Volume Renderer

For use with MATLAB®

Documentation
Version 1.0

Raphael Schmitt
`schmittr@cs.uni-freiburg.de`

14.07.2012

Acknowledgments

This software was developed as a student project at

Albert-Ludwigs-University Freiburg im Breisgau
Department of Computer Science
Chair of Pattern Recognition and Image Processing.
Image Analysis Group

This project was supervised by Benjamin Ummenhofer and Junior-Prof. Dr. Olaf Ronneberger. Herewith, I thank my supervisors for the patience and support.

Contents

Notice

Tademarks

Copyright

As the program consists of an open source and a closed source part, the usage of two licenses is required. The open source part is published under New BSD License:

Chapter 1

Introduction

In medical and biological research volumetric data sets are commonly used. They are recorded by imaging methods like X-rays, magnetic resonance imaging and 3D microscopy. To deal with complex 3D data, 2D projections are often used for analysis and visualization. This mapping from 3D to 2D is done via a render process. This process uses a common ray casting approach that ends up with a 2D view of the 3D volume. To offer a physically plausible view and multiple possible perspectives several manipulation processes are necessary.
In order to handle all the data efficiently, programs like *MATLAB*® have been developed. *MATLAB*® provides an interface with its own script-language offering optimized functions for vector calculation. In older versions, many of these operations are still computed on the CPU. Since vector computation is in many cases very convenient for a high parallelization, a computation on the GPU can offer much more performance compared to the CPU. In addition, the resolution of volumetric data will successively increase in future. Thus, fast algorithms to manipulate and handle these data are of utmost importance.

1.1 Contribution

In this project we built an efficient *MATLAB*® offline render command that handles 3D volumetric data. To guarantee fast computations the render process computes on the GPU. Due to restricted GPU memory and the requirement to render more than one volume in one scene, we developed a special memory management to enable the rendering of huge data sets in separate rendering passes. Afterward these separately rendered images are combined to one image using *MATLAB*® . Additionally we developed a generic illumination model that is easy to extend with other illumination functions.
What is more, in some use cases there is a nice feature to work with stereo images. Thus, our renderer offers the possibility to render off-axis stereo images.
To enable a high usability we developed a *MATLAB*® interface consisting of several *MATLAB*® classes that is easy to use. Due to this interface it is uncomplicated to generate movies.

1.2 Structure of this work

In Chapter 2 we explain the theory on which *Volume Renderer* is based. This includes the rendering equation and the rendering pipline with all its components.
Furthermore, we introduce our architecture and implementation details in Chapter 3 and go into detail in the following sections. There we explain our memory management in Section 3.3 and our illumination model interface in Section 3.4.
Section 3.5 explains how the *MATLAB*® interface is designed. Moreover, we explain all its properties and options. Only small examples are given here. For more detailed code examples you should take a look at Appendix A.
In the end, in Chapter 4 we discuss our result and suggest some future work.

Chapter 2

Background

In this chapter we explain the theoretical background of the renderer. First the rendering equation is introduced and discretized. Section 2.2 explains how the rendering pipeline is working. Its operations sampling, illumination and compositing are introduced as well. Finally, in Section 2.3 we explain the theoretical background of stereo rendering.

2.1 Rendering Equation

There are two methods to render a volume: indirect and direct volume rendering (see Figure 2.1).
Indirect rendering is used to render the isosurface of a volume. First the generation of an intermediate representation of the data set is required (e.g. a polygonal representation of an isosurface generated by Marching Cubes [7]). The second step is the rendering of this representation. In a polygonal representation only a limited number of the 3D data set directly contributes to the 2D output. The interior of the volume is not considered.
Since in biological or medical application the interior is of utmost importance, for our application we use direct rendering [6]. Direct rendering techniques do not need the generation of an intermediate representation of the data set, since every voxel contributes to the resulting 2D image. To increase the realism of the rendering process, we use a model that takes

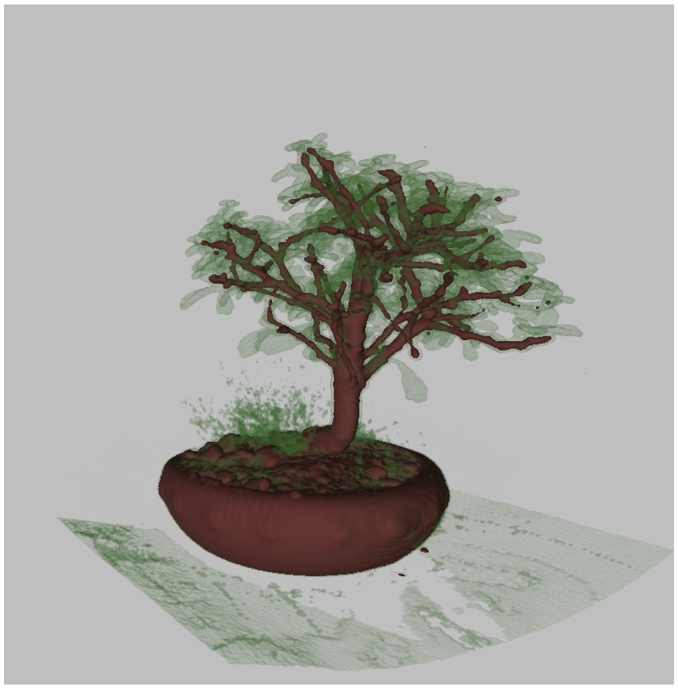


Figure 2.1: This illustration shows a CT scan of a bonsai. The leafs are visualized by direct rendering whereas the trunk and the branches are visualized using an isosurface rendering. Obviously in the direct rendering each voxel contributes to the resulting 2D image. (Source: [11])

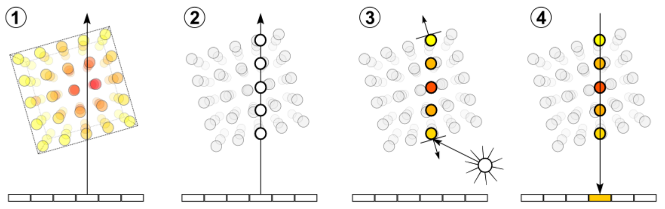


Figure 2.2: This illustration shows the 4 main steps of the rendering process: (1) ray intersection (2) sampling (3) shading and (4) compositing. As our renderer uses a simplified model, we neglect the attenuation of light on the path from the light source to an enlighted particle. What is more we neglect the attenuation of light from the volume to the camera as well. (Source: [14])

emission and absorption into account. In this model every particle emits and absorbs light. The combination of all these properties maintains following rendering equation. We use a simplified version of the equation introduced by [8]:

$$\begin{aligned} I(D) &= \int_0^D g(s) \cdot e^{\left(-\int_s^D \tau(t)\mathrm{d}t\right)} \mathrm{d}s \\ &= \int_0^D g(s) \cdot t(s)\mathrm{d}s \end{aligned} \tag{2.1}$$

This equation integrates from 0, the edge of the volume, to D at the eye. $I(D)$ describes the value of accumulated light intensity of one ray traversing the volume. $\tau(t)$ is the absorption at t and $g(s)$ is the sampled value at s. In fact, the equation consists of two main terms: $g(s)$ denotes emission and $t(s)$ the transparency at point s.

2.1.1 Emission

In general the emission part of the used model described in Section 2.1 assumes that each particle of the volume denotes a tiny light source. Thus, even if no external light source is applied, the volume can emit a several amount of light. This behavior is motivated by the application:

1. We model self-luminous particles. This occurs during fluorescence microscopy, as certain proteins glow.
2. The particle is illuminated by a light source. This leads to a better spatial impression.

In the used model this light is not scattered. What is more, we neglect the attenuation of light on the path from the light source to an enlighted particle. What is more, we neglect the attenuation of light from the volume to the camera as well. This is not completely physically correct but enables a simple model and thus a high computational performance. Figure 2.2 illustrates this simplification.

2.1.2 Absorption

In our case absorption describes the attenuation of light. Eq. 2.2 describes the transparency between point s and the eye. If we assume a transparency of $t \in [0..1]$ we can simply reformulate this transparency such that it becomes opacity by $\alpha(s) = 1 - \tau(s)$.

$$t(s) = e^{\left(-\int_s^D \tau(t)\mathrm{d}t\right)}\mathrm{d}s \tag{2.2}$$

2.1.3 Discretization

Since Eq. 2.1 cannot be solved analytically for all interpolation methods, to obtain a general discretization one has to solve it numerically. This can be easily done by the Riemann sum and a fixed step size Δs. The step size scales the current sampling position along the ray. More precisely, the ray gets divided into n equal segments, each of size Δs [8].

$$
\begin{aligned}
t(s) = e^{\left(-\int_s^D \tau(t)\mathrm{d}t\right)}\mathrm{d}s &\approx e^{\left(\sum_{i=i+1}^{n} \tau(k\cdot\Delta t)\Delta t\right)} \\
&= \prod_{j=i+1}^{n} e^{(\tau(k\cdot\Delta t)\Delta t)} \\
&= \prod_{j=i+1}^{n} t_j
\end{aligned}
\tag{2.3}
$$

Thus, we end up with the discretized rendering equation that approximates Eq. 2.1:

$$
\begin{aligned}
I(D) &= \int_0^D g(s)\cdot e^{\left(-\int_s^D \tau(t)\mathrm{d}t\right)}\mathrm{d}s \\
&\approx \sum_{i=1}^{n} g(i\cdot\Delta s)\Delta s \cdot \prod_{j=i+1}^{n} t_j \\
&= \sum_{i=1}^{n} g_i \cdot \prod_{j=i+1}^{n} t_j
\end{aligned}
\tag{2.4}
$$

2.2 Rendering Pipeline

The rendering pipeline defines the order of the operations that are processed to compute the discretized rendering equation.
Our renderer uses the three operations *sampling*, *illumination* and *compositing* that are explained more precisely in the following sections.

2.2.1 Sampling

As described in Section 2.1.3 the ray is divided into n equal segments. This transformation of a continuous ray to discretized volume intersection locations is called sampling. To perform good quality, the discretized ray

positions are interpolated. What is more, in order to avoid artifacts, the sampling rate should be twice as high as the grid resolution [12].
Each image pixel is computed by one ray. A ray passes through the center of its corresponding pixel.

2.2.2 Illumination

To improve the realism of the rendered scene we provide an interface for local illumination techniques. This interface is documented in Section 3.4. We decided to use the Henyey-Greenstein phase function [3].

$$HG(\theta, g) = \frac{1}{4\pi} \cdot \frac{(1-g^2)}{[1+g^2-2g\cdot cos(\theta)]^{3/2}} \tag{2.5}$$

HG behaves physically more correct than other shading functions like e.g. Blinn-Phong [1] but is also computationally inexpensive. Figure 2.3 depicts the function for several g.
At each sample position this function is evaluated locally. This illumination model ignores scattering as well as diffusion. Any interaction with other particles are neglected and the light arrives unextenuated at the sample point.

2.2.3 Compositing

Compositing denotes the accumulation of the sampled, illuminated and colored values along the ray to a coherent result. Therefore, in this case we use the discretized rendering equation we explained in the previous section.
One can reformulate Eq. 2.4 and use the basic compositing *over*-operator

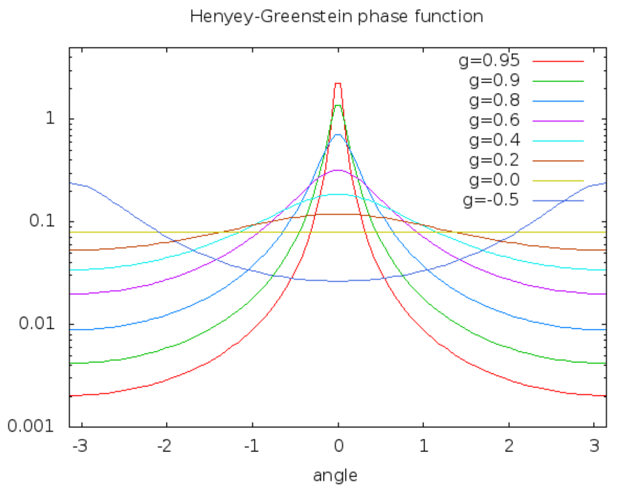


Figure 2.3: Henyey-Greenstein phase function with different g.

introduced by [10]:

$$\begin{aligned} I(D) &= \sum_{i=1}^{n} g_i \cdot \prod_{j=i+1}^{n} t_j \\ &= g_n + t_n(g_{n-1} + t_{t-1}(g_{n-2} + t_{n-2}(g_{n-3} + \cdots (g_1 + t_0 \cdot I_0) \cdots))) \\ &= g_n \text{ over } (g_{n-1} \text{ over } (g_{n-2} \text{ over } (g_{n-3} \text{ over } \cdots (g_1 \text{ over } 0) \cdots))) \end{aligned} \tag{2.6}$$

Since our samples are sorted in front-to-back order our renderer uses the *under*-operator. Combined with alpha compositing this yields to the following equations [4]:

$$\begin{aligned} \hat{c}_i &= (1 - \hat{\alpha}_{i-1})c_i + c_{i-1}, \\ \hat{\alpha}_i &= (1 - \hat{\alpha}_{i-1})\alpha_i + \alpha_{i-1}, \end{aligned} \tag{2.7}$$

where $\hat{c}_i$ and $\hat{\alpha}_i$ denotes the accumulated color including the illumination and opacity as described in Section 2.1.2. Besides front-to-back compositing has the advantage, that the ray can be stopped if a given threshold (of

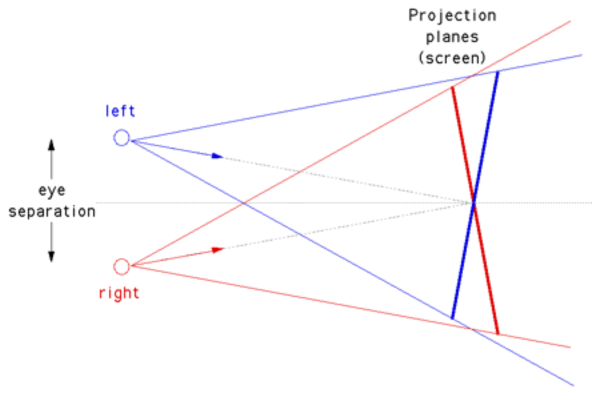


Figure 2.4: In the toe-in camera setup both cameras have the same focal point. (Source: [2])

transparency) is reached.

2.3 Stereo Render

There are many possibilities for a stereo rendering setup. Some of these approaches do not produce correct results.
Toe-in is easy to implement. Since in this projection the two stereo cameras are pointing to the same focal point, a rotation of the object suffices to render the two images. Unfortunately, one suffers from vertical parallax. The higher the distance to the center of the projection plane, the more this effect occurs. Thus this approach does not result in correct stereo images. Figure 2.4 illustrates this rendering setup.

2.3.1 Off-axis

Off-axis is the correct projection. No vertical parallax is introduced. Both cameras have a different focal point (see Figure 2.5). The viewing directions are parallel. Thus it is necessary to render two images with different camera view frustums. As we can see in Figure 2.6 the two extended camera frustums do not overlap completely. To obtain the off-axis projection plane

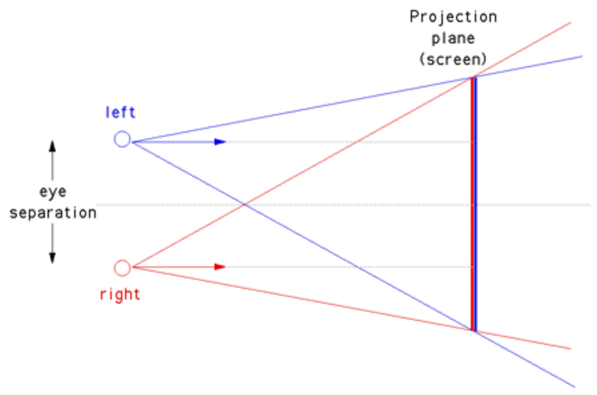


Figure 2.5: The correct off-axis stereo camera setup. (Source: [2])

of both images, one has to trim the projection of the extended frustums. Therefore, one has to compute the non-overlapping amount of pixels δ [2]:

$$\delta = \frac{b \cdot w}{2 \cdot f_o \cdot tan(\frac{\alpha}{2})}, \tag{2.8}$$

where w is the image width in pixel and f_o is the focal length. b is the stereo base, i.e. half of camera x-offset. The angle of view α can be computed as follows:

$$\begin{aligned} \alpha &= 2 \cdot arctan(\frac{d}{2 \cdot f_o}) \\ &= 2 \cdot arctan(\frac{1}{f_o}) \end{aligned} \tag{2.9}$$

where d is the width of the normalized image plane. In our case $d = 2$ because the range of the normalized image plain goes from -1 to 1.
If now a stereo image of resolution w x h is rendered, first this resolution will be extended by δ. Then the left and right images with a resolution of $(w + \delta)$ x h are rendered. Finally, to obtain the off-axis projection plane both images are trimmed respectively to w x h again.

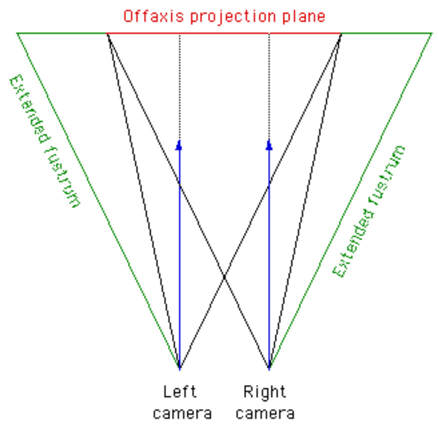


Figure 2.6: The extended frustums are depicted. To obtain the offaxis projection plane one has to trim the projection plane of each extended frustrum. (Source: [2])

Chapter 3

Implementation

In the following sections implementation details are introduced. In Section 3.1 we explain the choice of the used hard and software and give an overview of how the different components work together. Furthermore we illustrate the ray casting. This includes the memory management and the illumination model interface. For the latter we also show an example in Section 3.4. Finally, in Section 3.5 we introduce the *MATLAB*® interface that enables an easy handling of the renderer.

3.1 Architecture

To obtain a high computational performance we decide to use the *NVIDIA CUDA* toolkit as it provides an interface for highly parallelized computation on GPU. Since *CUDA* is only available for *NVIDIA* devices, we are restricted in the choice of the graphic card. Moreover, due to some features we use in our code compute capability 2.0 is required that was introduced with the *Fermi* architecture by *NVIDIA* .
CUDA provides a special *CUDA* -C compiler. This GPU device code can easily be connected to the host (CPU) C++ code. The host code provides some data structures and functions that handle the communication between host and device. Using the *MATLAB*® mex-interface we built a *MATLAB*® command. For this purpose, *MATLAB*® provides a special

compiler, that translates all the compiled C object files into a $MATLAB^{®}$ command. As the $MATLAB^{®}$ command requires a lot of parameters we developed some wrapper classes with special properties to make the handling more comfortable.

3.2 Ray casting

Ray casting describes the process of shooting rays starting from the viewer/ camera through the volume. If a ray intersects the volume, an accumulated light intensity is determined while traversing the volume as explained in Section 2.2. For each ray a test is performed if it intersects the volume. As we only provide the opportunity to render one object per rendering pass, this test is only performed once for each ray. As [15] introduces a fast intersection test approach that also performs well if there is only one intersection per ray, we decided to implement a slightly simplified version of this algorithm. In our case of only one intersection test per ray we omit complex data structures.
Moreover, we defined the origin of the world coordinate frame in the middle of the volume. A scalar d describes the distance to the volume. Through this, the camera is moved along the negative z-axis to its position. The distance between projection plane and camera is defined to 1. What is more, the focal length f_o can be customized. With a rotation matrix $\mathbf{R}$ we can now rotate the camera around the object. Figure 3.1 depicts such a example scene.
The projection plane is defined in normalized coordinates from $[-1, -1]$ to $[1, 1]$. Each rendered pixel value is determined by one ray. Thus, we have to project the respective x and y coordinate of the ray to the normalized projection plane coordinates u and v.
With all these information, we can compute the direction of a ray and its

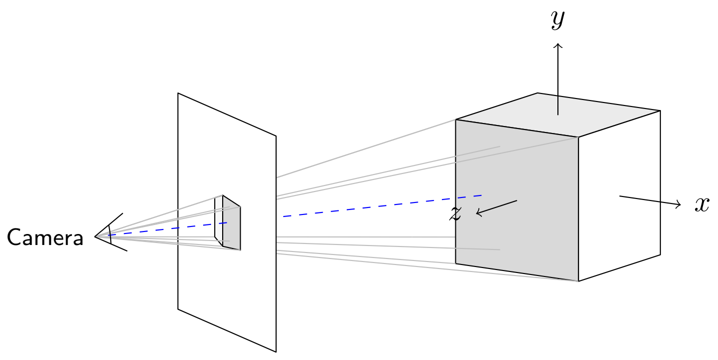


Figure 3.1: Projection of a volume onto the projection plane. The blue dashed line depicts the distance to volume. Additionally the eight rays of the object's corners are drawn.

position with

$$\begin{aligned} dir_{ray} &= \frac{u \cdot \vec{x} + v \cdot \vec{y} + f_o \cdot \vec{z}}{||u \cdot \vec{x} + v \cdot \vec{y} + f_o \cdot \vec{z}||} \\ pos_{ray} &= c_o \cdot \vec{x} + (-1) \cdot d \cdot \vec{z} \end{aligned} \tag{3.1}$$

$\vec{x}$, $\vec{y}$ and $\vec{z}$ are the particular column vectors of the rotation matrix. c_o is the camera x-offset that can be defined by the user.
Since we are computing on the GPU each ray can be computed by one *CUDA* thread, i.e. each pixel of the rendered image is computed by one thread. Thus, we obtain a highly parallelized rendering program.

3.3 Memory Management

GPU memory is very limited and not extendable. Usually our renderer requires six different volumes: one for emission, one for absorption, one for reflection and one for each gradient direction. If all these volumes are copied to the GPU this might lead to a high memory consumption. In some cases these volumes can be similar or they differ only by a scalar factor. E.g. one can have an emission volume vol_{em} and an absorption volume $vol_{ab} = \alpha \cdot vol_{em}$. Due to the texture mapping *NVIDIA CUDA* provides to perform efficient lookups, it is possible to map one volume to multiple

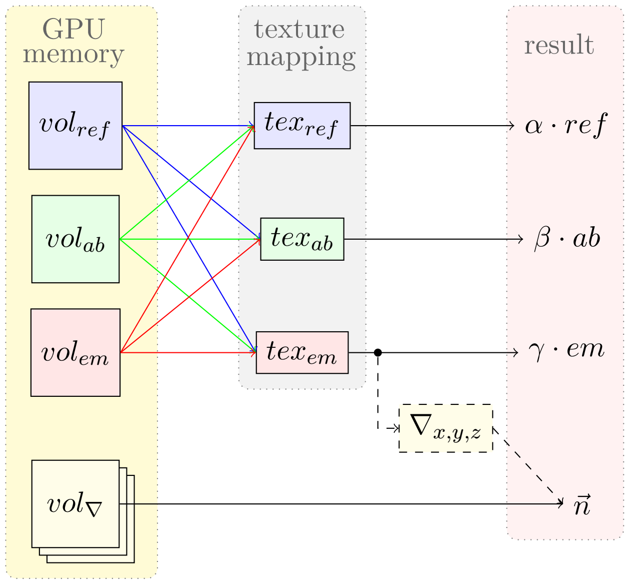


Figure 3.2: In order to save GPU memory one volume can be mapped to multiple textures. Additionally, the gradient can be computed on the fly. Thus, it is possible to setup the renderer with only one volume. This can be required if one would render a high-resolution volume. α, β and γ are scalar multiplicators that can be defined to adjust the looked up values.

textures [9]. This enables us to map vol_{em} to tex_{ab}. To provide the possibility that the looked up value is multiplied by a scalar factor, the renderer is enabled to setup one scalar multiplicator for the emission, absorption and reflection volume. To be able to save more GPU memory one can setup the renderer to compute the gradient on the fly. As expected this is computationally more expensive, especially if a movie sequence of one scene is rendered. Figure 3.2 shows the possible options.

The illumination volume and the light sources are copied to the GPU memory as well. As described in Section 2.1 every particle denotes a tiny light source, it is also possible to run the renderer without any light source.

3.4 Illumination Model Interface

In order to provide a maximum of freedom for the illumination model, we use a 3D lookup table.
The lookup table describes the interaction of a particle with a light source as depicted in Figure 3.3. Since we know where the light source and the view point are located, the vector of the incoming light $\vec{L}_i$ and the vector of the outgoing light $\vec{L}_o$ are known. $\vec{L}_o$ equals the view direction. The normal vector $\vec{n}$ is approximated by the negative gradient. As described in Section 3.3 the gradient is either determined by vol_∇ or computed on the fly using the finite difference scheme. With this information α and β can be computed.
γ is the angle between $\vec{L}_i'$ and $\vec{L}_o'$, the projections of $\vec{L}_i$ and $\vec{L}_o$ onto the surface plane. We can compute these projected vectors as follows:

$$\begin{aligned}\vec{L}_i' &= \vec{L}_p - \langle\vec{L}_i, \vec{n}\rangle\vec{L}_i \\ \vec{L}_o' &= \vec{V}_p - \langle\vec{L}_o, \vec{n}\rangle\vec{L}_o\end{aligned} \tag{3.2}$$

where $\vec{L}_p$ is the position of the light source and $\vec{C}_p$ is the $\vec{V}_p$ is the view point.
After the calculation of these angles the light intensity can be established by performing a look up. Since the LUT contains only a finite number of entries, linear interpolation is applied. The underlying LUT can be built up with a lot of illumination models. In the next section we present an example of how to build such a LUT.

Example: Henyey-Greenstein

As introduced in Section 2.2.2 our renderer uses the Henyey-Greenstein phase function to compute the light intensity.

$$HG(\theta, g) = \frac{1}{4\pi} \cdot \frac{(1-g^2)}{[1+g^2-2g\cdot cos(\theta)]^{3/2}} \tag{3.3}$$

The LUT is built up with the angles α, β and γ. Unfortunately, θ is the

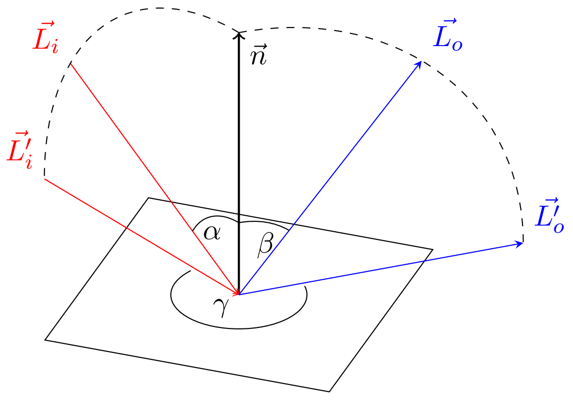


Figure 3.3: The angles α, β and γ suffice to describe the whole illumination scene. $\vec{L}_i$ is the vector of incoming light and $\vec{L}_o$ is the vector of outgoing light towards the viewer. $\vec{L}'_i$ and $\vec{L}'_o$ are the projections of these vectors onto the surface plane.

angle between $\vec{L}_i$ and $\vec{L}_o$. Thus we have to compute θ for each α, β and γ. In order to compute the LUT we choose

$$\vec{L}_o = \begin{pmatrix} \sin(\alpha) \\ \sin(\alpha) \\ 1 \end{pmatrix}, \vec{L}_i = \begin{pmatrix} \sin(\beta) \\ \sin(\beta) \\ 1 \end{pmatrix} \text{ with } ||\vec{L}_o|| = ||\vec{L}_i|| = 1 \tag{3.4}$$

while α and β are iterated respectively dependend on the resolution of the LUT. To perform the rotation around the surface normal we build a rotation matrix $\mathbf{R}$ around the surface normal dependend on γ. Because of the properties of the unit circle that we are working with, the rotation matrix is around the x-axis. $\vec{L}_i$ is kept fix while $\vec{L}_o$ is rotated. Now γ can be computed by using the dot-product:

$$\begin{aligned} \vec{rot} &= \vec{L}_o \cdot \mathbf{R}, \\ \gamma &= \langle \vec{rot}, \vec{L}_i \rangle. \end{aligned} \tag{3.5}$$

Because the three angles are iterated, we end up in a three time nested loop. To obtain high performance we decided to implement the computation of the LUT in C++ and connected this to *MATLAB*® . Thus we provide a fast computation of this LUT via a *MATLAB*® command. The parameters are the resolution of the LUT and g (see Eq. 3.3). As mentioned before, the LUT contains only a finite number of entries. Thus, during a lookup linear interpolation is applied.

3.5 *MATLAB*® Interface

The compiled *MATLAB*® command does take a lot of input parameters. Thus, we decided to write some wrapper classes. In the following sections we explain the design of these classes and how this interface works.
Figure 3.4 shows the inheritance and relations of the *MATLAB*® classes we provide and use. In the flollowig chapter these classes are described in UML like notation. Since *MATLAB*® does not have a strict type system, we are checking types manually in the setter functions of the appropriate attribute. To provide a short writing we define a few just for notation. Table 3.1 explains these types in the meaning of *MATLAB*® data types. Code examples are listed in Appendix A.
Important: All vectors are given by [level, row, column]. Apart from *ElementSizeUm*, all extents of distances and positions are relative to the world coordinate frame.

3.5.1 Handle Superclass

The realization of the memory management described in Section 3.3 requires some special techniques on the part of *MATLAB*® . Usually a *MATLAB*® class member work with call by value. But since we require the possibility to check the pointer of the volumes, call by reference is necessary. More precisely, on the C++ side we want to check if the volumes are pointing to different pointer adresses or to the same one. Consequently only the unique volume data are copied to the GPU whereas the other assignments

notation type	*matlab meaning*
scalar	a scalar value of size [1,1]
vec2	2D row vector of size [1,2]
vec3	3D row vector of size [1,3]
m3x3	a 3x3 matrix
array2	a 2D matrix of userdefined extent, such as an greyscale 2D image
array3	a 3D matrix of userdefined extent such as a volumetric data set (greyscale) or a RGB image
array4	a 4D matrix of userdefined extent such as image sequence of RGB images. 4th component is time.
X[ ]	this denotes a vec2 with data of type X

Table 3.1: Types for notation and its *MATLAB*® meaning.

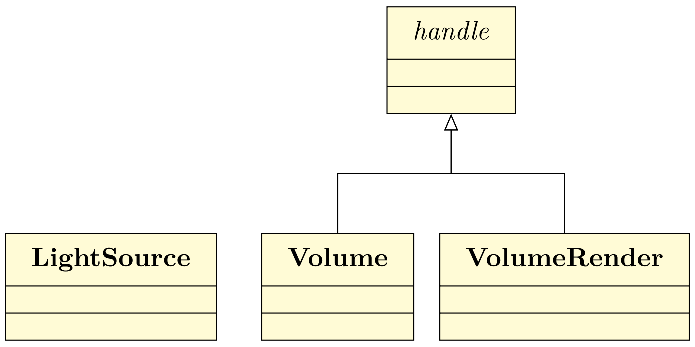


Figure 3.4: To provide call by reference instead of call by value *Volume* and *VolumeRender* inherit *handle*. Additionally, the assignment of members does not return a deep copy of the object. Since, LightSource does only consume low memory it does not inherit *handle*.

are realized by texture mapping to the assigned volume data. For us this seemed to be the most user friendly way to implement our memory model. Fortunately, *MATLAB*® offers the possibility for call by reference. A class that inherits from the special *handle* superclass automatically uses call by reference instead of call by value [13]. Hence, the class *Volume* inherits *handle*. Through this, all properties stored in a *Volume* object are pointer. What is more, a regular value assignment through a setter can be expensive, since the old object is replaced by the new one. This new object is finally returned.

```
methods
  function obj=set.SomeValue(obj, newValue) % Value class
end
```

This is a regular member of a *MATLAB*® value class. The modified *obj* with the new assigned value is returned. In case of huge data this could be very inefficient. Handle classes are not required to return any object:

```
methods
  function set.SomeValue(obj, newValue) % Handle class
end
```

As the volumetric data can be large we decided to derive the class *VolumeRender* from *handle*, as well. Figure 3.4 illustrates the class hierarchy of the classes of *MATLAB*® interface.

3.5.2 LightSource

A light source typically consists of a color, a position and an intensity. We model the color and intensity in one property. Thus, the property *Color* of the class *LightSource* denotes the color intensity. Figure 3.5 illustrates the class *LightSource* as an UML class diagram.

3.5.3 Volume

The class *Volume* has only *Data* as property (see Figure 3.6).
Moreover, as described in Section 3.5.1, *Volume* inherits *handle*. Thus, the memory management described in Section 3.3 can be easily realized. But

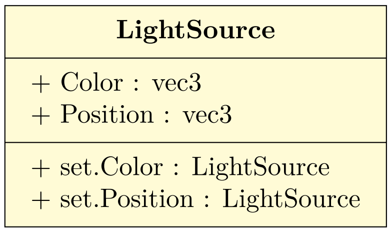


Figure 3.5: Class diagram of class *LightSource*.

be careful of the behavior of a *handle* subclass. If we built a volume with some data and then change the value of data outside the object, the data inside the object also will change. The following code example shows this behavior.

```
data=1;
vol1=Volume( data );
vol2=Volume( data );

data=2; % assign new value
display( vol1.Data ); % will now display 2
display( vol2.Data ); % will also display 2
```

The reason for this behavior is that *vol1.Data* as well as *vol2.Data* is a pointer to *data*. If we construct multiple volumes with exactly the same data, they share this pointer. The next example code also shows a behavior of which we have to pay attention.

```
vol1=Volume( 1 ); % create volume
vol2=vol1;

vol2.Data=2; % assign new value
display( vol2.Data ); % will also display 2
```

Since *vol1* is a *handle*, the assignment *vol2=vol1* does not create a deep-copy. Consequently, the data of *vol2* point to the data of *vol1*.
Additionally, we implemented a resize method, that resizes the data to a given size. One can either give an *vec2* for a *array2* or an *vec3* if *Data* is a volumetric data set (*array3*).

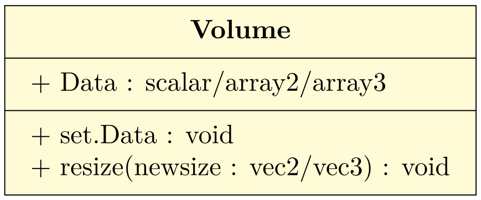

Volume
+ Data : scalar/array2/array3
+ set.Data : void
+ resize(newsize : vec2/vec3) : void

Figure 3.6: Class diagram of class *Volume*.

3.5.4 VolumeRender

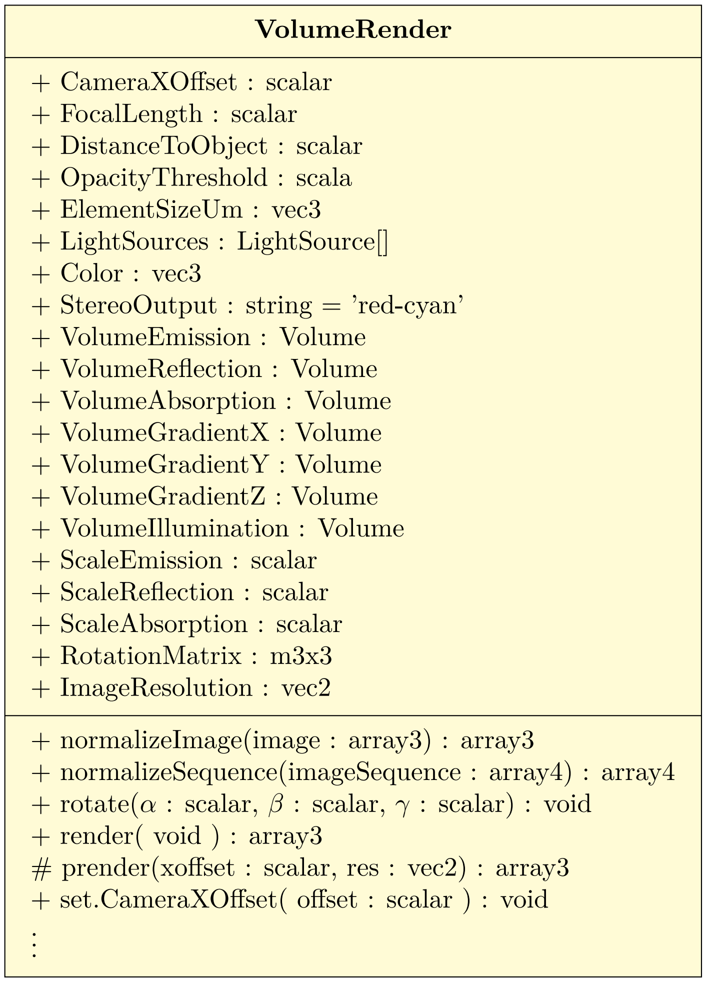
Figure 3.7 illustrates the class diagram of *VolumeRender*. As this class is the main class of our interface it provides a lot of properties and thusly configuration opportunities.
In the following sections we describe the features *Volume Renderer* provides.

Rendering

Our interface makes it easy to render a scene. It suffices to assign a emission and absorption. The reflection volume is only required if an illumination volume and one ore more light sources are defined. The default value of *ElementSizeUm* is 1 for all dimensions and the default rotation matrix is the identity matrix. Thus a minmal rendering code could be:

```
volume = Volume(SomeData); % create volume object
renderer = VolumeRenderer; % create renderer object
renderer.VolumeEmission = volume;
renderer.VolumeAbsorption = volume;
render.ImageResolution=someResolution;
image = renderer.render(); % call render
```

In this example the camera distance to the object is 0. Thus, the camera is inside the volume. *DistanceToObject* defines the distance between camera and object. Additionally, one can define a focal length.
The object in the example code is white as this is the default color. It is possible to change *Color* by a RGB row vector.
In Section 2.2.3 we introduced that the renderer uses front-to-back render-



```
+ CameraXOffset : scalar
+ FocalLength : scalar
+ DistanceToObject : scalar
+ OpacityThreshold : scala
+ ElementSizeUm : vec3
+ LightSources : LightSource[]
+ Color : vec3
+ StereoOutput : string = 'red-cyan'
+ VolumeEmission : Volume
+ VolumeReflection : Volume
+ VolumeAbsorption : Volume
+ VolumeGradientX : Volume
+ VolumeGradientY : Volume
+ VolumeGradientZ : Volume
+ VolumeIllumination : Volume
+ ScaleEmission : scalar
+ ScaleReflection : scalar
+ ScaleAbsorption : scalar
+ RotationMatrix : m3x3
+ ImageResolution : vec2
```

```
+ normalizeImage(image : array3) : array3
+ normalizeSequence(imageSequence : array4) : array4
+ rotate(α : scalar, β : scalar, γ : scalar) : void
+ render( void ) : array3
# prender(xoffset : scalar, res : vec2) : array3
+ set.CameraXOffset( offset : scalar ) : void
⋮
```

Figure 3.7: Class diagram of class *VolumeRender*.

ing as this method has the advantage that a ray can be stopped when the accumulated alpha value does exceed a given threshold. This threshold is given by *OpacityThreshold*. Its default value is 0.95.
For easy rotation, *VolumeRender* provides a method *rotate* the three angles around the axis as parameters. γ denotes the rotation around the level axis, β around the row axis and α around the column axis. This method rotates the current rotation matrix respectively. The angles are given in degree. Alternatively, one can also change the rotation matrix manually.
With this function there is the possibility to rotate the camera around the volume with a few lines of code. Thus, one can easily create a movie.
Our renderer only renders one object per rendering pass. By a clever manipulation of the emission, absorption and reflection volumes, it is possible to render multiple objects in seperate rendering passes and combine the rendered images so that the outcome looks like one rendering pass of multiple objects. Such complex examples like creating a movie of multiple objects are shown in Appendix A.

Notice: Make sure that the data of absorption are $>= 0$!

Rendering Stereo

As in some use cases it is nice to get a 3D impression of the image our renderer offers the opportunity to render off-axis stereo images. Therefore, the renderer needs to allow to setup a camera x-axis offset, the distance between the two stereo cameras. The theoretical background is described in Section 2.3.1.
Knowing the camera x-offset, the focal length and the image resolution we can render two images. As explained in Section 2.3.1, to obtain an image with a given resolution w x h we first extend the image size, then render the two images and finally trim these images again. After the trimming we can easily combine the two images to one stereo anaglyph image. All these operations are done by our *MATLAB*$^{®}$ interface. *VolumeRender* provides the opportunity to assign *CameraXOffset* that defines the baseline of the two stereo cameras.

The default value of *StereoOutput* is set to 'red-cyan'. Thus a 2D anaglyph image is returened by the render call. If one requires both stereo images separately, *StereoOutput* must be set to 'left-right-horizontal'. Then the returned value is of type *[array3, array3]*.

Notice: If *CameraXOffset* of the *MATLAB*® render object is different to 0, the render-call will return automatically a stereo anaglyph image. *CameraXOffset* is set in voxel.
Set *StereoOutput* to 'left-right-horizontal' if you need the left and right view separately. Then the returned value is of type *[array3, array3]*.

Normalizing

The images the renderer computes are not in the interval of 1 and 256. Thus, we provide the method *normalizeImage* that does this job. Moreover we also provide the method *normalizeSequence* that normalizes an image sequence e.g. of a movie.

Notice: Applying *normalizeImage* to each frame of a movie sequence will most likely result in flickering. Hence, use *normalizeSequence* with the whole image sequence as input parameter.

Chapter 4

Conclusion and Future Work

4.1 Conclusion

With *Volume Renderer* we provide an offline renderer with a lot of functionalities. Due to the GPU acceleration that leads to high parallelization and the simplified render equation the runtime is kept low. Since the GPU memory size is very limited, the memory management we designed provides the opportunity to render big volumes.
Moreover, the render equation works with absorption and emission. Thus, it is possible to setup the renderer that it renders multiple volumes in separated renderings in a way that the combination of these rendered images look like rendered in one scene at the same time.
Our illumination model interface allows to use several illumination equations. Several light sources with several light color intensities can be defined. This increases the realism of the scene and thus the spatial impression of the scene.
The *MATLAB*® interface is easy to use. Hence, one can easily construct e.g. a movie scene. What is more, our renderer can render stereo images and combine them to an anaglyph. Since the rendered images are float values, the interface also provides normalization methods for both, single images and whole images sequences. Finally we optimize the runtime of the *MATLAB*® code. Therefore, and in order to be able to realize the memory

management we make use of the *handle* class of *MATLAB*® , that allows us to use call by reference instead of call by value.
In conclusion we offer a new convenient *MATLAB*® toolkit for volume rendering.

4.2 Future Work

There is still some work that could be improved. To increase the realism and spatial impression of the scene the renderer could be enabled to use some shadowing techniques as described in [5].
Our renderer just uses linear interpolation. To get more accurate interpolation values bicubic interpolation could be added as an additional option.

Appendix A

Matlab Examples

In the following examples we do not take into account how data is read in. As the data are often h5 files we show one example of how a read in could look like.

```
filename ='/path/to/data.h5';
dataset = '/someChannel';
data = hdf5read( filename, dataset); % read h5 file

render = VolumeRender(); % create renderer object
render.ElementSizeUm = ...
  hdf5read( filename, strcat(dataset,'/element_size_um'));
```

In the following sections we assume, that data are already read in. Thus, we omit the read in code.

A.1 Movie of a rotating Object

In this example we rotate an object around the x-axis.

```
emission = Volume(data);

render = VolumeRender();
render.VolumeEmission = emission;
render.VolumeAbsorption = emission; % min value >= 0

render.ImageResolution = ...
  [size(emission.Data,2), size(emission.Data,1)];

render.ElementSizeUm = elementSizeUm;
render.DistanceToObject = 10;
render.FocalLength = 3.0; % set come focal length
render.Color = [1,0,0]; % set object color to red

% initialize rendered_image
rendered_image = ...
  zeros([size(emission.Data,2),size(emission.Data,1), 3, 360]);

% #frames
nSteps=50;
angle = 360/nSteps;
for i=1:nSteps
    render.rotate(0,0,angle); % rotate around x-axis
    rendered_image(:,:,:,i) = render.render();
end

% normalize movie frames
normalizedImages = ...
  VolumeRender.normalizeSequence(rendered_image);

% create and show movie
mov = immovie(normalizedImages);
movie(mov);
```

A.2 Movie of a rotating Light Source

In this example we set up three different light sources. One of these light sources (blue light) rotates around the y-axis.

```
emission = Volume(data);

render = VolumeRender();
render.VolumeEmission = emission;
render.VolumeAbsorption = emission; % min value >= 0

render.ImageResolution = ...
  [size(emission.Data,2), size(emission.Data,1)];

render.ElementSizeUm = elementSizeUm;
render.DistanceToObject = 10;
render.FocalLength = 3.0; % set come focal length
render.Color = [1,1,1]; % set object color to white

% build illumination model (Henyey-Greenstein)
render.VolumeIllumination=Volume(HG(64));

% setup 3 different light sources (pos/color)
render.LightSources = [LightSource([0,0,1], [0.5,0,0]), ...
                       LightSource([0,0,-1], [0,0.5,0]), ...
                       LightSource([-1,0,0], [0,0,0.5])];
% #frames
nSteps=50;
angle = 360/nSteps;
for i=1:nSteps
    % rotation matrix
    RotationY = [cosd(beta),0,sind(beta);
                 0,1,0;
                 -sin(beta),0,cos(beta)];

    % rotate light source
    render.LightSources(3).Position = ...
      (RotationY*render.LightSources(3).Position');
    rendered_image(:,:,:,i) = render.render();
end

```

```
% normalize movie frames
normalizedImages = ...
  VolumeRender.normalizeSequence(rendered_image);

% create and show movie
mov = immovie(normalizedImages);
movie(mov);
```

A.3 Rendering of two Objects

This example shows how to render two objects. This example can easily be adapted to more objects. In the following example we render two volumes by additively combining their absorption. The depth impression of both objects.

```
emission1 = Volume(data1);
emission2 = Volume(data2);

render = VolumeRender();
render.ElementSizeUm = elementSizeUm;
render.DistanceToObject = 10;
render.FocalLength = 3.0; % set come focal length

render.ImageResolution = ...
  [size(emission1.Data,2), size(emission1.Data,1)];

% combine absorption
render.VolumeAbsorption = Volume(data1 + data2); % min value >=
    0

% setup and render first volume
render.VolumeEmission = emission1;
render.Color = [1,0,0]; % set object color to red
rendered_image = render.render();

% setup and render second volume
render.VolumeEmission = emission2;
render.Color = [1,1,1]; % set object color to white
rendered_image = render.render();
```

```
% combine the two rendered images
rendered_image = rendered_image + render.render();

% show normalized image
imshow(VolumeRender.normalizedImage(rendered_image));
```

Bibliography

[1] James F. Blinn. Models of light reflection for computer synthesized pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977.

[2] Paul Bourke. Calculating Stereo Pairs. `http://paulbourke.net/miscellaneous/stereographics/stereorender`, July 1999. [Online; accessed 23-June-2012].

[3] L.G. Henyey and J.L. Greenstein. Diffuse radiation in the galaxy. *The Astrophysical Journal*, 93:70–83, 1941.

[4] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *GPU GEMS Chapter 39, Volume Rendering Techniques.* Addison Wesley, 5th edition, September 2007.

[5] Milan Ikits, Joe Kniss, Aaron Lefohn, and Charles Hansen. *GPU GEMS Chapter 39, Volume Rendering Techniques.* Addison Wesley, 5th edition, September 2007.

[6] Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics and Applications*, 8:29–37, 1988.

[7] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21(4):163–169, August 1987.

[8] Nelson L. Max. Optical Models for Direct Volume Rendering. *IEEE Trans. Vis. Comput. Graph.*, 1(2):99–108, 1995.

[9] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2011. Version 4.0.

[10] T. Porter and T. Duff. Compositing digital images. *ACM SIGGRAPH Computer Graphics*, 18(3):253–259, 1984.

[11] Stefan Röttger, Martin Kraus, and Thomas Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection, 2000.

[12] C. E. Shannon. Communication in the presence of noise. In *Proceedings of the Institute of Radio Engineers (IRE)*, volume 37, pages 10–21, 1949.

[13] Inc. The MathWorks. Object-Oriented Programming. *Object-Oriented Programming*, R2011b, 2011.

[14] Wikipedia. Volume ray casting — wikipedia, the free encyclopedia. `http://en.wikipedia.org/w/index.php?title=Volume_ray_casting&oldid=498233905`, 2012. [Online; accessed 8-July-2012].

[15] Amy Williams, Steve Barrus, R. Keith Morley, and Peter Shirley. An Efficient and Robust Ray-Box Intersection Algorithm. *journal of graphics, gpu, and game tools*, 10(1):49–54, 2005.