(d^n, d^f, x⃗)
2:      x ← getFragCoordX(x⃗)
3:      x_l ← x + d^n
4:      k ← getLayerIdx(x⃗_l)                    ▷ x⃗_l: left image pixel
5:      n ← d^n − d^f
6:      for i ∈ [0..n−1] do
7:          j ← (k + i) mod d^p                 ▷ d^p : total number of layers
8:          C' ← blend(C', L_{j,x})             ▷ L_{j,x}: pixel color in layer j
9:      end for
10: end procedure
```

5 System Design for Virtual Reality Volume Rendering

5.1 Concept

The concept of the rendering system picks up on the idea of an asynchronous data flow and is inspired by the approach by van Waveren [63]. The rendering data flow is divided into two swap-chains that run concurrently to each other (cf. Figure 10). Using framebuffer objects (FBOs), the ray casting swap-chain consists of two FBOs that resemble the common front and back buffers. The back buffer represents the current rendering target to which the ray caster component writes its results. Meanwhile, the front buffer represents the currently presented image. However, instead of presenting the image to the user by submitting it to the display, it is used as the current source for the image warper component. The display swap-chain then resembles the common double buffer of the display. The front buffer is the image that the user currently sees, while the back buffer is the render target to which the image warper writes its result for the next frame.

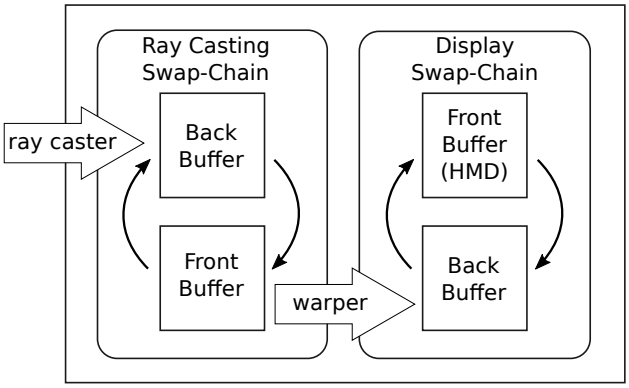


Figure 10: *Asynchronous Data Flow*

Asynchronicity in the system occurs when the ray caster component and the image warper component operate at different frequencies. As stated by van Waveren [63], this is desirable, as the display swap-chain can reliably be synchronized with the display refresh rate. With this, the perceived frame rate can be increased and inconsistent frame rates can be smoothed out by appropriately warping the last results using the latest head tracking information. As the motion-to-photons latency should be as low as possible, the image warper component needs to be able to execute within milliseconds before the display refreshes. The performance requirements for the ray casting component can thus be relaxed. Instead of aggressively trading image quality for performance improvements, more practical acceleration techniques can be employed. Either way, new images should be produced at least at interactive rates, as image warping should only be used to compensate for missing frames.

5.2 Asynchronous Rendering

Using OpenGL on systems that are equipped with a single GPU, special considerations have to be made to enable an asynchronous data flow. Even though vertex and fragment processing is generally executed in parallel, draw calls are processed sequentially. Additionally, there is no direct way to pause the current execution of one task at a particular time to execute another. The task that poses the problem here is ray casting a view in one draw call. It takes presumedly longer than one display refresh cycle to finish, thus the instant of time at which image warping should be executed is potentially missed.

To overcome this problem, the ray casting task is split into multiple sub-tasks of managable execution times that can be distributed across multiple frames. Another challenge arises from the fact that a running task cannot be stopped at a particular time, but instead only after execution of all queued rendering tasks has finished. As much time of a frame as possible should be spent on ray casting and only as little time as necessary on image warping, as close to the display refresh as possible.

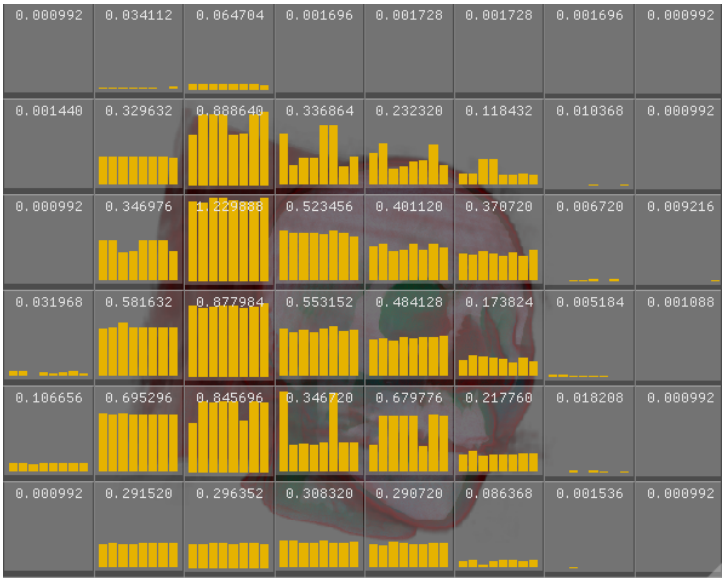


Figure 11: *Splitting the viewport into uniform grid of cells and tracing the render time for each cell over time. The portrayed numbers indicate the last measured render time in milliseconds.*

The solution employed by the proposed system is to split the ray casting viewport into a grid of uniformly sized cells and to render only a subset of cells per frame, always leaving sufficient time for image warping at the end. As the execution time for a cell is largely incomputable beforehand, it is traced over time (cf. Figure 11) and predicted conservatively for the next frame. With this design choice, the ray caster adapts dynamically to the computational workload.

Algorithm 4 depicts a simplified progress of a frame's rendering iteration. It is initialized with the frame time T_f, the time reserved for warping T_w, the last drawn cell index i', and the total number of cells n. As many cells are drawn as the remaining render time T allows, however at least one cell is rendered each frame (lines 4 to 12). If the draw call for the last cell has been comitted, the ray casting front and back buffers are swapped and rendering for this frame is stopped (lines 8 to 11). Either way, the last drawn index is saved for the next frame (Line 13).

The prediction of the render time of a cell is largely arbitrary. For example, the render time recorded from the last frame could simply be reused, scaled, or extrapolated from the last couple of frames. The implemented algorithm (cf. Algorithm 5) predicts with regards to the cell's neighborhood using the render times from the last frame. The prediction is the result of a relaxed maximum filter and average. This heuristical approach aims at compensating for head movement between consecutive frames. Render times are slightly overestimated (Line 6) to prevent missing the critical time for image warping.

Achieving asynchronicity in this way comes at the price of inherently reducing the ray casting update rate. As Figure 12 illustrates, the rendering task (R) is split

Algorithm 4 Chunked Adaptive Rendering

```
1:  procedure RENDERITERATION(T_w, T_f, i', n)
2:      T ← T_f − T_w
3:      i ← i' mod n
4:      do
5:          T ← T − predictTimeForCell(i)
6:          renderCell(i)                          ▷ ray cast pixels in cell
7:          i ← (i+1)
8:          if i = n then
9:              swapBuffers()                      ▷ ray casting swap-chain
10:             break
11:         end if
12:     while T > predictTimeForCell(i)
13:     i' ← i
14: end procedure
```

Algorithm 5 Cell Render Time Prediction

```
1:  procedure PREDICTTIMEFORCELL(i)
2:      T̄ ← 0
3:      for j ∈ 𝒩(i) do          ▷ 𝒩(i): indices of cells in neighborhood
4:          T̄ ← T̄ + max(T_i, T_j)          ▷ T_{i/j}: cell render time from last frame
5:      end for
6:      T̄ ← bT̄/|𝒩(i)|                          ▷ b: bias factor ≥ 1
7:      return T̄
8:  end procedure
```

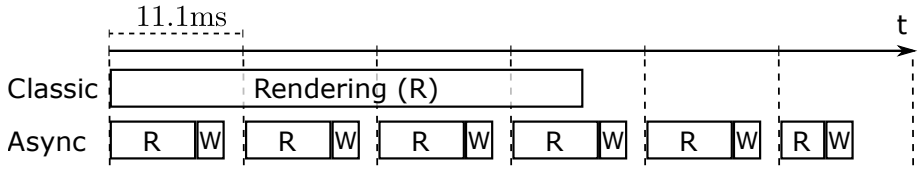


Figure 12: *Timeline overview of classical rendering versus asynchronous rendering. Dashed vertical lines indicate display updates of the 90Hz display. After a task finishes, the results are submitted to the display on the next update.*

in multiple frames. Even though the total time spend with ray casting is identical to the classical approach, the time until the frame is finished is longer. This is due to the additional time reserved for warping (W) and idle time until the display updates which results from conservative prediction. Ideally, both are as small as possible, allowing to spend as much time with ray casting as possible.

5.3 Programmable Display Layer

The *programmable display layer* (PDL) is a component of the architecture described by Smit et al. [58, 57]. It is an image warping architecture specifically designed for VR applications and is very similar to the system described here. Essentially, the PDL is the image warper component that updates the view at the refresh rate of the display. The display layer program is customizable and in this section, multiple warping techniques are explored.

The first warping technique, presented in Section 5.3.1, represents the most basic approach to achieve very low-latency head tracking. While this technique compensates nearly entirely for rotation, it undercompensates for translation which is especially noticable in the near field. Hence, Section 5.3.2 describes an experimental technique inspired by depth-image-based warping that addresses this problem. Finally, Section 5.3.3 describes a state-of-the-art volumetric warping technique and performance critical alterations.

5.3.1 Simple Image Warping

This method is based on the algorithm proposed by Peek et al. [50]. It aims at providing a practical and fast warping algorithm that efficiently smooths (rotational) head tracking on HMDs. Essentially, the image data of the last ray casting result is mapped as a texture to a quad that is aligned with the location of the far clipping plane of the corresponding view frustum. This quad is then rendered with the updated view transformation that holds the most recent head tracking information. In an image-based implementation, the image texture is sampled by re-projecting the fragment positions from the far plane to the image space of the old view (cf. Source 2 & 3).

```
1 in  vec2 pos; // quad vertex positions from -1 to 1
2 out vec4 passPos;
3 void main()
4 {
5     // set z,w to 1 to project to the far plane
6     passPos = vec4(pos, 1, 1);
7     gl_Position = passPos;
8 }
9
```

Source 2: *Simple Warp Vertex Shader*

```
uniform mat4 uProjection;
uniform mat4 uViewOld;
uniform mat4 uViewNew;

uniform sampler2D color_map;

in vec4 passPos;
out vec4 fragColor;

void main()
{
    // unproject fragment position, then reproject to old view
    vec4 viewPos = inverse(uProjection) * passPos;
    viewPos /= viewPos.w;
    vec4 projPos  = uProjection * uViewOld * inverse(uViewNew)
    * viewPos;
        projPos /= projPos.w;
    vec2  texPos  = projPos.xy * 0.5 + 0.5;

    fragColor = texture(color_map, texPos);
}

```

Source 3: *Simple Warp Fragment Shader*

5.3.2 Experimental First-Hit Map Mesh-Warping

The method is closely related to the works presented by Mark et al. [43] and Shade et al. [56]. As a by-product of the computations for the occlusion frustums (cf. Section 3.2), the so called first hit map is produced. For each ray it contains the first depth at which a semi-transparent voxel is hit. For a binary transfer function that produces only fully transparent and fully opaque voxels this essentially relates to a common depth buffer. By projecting a polygonal mesh through the original viewport using this depth and color information the scene can be approximated and then efficiently be rendered from another view point.

There are apparent drawbacks to this simple view synthesis method that will be discussed later. However, it is very efficient, compensates for rotation and presumably improves on the perceived effect of translational stuttering. Thus, it has been implemented as an experimental technique. In fact, the method can emulate the method presented in Section 5.3.1 by using a cleared depth map and the last ray casting result.

```
uniform mat4 uProjection;
uniform mat4 uViewOld;
uniform mat4 uViewNew;

uniform sampler2D depth_map;

in  vec2 uv; //grid vertex uv coordinates from 0 to 1
out vec2 passUV;

void main() {
        passUV = uv;
    float depth = texture(depth_map, uv).x;

    // unproject from depth map, then reproject to new view
    vec4  projPos  = vec4(uv, depth, 1.0) * 2.0 - 1.0;
    vec4  viewPos  = inverse(uProjection) * projPos;
          viewPos /= viewPos.w;
    vec4 worldPos  = inverse(uViewOld) * viewPos;
    gl_Position    = uProjection * uViewNew * worldPos;

    // clamp to far plane
    if (gl_Position.z < -gl_Position.w){ gl_Position.z = -
    gl_Position.w; }
}

```

Source 4: *Grid Warp Vertex Shader*

There are problems to be expected with this method that will be especially apparent for volume rendering. The per-pixel color that results from solving the volume rendering integral is only valid for the original ray direction. Mapping the colors as a texture to a mesh essentially paints this volumetric, directional information onto a surface. The surface color and the volumetric information coincide for every pixel only from the original view. Using the first hit map, all information that lies behind the first non-transparent voxel will be painted to the surface. This is especially noticable in missing or unexpected parallax effects.

5.3.3 Novel-View Synthesis for Volume Rendering

The novel-view synthesis algorithm presented by Lochmann et al. [40] was specifically designed for volume rendering. The paper pre-eminently adresses client-server systems in which the server-side handles volume rendering and the client-side generates intermediate images using the last result. It is assumed that in any case the server takes orders of magnitues longer to render the volume compared to the client-side generating novel views. The server compresses the view into a layered piecewise-analytic emission-absorption representation. The client receives the result textures and synthesizes a new image by efficiently ray casting this structure. Typically, a low and fixed number of layers suffices to produce novel views in the matter of milliseconds. The compressed representation consists of depth textures

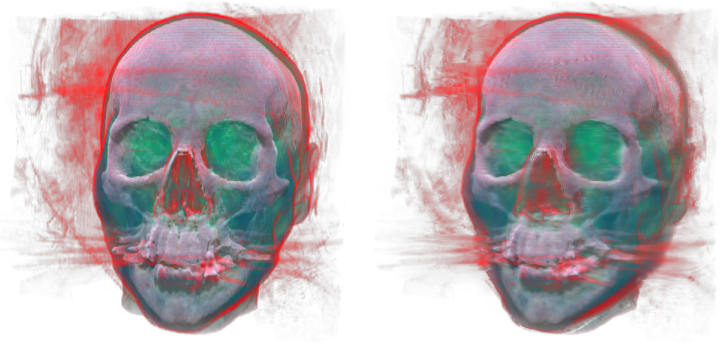
that mark the start and end points of the layers and RGBA-textures. For the latter, emission coefficients are written to the RGB-channels and absorption coefficients are written to the α-channel.

For the source data generation, also called *original view synthesis*, each ray is split into a small number of segments during the ray traversal. Utilizing the *Beer-Lambert* law, emission and absorption coefficients are calculated analytically from the segment length and the accumulated color and opacity values, by inversion of Equation 3. Essentially, the ray segment is compressed to these coefficients as if it ran through a matter of uniform, homogeneous color and density. For each layer, the coefficients are packed into one target while the layer depth is stored in another, utilizing multiple render target (MRT) functionality. Accumulated color and opacity are reset for each ray segment. When a view has finished to be encoded, all textures and the original view matrix are sent to the client to be used for novel-view synthesis. To generate a novel view, the client re-projects its hypothetical rays to the source's pixel space. Each ray is traversed by intersecting it with pixel borders and layers. For each segment, the layer's corresponding emission and absorption values are read. With emission, absorption and segment length, color and opacity values can be calculated. In turn, these can be accumulated in the common ray casting manner, before the ray traversal is advanced. The straight forward conversion between the respective values can be implemented as depicted in Algorithm 6.

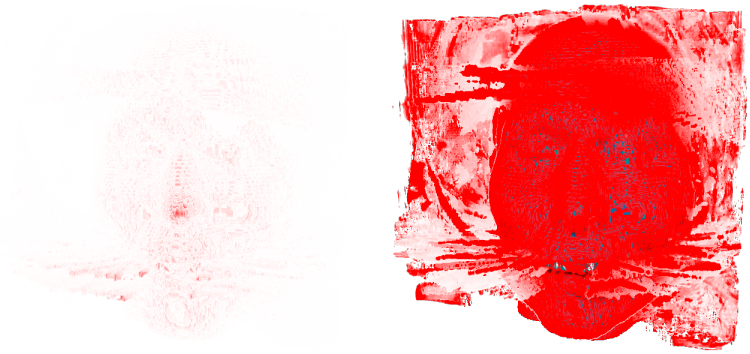
Algorithm 6 Beer-Lambert Conversions

```
1:  procedure GETEMISSIONABSORPTION( C, C_α, d)
2:      κ ← −log(1 − C_α)/d
3:      c ← C/(1 − κ)
4:      return c, κ
5:  end procedure
6:  procedure GETCOLOR( c, κ, d )
7:      T ← e^(−κd)
8:      C ← cT
9:      C_α ← 1 − T
10:     return C, C_α
11: end procedure
```

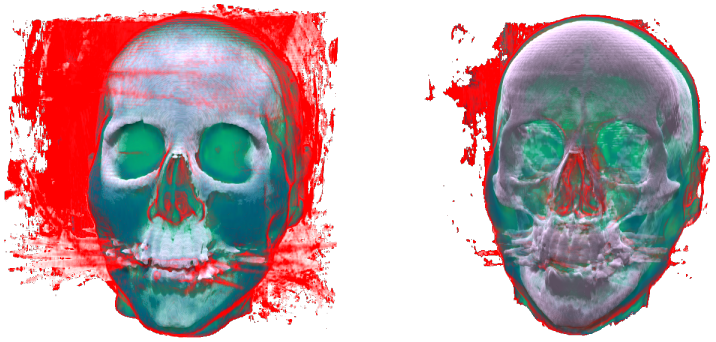
For the sake of real-time feasibility, some alterations were made to the original algorithm. For the original view synthesis, some simplifications were applied to the layering process (cf. Algorithm 7). In the original paper, layer borders are produced at regular intervals of the normalized transmittance which is calculated as the transmittance (i.e. $1-\alpha$) along the ray divided by the transmittance of a point at infinity. However, this implies that, before the thresholds can be determined, the final transmittance must be computed which requires to march the entire ray once to accumulate it. The implemented algorithm splits rays at user-defined thresholds $y_1, \ldots, y_k$ which requires only one pass along the ray instead of two.



(a) *Original View* **(b)** *Novel View*



(c) *Layer 1* **(d)** *Layer 2*



(e) *Layer 3* **(f)** *Layer 4*

Figure 13: *Example emission-absorption results for original view synthesis using* $k = 4$ *layers and opacity thresholds* 0.001, 0.05, 0.5 *and* 1.0. *Textures hold unbounded floating point emission and absorption values. Figure (b) shows a generated image from a novel view offset by* 0.2m *and* 32 *steps.*

For the novel view synthesis, some simplifications were applied to the ray traversal. The paper suggests using a modified 3D-DDA algorithm to traverse and

Algorithm 7 Original View Synthesis

1: **procedure** RAYCASTEMISSIONABSORPTIONLAYERS($y_1, \ldots, y_k, \vec{x}$)
2: $d_0 \leftarrow t_{\text{start}}$
3: $j \leftarrow 1$ ▷ j: current layer
4: $S' \leftarrow \vec{0}$ ▷ S': composite segment color
5: **for** $t \in [t_{\text{start}}, t_{\text{end}}]$ **do**
6: $C' \leftarrow \text{blend}(C', C_i)$
7: $S' \leftarrow \text{blend}(S', C_i)$
8: **if** $C'_\alpha \geq y_j$ **then**
9: $d_j \leftarrow t$
10: $d \leftarrow d_j - d_{j-1}$
11: $\bar{c}_j, \bar{\kappa}_j \leftarrow \text{getEmissionAbsorption}(S', S'_\alpha, d)$ ▷ write to MRT
12: $j \leftarrow j + 1$
13: $S' \leftarrow \vec{0}$ ▷ reset segment color
14: **end if**
15: **end for**
16: **end procedure**

sample the slice textures. This is very precise, but with the degree of novelty of the viewing angle in relation to the original view, the number of intersected pixels quickly increases. The original paper states computation times for a novel view at a 1024×1024 resolution that range between 21.8ms ($k = 2$, 5° offset) and 55.4ms ($k = 8$, 15° offset). As the target time window for warping is only a few milliseconds, a different sampling method is used. A quasi-uniform sampling method is implemented instead (cf. Algorithm 8). The novel ray is traversed in a fixed amount of steps n (lines 5 to 23). The methods *getTextureCoord* and *getDistance* return values with respect to the original view. At each step the ray segment is intersected with the layer information from the current texture position (lines 9 to 14). Colors and opacities are recovered using the resulting distances and emission-absorption coefficients (lines 15 to 20). This way, the original view is accurately reconstructed and the computational complexity can be controlled. As a trade-off, reconstruction quality is reduced for increasing degree of novelty. Figure 13 shows an exemplary result for the emission-absorption layers and the novel view synthesis using the alterations stated above.

6 Evaluation

6.1 Single-Pass Stereo Ray Casting Experiments

The performance experiments were conducted on a Windows 10 PC equipped with a Nvidia GTX 1080 graphics card, an Intel Core i7-6700K CPU and 64 GB of RAM. A variety of CT, MRI, DTI and synthetic sample data sets of varying resolutions and density properties were used for the experiments. An overview is given

Algorithm 8 Novel View Synthesis

procedure NOVELVIEWSYNTHESIS($n, \vec{x}$)
  $\vec{u} \leftarrow \text{getTextureCoord}(\vec{X}_0)$
  $j, m \leftarrow 0$ ▷ j: current layer, m: previous layer
  $d'_1 \leftarrow d_m(\vec{u})$ ▷ read texture d_m at position $\vec{u}$
  for $i \in [1 \,..\, n]$ **do**
    $j, m \leftarrow 0$
    $d'_2 \leftarrow \text{getDistance}(\vec{X}_i)$
    $\vec{u} \leftarrow \text{getTextureCoord}(\vec{X}_i)$
    while $d'_2 \geq d_j(\vec{u})$ **do** ▷ find layer of current sample
      $j \leftarrow j + 1$
    end while
    while $d'_1 \geq d_m(\vec{u}) \wedge m \leq j$ **do** ▷ find layer of previous sample
      $m \leftarrow m + 1$
    end while
    while $m < j \wedge d'_1 < d_j(\vec{u})$ **do** ▷ intersect and composite layers
      $S_i \leftarrow \text{getColor}(\bar{c_m}(\vec{u}), \bar{\kappa}_m(\vec{u}), d_m(\vec{u}) - d'_1)$
      $C' \leftarrow \text{blend}(C', S_i)$
      $d'_1 \leftarrow d_m(\vec{u})$
      $m \leftarrow m + 1$
    end while
    $S_i \leftarrow \text{getColor}(\bar{c_j}(\vec{u}), \bar{\kappa}_j(\vec{u}), d'_2 - d'_1)$
    $C' \leftarrow \text{blend}(C', S_i)$
  end for
end procedure

in Table 2. A volume is loaded into a single-component floating point 3D texture.

The ray caster is implemented as a fragment-shader. Each ray samples the 3D texture at a step size of $\frac{1}{2}v$ between the front and back faces of the volume's bounding box where v is defined as the minimal voxel extent in texture space. The sample color is retrieved using a 1D transfer function texture lookup. Early ray termination is implemented with an opacity-threshold of 0.99. The volume is scaled such that $r = 1$ metre. The focal length h is calculated according to Equation 22.

The *structural dissimilarty measure* (DSSIM) is used to assess the image quality of the result image. it is a measure derived from the *structural similarity measure* (SSIM) first introduced by Wang et al. [68]. Luminance, structure and contrast differences between the images influence its value. It ranges from 0 to 1 where a value of 0 indicates equality. For the experiments, standard SSIM stabilization parameters and a window size of 8×8 were used. The definition for speed-up (time saving) is adopted from He and Kaufman [23]:

$$V = 1 - \frac{T_{l+r} - T_l}{T_r}. \tag{23}$$

Volume	Size	Voxels
Solid Box	$64 \times 64 \times 64$	$262,144$
DTI	$128 \times 128 \times 58$	$950,272$
CT Head	$256 \times 256 \times 113$	$7,405,568$
Visibile Male	$128 \times 256 \times 256$	$8,388,608$
Engine	$256 \times 256 \times 256$	$16,777,216$
Sheep Heart	$352 \times 352 \times 256$	$31,719,424$
Piggy Bank	$512 \times 512 \times 134$	$35,127,296$
Bonsai	$512 \times 512 \times 154$	$40,370,176$

Table 2: *Overview of volume data sets used for experiments.*

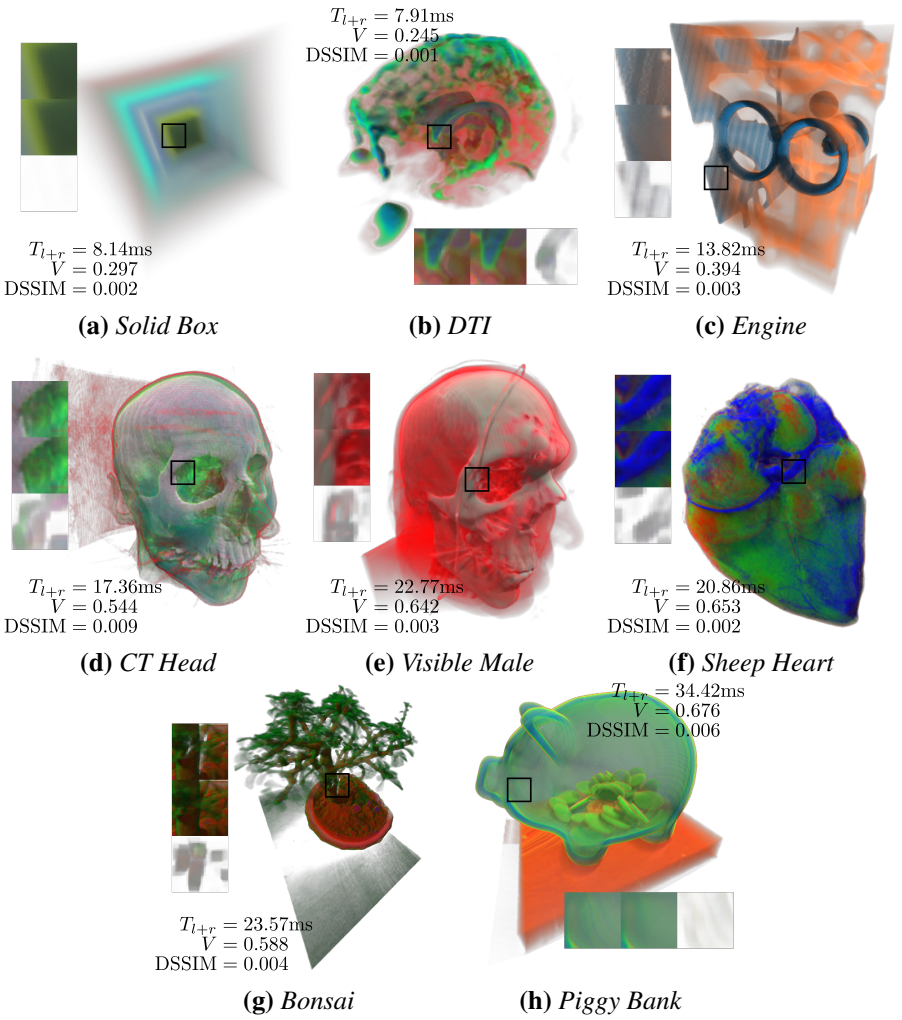
Here, T_{l+r} is the total time for rendering of a stereo pair in one pass, T_l is the time for rendering the left image, and T_r is the time for rendering the right image. Thus, the definition reflects that the technique does not produce a speed-up for rendering the left image, but only that time can be saved on generating the right image. The stated values for DSSIM, T_{l+r} and V are the averages over a 50-step 360° rotation of the volume around the y-axis.

6.1.1 Qualitative Results

For a qualitative evaluation Figure 14 gives an overview of generated views of the sample volumes. For each ray sample a shadowing effect is applied by accumulating the opacities of 24 samples in direction of a parallel light source and multiplying the result with the sample's color (cf. Section 2.2.4). Additionally, an ambient occlusion effect is applied by averaging the opacitites of 14 samples on a sphere around the sample and multiplying the result with the sample's color.

DSSIM values are generally low, but not zero. Part of the error is due the fact that distances between samples on the left rays are slightly different from the right. Therefore the emission and absorption integral is slightly different aswell. Similar to the approach in [40] the error could potentially be reduced by encoding emission-absorption-coefficients in each layer and calculating the distance between layers during the compositing phase. Another approach would be adapting the linearly-interpolated re-projection scheme [23]. Note that neither approach would be able to achieve full equality.

Due to early-ray termination of left view's rays, some information might be missing near high opacity structures. For example, missing samples become evident near the bonsai tree's trunk in Figure 14g. However, more often not all rays that run across the same image region are terminated early. Thus, the effect is not as drastic as in common LDI rendering where hole-filling strategies [59] become necessary. Also note that in the experimental VR setup with a high field of view the volumes generally did not cover the entire viewport, arguably affecting the DSSIM values in a favorable way.



(a) *Solid Box* **(b)** *DTI* **(c)** *Engine*

(d) *CT Head* **(e)** *Visible Male* **(f)** *Sheep Heart*

(g) *Bonsai* **(h)** *Piggy Bank*

Figure 14: *Experimental results for different volumes. Detail views from top to bottom (resp. left to right): Result, reference, per-channel DSSIM image (on white background). Rendered at resolution* 768^2*, 32 layers,* $e = 0.065$m, $2\alpha = 110°$, $h = 0.435$m.

6.1.2 Quantitative Results

Figure 15 illustrates the arbitrary shading method used to parametrize shading complexity on a continuum. The method is inspired by Monte Carlo Volume Rendering [14] in that a sample's color results from additional samples taken from random directions around it. Each direction is traced for up to 4 steps to estimate the amount of light coming through from that direction. Increasing the amount of directions and number of additional samples increases image quality and reduces noise. Thus, the number of additional texture reads is used as an indicator for shading complexity in general.

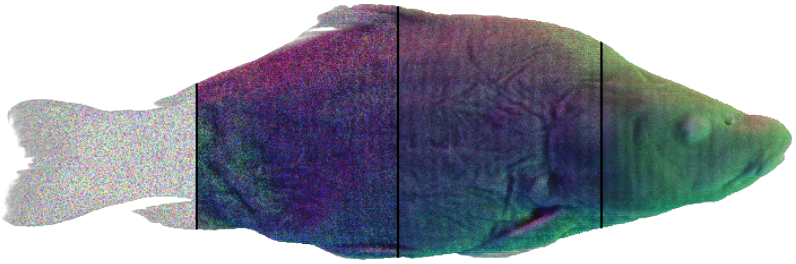


Figure 15: *Shading method using random directional vectors as directional light sources. The (x,y,z) coordinates of a vector are used as the (r,g,b) light intensities coming from that direction. Additional samples along a direction accumulate the occlusion towards that light source. Left to right: 1 direction totaling 0 samples, 1 direction totaling 4 samples, 4 directions totaling 16 samples, 16 directions totaling 64 samples.*

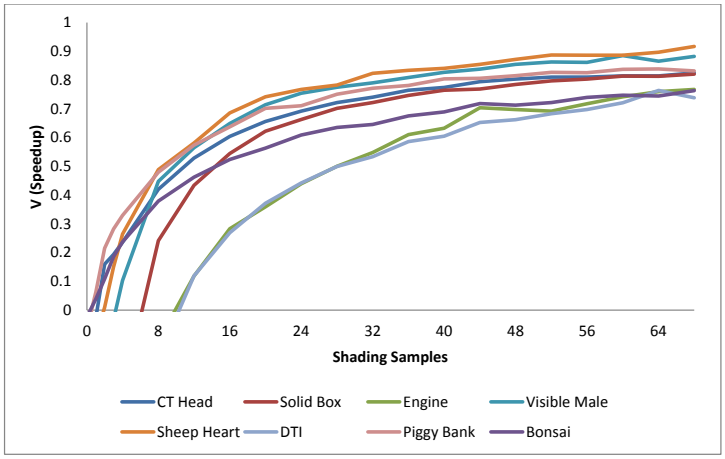
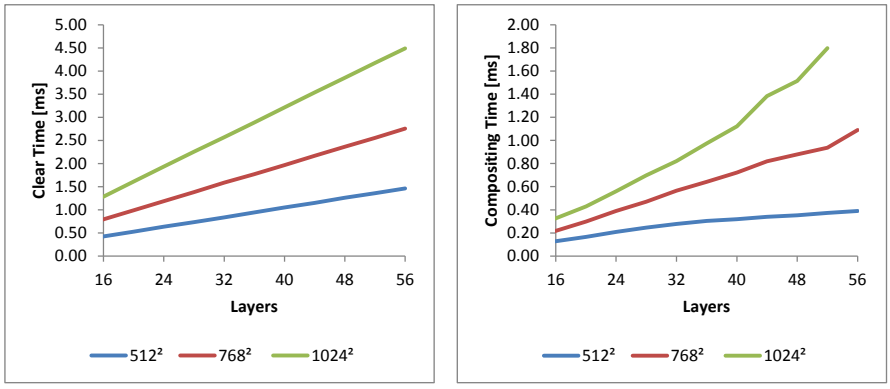


Figure 16: *Recorded speed-up V plotted against shading complexity, represented by number of additional samples required to shade a ray sample.*

Figure 16 depicts the measurements of the average speed-up versus the number of additional shading samples. All experiments were conducted at a resolution of 768^2 pixels using 32 layers. Generally, the break-even point at which the overall cost of shading outweighs the memory access overhead is reached at relatively low shading complexities. For example, with only 4 shading samples, a stereo image pair of the Visible Male volume is produced at 10.75 milliseconds, 10.2% speed-up. For experiments with lower sampling densities for reasons such as regions of fully transparent samples, lower volume resolution, frequent early ray termination, positive speed-up is reached at slightly higher shading complexity. In either case, positive speed-up is reached at real-time rendering times of 10 to 20 milliseconds per stereo image pair. At high shading complexity, speed-up of well above 75% is achieved.

The speed-up V depends on many factors. Most notably the sampling and shading complexity and the relation between the computational overhead to the total render time. The time needed each frame for clearance and compositing is nearly constant for a given size and number of layers. Figure 17 depicts measured overhead computation times. Note that while compositing is generally quick, texture clearing is more prone to become a bottleneck at short rendering times. For example, an average of 1.65 milliseconds of the 7.91 milliseconds recorded for the DTI data set (cf. Figure 14b) were spent on texture clearing. Additionally, the arbitrary image write performance is generally not as optimized as the fragment shader output pipeline. In fact, for simple volume rendering using only a transfer function look-up, but no additional shading samples, the proposed method could generally not out-perform rendering the two views seperately. The minimal focal length h dictates the near plane distance used for rendering. Fortunately for VR applications, current generation HMDs generally have a high field of view such that h is relatively small even at high resolutions.



(a) *Texture Clearing* **(b)** *Compositing*

Figure 17: *Plots of overhead computation times against number of layers for different image resolutions. Texture clearing perfomed by copying data from a pixel buffer object (PBO).*

The texture array creates a memory overhead that is not insignificant. Given an image size of w^2, layer count l, bit depth b and number of channels c the consumed memory M can simply be calculated as $M = \mathrm{w}^2 lbc$. Table 3 exemplifies the memory consumption for different image sizes and number of layers. Using higher bit depths and floating point formats reduces the effect of quantization on the values produced by ray segments.

6.2 Programmable Display Layer

The perceptive quality of the warping technique is most important. In human perception, some aspects have a stronger impact than others. The displayed image should have a very low motion-to-photons latency. Thus, what is displayed should

Size [px]	32 layers	64 layers	96 layers
512^2	64 MB	128 MB	192 MB
768^2	144 MB	288 MB	432 MB
1080^2	284.76 MB	569.53 MB	854.29 MB
1512^2	558.14 MB	1116.28 MB	1674.42 MB

Table 3: *Memory consumption for different combinations of number of layers and image sizes. Each pixel holds four* 16 *bit floating point components (RGBA).*

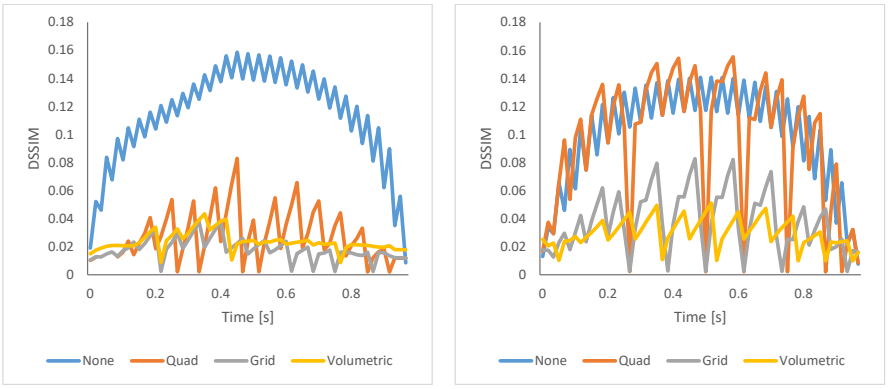
be as close to the actual head-movement as possible. To evaluate and compare this property each mode was used to compensate for virtual head-movements in a simple HMD simulation. At any given time, this allows to calculate the ideal view configuration which should be displayed on the HMD. Using this information the actual output produced by the PDL can be compared.

Tracing the DSSIM value over time gives an indication for the perceptive quality of the image stream. The closer the displayed image is to the reference view, the lower is the DSSIM value. Latency is indicated by growing DSSIM values as the reference view diverges from the displayed image. Stuttering, that occurs when due to a translational parallax effect parts of the image suddently jump to another position, usually occurs at points of large drops of the DSSIM value. However, the values don't directly reflect how pleasant or unpleasant it actually feels to the user. Still, none of the mentioned effects should be present in an ideal system.

For the simulation the HMD model by Peek et al. [50] was adopted which takes into account that a head can only rotate around its neck. This introduces small translational changes to the eye positions. Using a cosine-interpolation the model's origin at the base of the neck is either rotated or translated. For translation, the head moves a set horizontal distance. For rotation, the head rotates a set number of degrees around the y-axis. The cosine-interpolation simulates the intertia of mass as it smooths out the start and end of the animation. The duration and extents are set such that they are realistic, but relatively fast, as this is when the effects become the most noticible.

The graphs in Figure 18 show the results for the three implemented warping techniques and using none at all. Based on the last buffer swap time the ray caster renders an image using the predicted head-position. As the animation and interpolation model is known, the prediction is optimal with regards to position. The warping component in each mode uses the last result and the time for the next display update to create a warped result.

For both head movements the mode that uses no image warping (*None*) the DSSIM value does not drop back to zero which might indicate the latency between displayed image and head position. Regardless of simulation state each image is shown for the duration of approximately two display updates before the next rendering result becomes available. Note that with no latency the DSSIM value would oscillate between zero and close to the current values as only the first frame



(a) *Rotation (20°)* **(b)** *Translaton (0.5m)*

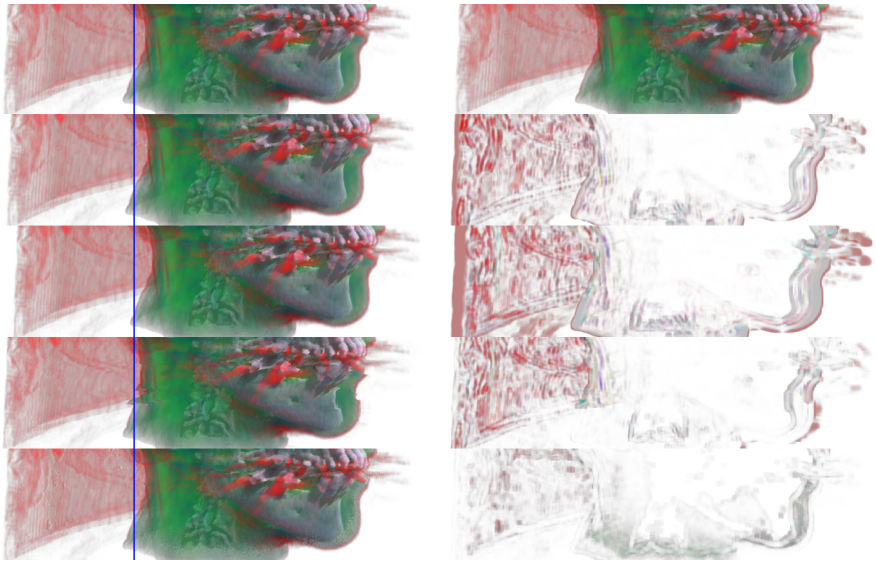
Figure 18: *Trace of displayed left image compared to reference view for implemented warping techniques. The animation lasts one second.*

coincides while the following diverge.

For the *Quad* warping mode (cf. Section 5.3.1) the error trace of rotational head movement (Figure 18a) shows that at any time the displayed image is much closer to the reference view. Over the course of a couple of frames the error slightly rises before the next ray casting result becomes available and the error drops close to zero. The *Grid* warping mode (cf. Section 5.3.2) shows even lower error, possibly by compensating for the translational changes induced by the head movement around the neck. The trace of the *Volumetric* warping mode (cf. Section 5.3.3) is the smoothest compared to the others indicating that frame updates might not be noticible at all. This smoothness may result from the reconstructed parallax effects inside the volume. The *Grid* warping method is unable to reproduce such effects.

These properties become more apparent in the error trace of translational head movement (Figure 18b). In this trace the *Quad* mode shows almost no difference to not using a warping technique at all. The *Grid* mode compensates much of the translation, but diverges from the reference as the parallax effects become more severe. However, drops are lower than with the *Quad* mode. The *Volumetric* mode again is the smoothest, but yields the longest update times. The same result is used for warping for up to five consequtive frames. Surprisingly, the DSSIM value still ranges below 5.2%.

Figure 19 shows details of a frame in the middle of the animation. In modes *None* and *Quad* the latency problem is most noticible. The *None* mode displays the last result which is already out-dated. The *Quad* mode cannot compensate for the translational difference and thus essentially also shows an image that is already more than one frame old. These deviations show in the high DSSIM values near the contours of the volume. The *Grid* mode noticibly compensates for the translation, however there are visible artifacts at the neck. Here, the grid was projected to a semi-transparent voxel in front of the neck, but painted with the color information of rays that originally ran through the neck. Yet, the average error is much smaller



(a) *Warping Results* **(b)** *DSSIM Image*

Figure 19: *Details of frame 35 (*0.583*s) of translation simulation. A blue vertical line has been added to the warping results for reference. From top to bottom: Reference,* None*,* Quad*,* Grid *and* Volumetric *warping mode.*

than with the first two modes. The *Volumetric* mode is able to compensate for a great deal of the translational changes, even inside the volume. Thus, the same neck area is warped more accurately than with the other modes. However, the ray marching implementation of the warping shader does not always reconstruct the correct opacity which leads to the more transparent areas near the chin and bottom of the neck.

The results show that for compensating rotation any of the implemented warping methods is well suited as the error is generally low. Most importantly, latency can always be reduced and intermediate frames are tracked. For translation the lack of parallax compensation of the *Quad* mode becomes an issue. At a similar update rate, the *Grid* mode is generally able to compensate for a big part of it. Even though the update rate doubles compared to the others, the *Volumetric* warp mode achieves the smoothest and closest results.

7 Conclusion

7.1 Future Work

In future work, the single-pass stereo technique is to be further enhanced regarding several shortcomings of the current implementation. Firstly, the size of the re-projection layer texture array is currently directly determined by the number of covered pixels of a re-projected ray. it is interesting to explore ways to employ

the idea of *interleaved sampling* [29] in one way or another. The general idea is to provide comparable image quality at lower sampling rates, by interleaving samples from several regular grids. In the context of the proposed stereo technique, there could be ways to reduce the number or the resolution of required textures. The sampling rate, in this context, refers to the number of layers that compose the right view. Interleaving could be established by changing the pattern in which left view's rays are assigned to layers and segments are re-projected. Possibly, the number of required textures could be further reduced. Another point of investigation is the texture clearing routine. In the current implementation, the texture is cleared by copying the content of a pre-cleared PBO. However, alternative methods exist to clear the content of a texture array, such as the OpenGL 4.4 function *glClearTexImage* [33]. it is interesting to compare execution times of these methods to evaluate whether the performance overhead can further be reduced. Additionally, it is interesting to compare the performance of the technique against state-of-the-art image warping algorithms, such as Lochmann et al. [40]. In this thesis it has been implemented as part of the PDL however using it to generate the stereoscopic view is just as viable.

Regarding the proposed asynchronous rendering system, one of the biggest concerns of the current implementation is the problem of insuring not to miss a frame. As one GPU is shared between the rendering tasks and preemption is not supported, the computation times within a frame are predicted, leaving a window of idle time for possible fluctuations. Preferably, this time window is as small as possible. Thus, it is interesting to implement the system on a multi-GPU system which would drastically simplify the problem, as both GPUs could keep rendering independently. Following the remarks of [51], the warping system could be implemented on the integrated GPU instead of a dedicated graphics card which facilitates wide-ranging applicability. However, as the OpenGL runtime generally loads only one installable client driver (ICD) at a time, an implementation could require a setup running multiple processes with shared memory. Furthermore, once the functionality of task preemption becomes available at the API level, the prediction approach could be replaced by a preferable just-in-time solution. With this, the ray casting would simply pause and switch to the warping task based on the time left until the next frame needs to be sent to the display.

Finally, the presented asynchronous rendering approach has essentially been evaluated heurisitcally, while the targeted benefits actually lie in the physiological domain. Therefore, it is interesting to evaluate it also by means of user studies, to assess to what extent the VR experience is improved with regards to the appearance of cybersickness.

7.2 Summary

This thesis has addressed the acceleration of volume rendering in the context of virtual reality applications. Accommodating current trends in consumer hardware, it specifically covers a distinctive set of problems related to HMD-based VR. The

non-negligible risk of appearance of cybersickness is countered by minimizing the magnitude of known driving factors, i.e. low frame rates and high latency. An asynchronous rendering system has been proposed which seperates the expensive ray casting task from the high-frequency displaying task. With this, the displaying task uses image warping techniques to retain a smooth frame rate and generate closely tracked views. A set of three warping techniques of varying complexity have been compared regarding their capabilities to smooth out rotational and translational movement of the head. The results suggest that even the simple image-warping technique is sufficiently capable to smooth out head rotation. The volumetric approach yields the most compelling results, while the grid-based mesh-warping technique makes for a viable, more capable alternative to the simple image-warping technique. Furthermore, a novel single-pass stereo rendering technique for GPU ray casting has been presented. To this end, the ray casting fragment shader is extended by a re-projection phase in which ray segment colors are written to layers of a texture array. The second view is efficiently produced in a fast compositing pass in which the buffered colors are accumulated. Multiple factors such as resolution, array size and field of view constrain the applicable near plane distance. Speed-up over rendering views seperately depends largely on the sampling and shading complexity. In future work, the single-pass stereo method could be enhanced to further reduce the computational overhead and to improve the image quality and the minimal near plane distance. Regarding the asynchronous rendering system, driver-level features for context switching and task preemption would be greatly beneficial for future implementations. Futhermore, a user study could reinforce the practical effect of the implemented systems.

In conclusion, the implementation of costly volume rendering in virtual reality requires special considerations regarding a users wellbeing, such that the acceleration of the stereoscopic image generation and domain-specific image-warping techniques for spatio-temporal upsampling should be considered, to secure a comfortable HMD experience.

References

[1] Stephen J. Adelson and Charles D. Hansen. Fast stereoscopic images with ray-traced volume rendering. In *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, pages 3–9, New York, NY, USA, 1994. ACM.

[2] AM Adeshinaa, R Hashimb, NEA Khalidc, and Siti ZZ Abidind. Medical volume visualization: decades of review. *Computer Science*, 1:152–157, 2012.

[3] Louis Bavoil, Steven P. Callahan, Aaron Lefohn, João L. D. Comba, and Cláudio T. Silva. Multi-fragment effects on the gpu using the k-buffer. In *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, I3D '07, pages 97–104, New York, NY, USA, 2007. ACM.

[4] Dean Beeler and Anuj Gosalia. Asynchronous timewarp on oculus rift. `https://developer3.oculus.com/blog/asynchronous-timewarp-on-oculus-rift/`, 2016. Online; accessed 04-April-2017.

[5] Dean Beeler, Ed Hutchins, and Paul Pedriana. Asynchronous spacewarp. `https://developer.oculus.com/blog/asynchronous-spacewarp/`, 2016. Online; accessed 04-April-2017.

[6] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. State-of-the-art in gpu-based large-scale volume visualization. In *Computer Graphics Forum*, volume 34, pages 13–37. Wiley Online Library, 2015.

[7] Jeff Bolz and Pat Brown. Ext_shader_image_load_store. `https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_shader_image_load_store.txt`, 2013. Online; accessed 30-June-2017.

[8] Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross. Iterative image warping. *Computer Graphics Forum*, 31(2pt1):237–246, 2012.

[9] Arend Buchacher and Marius Erdt. Single-Pass Stereoscopic GPU Ray Casting Using Re-Projection Layers. In Matthias Zwicker and Pedro Sander, editors, *Eurographics Symposium on Rendering - Experimental Ideas & Implementations*. The Eurographics Association, 2017.

[10] Sonny Chan, François Conti, Kenneth Salisbury, and Nikolas H Blevins. Virtual reality simulation in neurosurgery: technologies and evolution. *Neurosurgery*, 72:A154–A164, 2013.

[11] Weiya Chen, Anthony Plancoulaine, Nicolas Férey, Damien Touraine, Julien Nelson, and Patrick Bourdot. 6dof navigation in virtual worlds: Comparison of joystick-based and head-controlled paradigms. In *Proceedings of the 19th ACM Symposium on Virtual Reality Software and Technology*, VRST '13, pages 111–114, New York, NY, USA, 2013. ACM.

[12] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA, 2009. ACM.

[13] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive Indirect Illumination Using Voxel-based Cone Tracing: An Insight. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 20:1–20:1, New York, NY, USA, 2011. ACM.

[14] Balazs Csebfalvi and Szirmay-Kalos Szirmay-Kalos. Monte carlo volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 59–, Washington, DC, USA, 2003. IEEE Computer Society.

[15] Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. Perceptually-motivated real-time temporal upsampling of 3d content for high-refresh-rate displays. *Computer Graphics Forum*, 29(2):713–722, 2010.

[16] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS '01, pages 9–16, New York, NY, USA, 2001. ACM.

[17] Cass Everitt. Interactive order-independent transparency. `http://developer.download.nvidia.com/assets/gamedev/docs/order_independent_transparency.pdf`, 2001. Online; accessed 26-June-2017.

[18] Steffen Frey, Filip Sadlo, and Thomas Ertl. Explorable volumetric depth images from raycasting. In *Proceedings of the 2013 XXVI Conference on Graphics, Patterns and Images*, SIBGRAPI '13, pages 123–130, Washington, DC, USA, 2013. IEEE Computer Society.

[19] L. Gallo, G. D. Pietro, and I. Marra. User-friendly inspection of medical image data volumes in virtual environments. In *2008 International Conference on Complex, Intelligent and Software Intensive Systems*, pages 749–754, March 2008.

[20] Brian Guenter, Mark Finch, Steven Drucker, Desney Tan, and John Snyder. Foveated 3d graphics. *ACM Trans. Graph.*, 31(6):164:1–164:10, November 2012.

[21] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006.

[22] Markus Hadwiger, Andrea Kratz, Christian Sigg, and Katja Bühler. Gpu-accelerated deep shadow maps for direct volume rendering. In *SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware: Proceedings of the 21 st ACM SIGGRAPH/Eurographics symposium on Graphics hardware: Vienna, Austria*, volume 3, pages 49–52, 2006.

[23] Taosong He and Arie Kaufman. Fast stereo volume rendering. In *Proceedings of the 7th Conference on Visualization '96*, VIS '96, pages 49–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[24] Rama Karl Hoetzlein. Gvdb: Raytracing sparse voxel database structures on the gpu. In *Proceedings of High Performance Graphics*, HPG '16, pages 109–117, Aire-la-Ville, Switzerland, Switzerland, 2016. Eurographics Association.

[25] Thomas Hübner and Renato Pajarola. Single-pass multi-view volume rendering. In *Proceedings of IADIS Multi Conference on Computer Science and Information Systems*, pages 50–58, 2007.

[26] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. Volume rendering techniques. *GPU Gems: Programming Techniques Tips and Tricks for Real-Time Graphics*, pages 667–692, 2004.

[27] James T Kajiya and Brian P Von Herzen. Ray tracing volume densities. In *ACM Siggraph Computer Graphics*, volume 18, pages 165–174. ACM, 1984.

[28] Arie Kaufman and Klaus Mueller. Overview of volume rendering. *The visualization handbook*, 7:127–174, 2005.

[29] Alexander Keller and Wolfgang Heidrich. Interleaved Sampling. In S. J. Gortle and K. Myszkowski, editors, *Eurographics Workshop on Rendering*. The Eurographics Association, 2001.

[30] Robert S. Kennedy, Norman E. Lane, Kevin S. Berbaum, and Michael G. Lilienthal. Simulator sickness questionnaire: An enhanced method for quantifying simulator sickness. *The International Journal of Aviation Psychology*, 3(3):203–220, 1993.

[31] Franklin King, Jagadeesan Jayender, Steve Pieper, Tina Kapur, Andras Lasso, and Gabor Fichtinger. An immersive virtual reality environment for diagnostic imaging. 10 2015.

[32] Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on visualization and computer graphics*, 8(3):270–285, 2002.

[33] Daniel Koch. Arb_clear_texture. https://www.khronos.org/registry/OpenGL/extensions/ARB/ARB_clear_texture.txt, 2013. Online; accessed 1-July-2017.

[34] E.M. Kolasinski, U.S. Army Research Institute for the Behavioral, and Social Sciences. *Simulator sickness in virtual environments*. Number Bd. 4,Nr. 1027 in Technical report (U.S. Army Research Institute for the Behavioral and Social Sciences). U.S. Army Research Institute for the Behavioral and Social Sciences, 1995.

[35] J. Kronander, D. Jonsson, J. Low, P. Ljung, A. Ynnerman, and J. Unger. Efficient visibility encoding for dynamic illumination in direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 18(3):447–462, March 2012.

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

[37] Aaron Leiby. Steamvr beta updated (1477423729). https://steamcommunity.com/games/250820/announcements/detail/599369548909298226, 2016. Online; accessed 04-April-2017.

[38] Marc Levoy. Display of surfaces from volume data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988.

[39] Gerard Llorach, Alun Evans, and Josep Blat. Simulator sickness and presence using hmds: Comparing use of a game controller and a position estimation system. In *Proceedings of the 20th ACM Symposium on Virtual Reality Software and Technology*, VRST '14, pages 137–140, New York, NY, USA, 2014. ACM.

[40] Gerrit Lochmann, Bernhard Reinert, Arend Buchacher, and Tobias Ritschel. Real-time Novel-view Synthesis for Volume Rendering Using a Piecewise-analytic Representation. In Matthias Hullin, Marc Stamminger, and Tino Weinkauf, editors, *Vision, Modeling & Visualization*. The Eurographics Association, 2016.

[41] Gerrit Lochmann, Bernhard Reinert, Tobias Ritschel, Stefan Müller, and Hans-Peter Seidel. Real-time Reflective and Refractive Novel-view Synthesis. pages 9–16, Darmstadt, Germany, 2014. Eurographics Association.

[42] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *COMPUTER GRAPHICS*, 21(4):163–169, 1987.

[43] William R Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. In *Proceedings of the 1997 symposium on Interactive 3D graphics*, pages 7–ff. ACM, 1997.

[44] Michael E. McCauley and Thomas J. Sharkey. Cybersickness: Perception of self-motion in virtual environments. *Presence: Teleoper. Virtual Environ.*, 1(3):311–318, January 1992.

[45] Morgan McGuire and Louis Bavoil. Weighted blended order-independent transparency. *Journal of Computer Graphics Techniques (JCGT)*, 2(2):122–141, December 2013.

[46] Jörg Mensmann, Timo Ropinski, and Klaus Hinrichs. Accelerating volume raycasting using occlusion frustums. In *Proceedings of the Fifth Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG'08, pages 147–154, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[47] NVIDIA® VRWorks™. Vrworks - context priority. `https://developer.nvidia.com/vrworks/headset/contextpriority`, 2016. Online; accessed 05-April-2017.

[48] Marc Olano, Jon Cohen, Mark Mine, and Gary Bishop. Combatting rendering latency. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, I3D '95, pages 19–ff., New York, NY, USA, 1995. ACM.

[49] P. V. F. Paiva, L. S. Machado, and J. C. d. Oliveira. A peer-to-peer multicast architecture for supporting collaborative virtual environments (cves) in medicine. In *2012 14th Symposium on Virtual and Augmented Reality*, pages 165–173, May 2012.

[50] Edward Peek, Christof Lutteroth, and Burkhard Wünsche. More for less: Fast image warping for improving the appearance of head tracking on hmds. In *2013 28th International Conference on Image and Vision Computing New Zealand (IVCNZ 2013)*, pages 41–46. IEEE, 2013.

[51] Edward Peek, Burkhard Wünsche, and Christof Lutteroth. Using integrated gpus to perform image warping for hmds. In *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*, IVCNZ '14, pages 172–177, New York, NY, USA, 2014. ACM.

[52] Bui Tuong Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[53] Lisa Rebenitsch and Charles Owen. Review on cybersickness in applications and visual displays. *Virtual Reality*, 20(2):101–125, Jun 2016.

[54] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart hardware-accelerated volume rendering. In *Proceedings of the Symposium on Data Visualisation 2003*, VISSYM '03, pages 231–238, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[55] R. M. Satava and S. B. Jones. Current and future applications of virtual reality for medicine. *Proceedings of the IEEE*, 86(3):484–489, Mar 1998.

[56] Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. Layered depth images. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 231–242, New York, NY, USA, 1998. ACM.

[57] F. Smit, R. van Liere, S. Beck, and B. Froehlich. An image-warping architecture for vr: Low latency versus image quality. In *2009 IEEE Virtual Reality Conference*, pages 27–34, March 2009.

[58] F. A. Smit, R. van Liere, and B. Fröhlich. An image-warping vr-architecture: Design, implementation and applications. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology*, VRST '08, pages 115–122, New York, NY, USA, 2008. ACM.

[59] M. Solh and G. AlRegib. Hierarchical hole-filling for depth-based view synthesis in ftv and 3d video. *IEEE Journal of Selected Topics in Signal Processing*, 6(5):495–504, Sept 2012.

[60] Kay M Stanney, Robert S Kennedy, and Julie M Drexler. Cybersickness is not simulator sickness. In *Proceedings of the Human Factors and Ergonomics Society annual meeting*, volume 41, pages 1138–1142. SAGE Publications Sage CA: Los Angeles, CA, 1997.

[61] Ivan E. Sutherland. The ultimate display. In *Proceedings of the IFIP Congress*, pages 506–508, 1965.

[62] Stefan Suwelack, Eric Heitz, Roland Unterhinninghofen, and Rüdiger Dillmann. Adaptive gpu ray casting based on spectral analysis. In *Proceedings of the 5th International Conference on Medical Imaging and Augmented Reality*, MIAR'10, pages 169–178, Berlin, Heidelberg, 2010. Springer-Verlag.

[63] J. M. P. van Waveren. The asynchronous time warp for virtual reality on consumer hardware. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 37–46, New York, NY, USA, 2016. ACM.

[64] Andreas A. Vasilakis and Ioannis Fudos. K+-buffer: Fragment synchronized k-buffer. In *Proceedings of the 18th Meeting of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, I3D '14, pages 143–150, New York, NY, USA, 2014. ACM.

[65] Sebastian von Mammen, Andreas Knote, and Sarah Edenhofer. Cyber sick but still having fun. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 325–326, New York, NY, USA, 2016. ACM.

[66] Ming Wan, Nan Zhang, Huamin Qu, and Arie E. Kaufman. Interactive stereoscopic rendering of volumetric environments. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):15–28, January 2004.

[67] Junpeng Wang, Fei Yang, and Yong Cao. Cache-aware sampling strategies for texture-based ray casting on gpu. In *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pages 19–26, Nov 2014.

[68] Zhou Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*, 13(4):600–612, April 2004.

[69] Manfred Weiler, Rüdiger Westermann, Chuck Hansen, Kurt Zimmermann, and Thomas Ertl. Level-of-detail volume rendering via 3d textures. In *Proceedings of the 2000 IEEE Symposium on Volume Visualization*, VVS '00, pages 7–13, New York, NY, USA, 2000. ACM.

[70] Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. Image-based bidirectional scene reprojection. *ACM Trans. Graph.*, 30(6):150:1–150:10, December 2011.