

Visualisierungsinstitut der Universität Stuttgart
University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelor's Thesis Nr. 06

Interactive Volume Visualization with WebGL

Michael Becher

Course of Study:	Computer Science
Examiner:	Prof. Dr. Thomas Ertl
Supervisors:	M. Sc. Grzegorz K. Karch M. Sc. Finian Mwalongo
Commenced:	May 2, 2012
Completed:	November 1, 2012
CR-Classification:	I.3, H.3.5

Abstract

Web-based applications have become increasingly popular in many areas and advances in web-based 3D graphics were made accordingly.

In this context, we present a web based implementation of volume rendering using the relatively new WebGL API for interactive 3D graphics. An overview of the theoretical background of volume rendering as well as of the common approaches for a GPU implementation is given, followed by detailed description of our implementation with WebGL. Afterwards the implementation of advanced features is covered, before a short introduction to X3DOM, as a possible alternative for web based volume visualization, is given.

It is the aim of this work to incorporate both basic and advanced methods of volume rendering and to achieve interactive framerates with WebGL, using the power of client-side graphics hardware. With regard to that, the result of our implementation is discussed by evaluating its performance and by comparing it to an alternative solution. Finally, we draw a conclusion of our work and point out possible future work and improvements.

Contents

1	Introduction	9
2	Related Work	11
3	GPU Accelerated Volume Rendering	13
3.1	Volume Data	13
3.2	Optical Models	13
3.3	Volume Rendering Integral	14
3.3.1	Discrete Volume Rendering Integral	15
3.4	Compositing	15
3.4.1	Back to Front Compositing	16
3.4.2	Front to Back Compositing	16
3.5	Texture Based Volume Rendering	16
3.6	Raycasting	17
3.6.1	Generating and Casting a Ray	18
3.6.2	Sampling and Compositing	19
3.7	Transfer Functions	19
4	Volume Rendering with WebGL	21
4.1	Introduction to WebGL	21
4.2	Creating a WebGL Context	22
4.3	File Loading	23
4.4	Initializing Shaders	23
4.5	Emulating 3D textures	23
4.6	Initializing Textures	28
4.7	Initializing Render Plane	28
4.8	Transfer Function	29
4.9	Display Function	29
4.10	Vertex Shader	29
4.11	Fragment Shader	30
5	Advanced Features	33
5.1	Geometry Rendering	33
5.2	Isosurface Visualization	34
5.3	Lighting	35
5.4	Animation	37

6	X3DOM	39
6.1	Introduction to X3DOM	39
6.2	Volume Rendering with X3DOM	39
7	Evaluation	41
7.1	Stability	41
7.2	Performance	41
7.3	Comparison with X3DOM	42
	7.3.1 Features	42
	7.3.2 Performance	43
8	Conclusion and Future Work	45
9	Acknowledgements	47
A	Appendix	49
	A.1 Zusammenfassung	49
	Bibliography	51

List of Figures

3.1	Ray - Box Intersection	14
3.2	Texture Based Volume Rendering	17
3.3	Raycasting	18
4.1	OpenGL ES 2.0 Pipeline	21
4.2	Structure of the Application	22
4.3	2D Texture Layout	24
4.4	Adjustments for Bilinear Filtering	25
4.5	Index-Offsets	27
4.6	Transfer Function	30
5.1	Geometry Rendering	34
5.2	Iso-Surface	35
5.3	Lighting	36
6.1	X3DOM Screenshot	40

List of Tables

7.1	Benchmark	42
7.2	Benchmark Comparison	43

Listings

6.1	Example of the minimum necessary code for volume rendering with X3DOM.	39
-----	--	----

1 Introduction

The demand for exciting and elaborate, and often professional, web applications has risen to new heights. To keep up with it, in recent years a lot of effort was put forth to increase the 3D graphics capabilities of web applications. Presented solutions usually require the installation of browser add-ons in order to view the 3D content, forcing the user to install a plug-in first. To avoid such complications and make advanced 3D and 2D graphics accessible to a wide audience, in 2009 the Khronos Group started an initiative to develop a standardized, native, low-level API based on the OpenGL ES 2.0 specification, nowadays known as WebGL [Khr09]. Prior to that, experiments to embed an OpenGL 3D context into an HTML5 canvas element were already made by Mozilla [Vuk07] and Opera[Joh07], the latter using a more independent 3D context. A more detailed introduction to WebGL is given in Chapter 4.

With the native support for an OpenGL based API in the browser, the pathway to numerous new web applications has been opened. Because they run in a browser and are using the OpenGL ES 2.0 API, the applications are essentially platform independent and can be used on a wide range of hardware. This includes of course desktop PCs, but also mobile devices such as smartphones and tablets, which lately are outfitted with increasingly fast graphics hardware.

Apart from that, the availability is improved not only by the wide hardware support, but also by the fact that no additional software is required. As long as a browser is installed, applications featuring complex 3D graphics can be used on any device, without the need to install any special, often proprietary, software.

The solutions for web-based graphics content presented prior to WebGL are usually limited to rendering surface geometry (e.g. polygonal meshes). However, especially in some scientific fields of study, three dimensional volume datasets are often used and the implementation of corresponding algorithms proves to be difficult, if possible at all, with limited render options. One of these algorithms is volume rendering, the most popular method to visualize three dimensional datasets. A volume dataset is essentially a three dimensional grid filled with scalar values. A more detailed explanation is given in Chapter 4.1.

Volume rendering has been known, researched and actively used for more than two decades [Lev88]. It is a leading technique for the visualization of scientific data, often with medical background but also several other lines of research, including fluid simulations, meteorology or engineering. Volume rendering is generally divided in two categories: direct volume rendering and indirect volume rendering. The former aims to directly evaluate and display all aspects of a volume, whereas the latter focuses on extracting surface information from a volume to create a suitable surface representation for rendering.

Over time, a variety of algorithms were introduced that improved upon the original idea.

Some examples are texture-based volume rendering [CN94][EKE01], shear-warp factorizations [LL94] or GPU accelerated Raycasting [KW03]. An extensive and detailed overview of volume rendering is given in [HLSR08].

The visualization pipeline describes the steps that lead from a data source to a rendered image. It is usually divided into four stages: First, in the data acquisition stage, raw data is collected from any suitable source like simulations or sensor input. Second, in the filtering stage, the collected raw data is refined. Third, in the mapping stage, the refined data is mapped to the desired graphical primitives. Among others these can include points, surfaces or volumes, as well as attributes like color and transparency. The fourth and last stage is the rendering stage. Here the graphical primitives are rendered to create images. In terms of the visualization pipeline, we are concentrating on rendering and mapping, while data acquisition is completely skipped and filtering is generally omitted as well.

The aim of this work is to develop an interactive volume rendering application, combining the advances made in web based real-time 3D rendering with the well developed techniques for volume rendering. Besides the implementation of the basic volume rendering functionality with raycasting, some advanced features including lighting and iso-surface visualization are to be part of the application. Furthermore we want to compare our implementation to already existing solutions for web-based volume rendering, namely X3DOM or XML3D. However, volume rendering support for XML3D is planned for 2013 and not yet available. This leaves only X3DOM for comparison.

The thesis is structured as follows. First, in Chapter 2, an overview of previous work in the field of volume rendering and web based 3D graphics is given. Next, in Chapter 3, we review the theoretical principles of volume rendering and discuss different, suitable approaches for a GPU volume rendering algorithm.

Afterwards, in Chapter 4, our WebGL implementation is explained in detail, followed by a description of some advanced features in chapter 5. In Chapter 6, we give a short introduction to X3DOM [BEJZ09]. After an evaluation of our application and comparison with X3DOM in chapter 7, we hint at some possible future improvements in chapter 8.

2 Related Work

Originally, the typical method for GPU accelerated volume rendering is an object-order approach, first introduced in 1993 by Cullip and Neumann [CN94]. The basic idea is to cut slices from the volume and render these using proxy geometry. A short overview of this method is given in Chapter 3.

Another fast algorithm for volume rendering is shear-warp factorization, presented by Lacroute and Levoy [LL94]. A volume is transformed into a sheared space, where a parallel projection creates an intermediate image, which is then warped to produce the final image.

Krüger *et al.* improved GPU-based volume rendering by using the programmable graphics pipeline of modern graphics cards for a raycasting algorithm, that allowed the integration of acceleration techniques [KW03].

Volume rendering has not only been researched with regard to improved algorithms. A model for distributed volume rendering was presented by Magallón *et al.* [MHE01]. With a high-speed network, a cluster consisting of mainstream graphics hardware can be created, in order to be able to render larger datasets.

An interactive volume rendering application for mobile devices was presented by Moser and Weiskopf [MW08]. Their implementation used the OpenGL ES 1.0 API and therefore similar functionality restrictions, as imposed by WebGL, were addressed.

An overview of existing techniques for the 3D rendering in the web can be found in an article by Behr *et al.* [BEJZ09]. In the same article, Behr *et al.* introduced an integration for the X3D standard into the HTML5 Document Object Model called X3DOM. It offers a declarative approach to web-based 3D graphics.

Since the introduction of WebGL, numerous works featuring native 3D graphics support for browsers were published. Among these, an article by J. Congote *et al.* is of particular interest in regard to the subject of this thesis [CSK⁺11]. They presented a WebGL implementation of a volume renderer, focusing on the visualization of medical and meteorological data. Even though there are some differences in the render algorithms and focus of the work, the initial idea of the article and the approach of the implementation are very similar. The features they mention as possible future works are at least partially implemented as a part of this thesis, e.g. lighting and surface rendering.

Jacinto *et al.* published another work that uses WebGL for the visualization of medical data [JKD⁺12]. Instead of direct volume rendering, they used indirect volume rendering and put more emphasis on the interface of their application.

WebGL has also been used by P. Jiarathanakul to implement a real-time ray marching algorithm for distance fields [Jia12]. A scene is rendered by stepping through it, at each step checking

2 Related Work

the distance to the nearest surface and deciding the step size based on that, till a surface is reached.

3 GPU Accelerated Volume Rendering

Next to traditional surface rendering, which relies on some kind of surface representation like polygonal meshes, volume rendering is one of the most prominent rendering techniques in the field of computer graphics. This is aided by the fact, that for some time now, hardware accelerated volume rendering allows interactive framerates.

In this chapter, we solely discuss direct volume rendering. In the first part, a brief explanation of the theoretical background, namely optical models and the volume rendering integral, is given, followed by two volume rendering techniques suited for a GPU implementation in the second part.

3.1 Volume Data

In our practical context of volume rendering, a volume is defined as a discrete, three dimensional set of scalar values. It can be thought of as the result of sampling a continuous three dimensional function. A single value of this set is called a voxel, and is considered as the smallest unit of a uniform, three dimensional grid [HLSR08].

Volume datasets are mostly generated by medical scans and scientific simulations. However, since the acquisition of such data sets is not a part of this thesis, we will not pursue this topic any further.

3.2 Optical Models

Almost all volume rendering algorithms are based on an underlying physical model, that describes the interaction of light with the particles of a medium. The properties that contribute to the radiance of a particle are absorption, emission and scattering.

Several optical models with varying degrees of realism are known [Max95]. The three that are of relevance to this study, are briefly described below.

Absorption Only

This model assumes the particles to be perfectly black and cold [Max95]. Light that passes through a medium consisting of such particles loses some of its radiative energy, depending on the absorption coefficient. No light is emitted by a particle, and no scattering takes place.

Emission Only

The emission-only model neglects absorption and scattering. The particles only emit light.

Absorption and Emission

The effects of both absorption and emission are taken into account with this model, but scattering is omitted. This model is closer to reality as the previous two, since a medium usually both partially absorbs incoming light but also emits light again.

3.3 Volume Rendering Integral

We formulate the volume rendering integral for the absorption-emission model, the most frequently used one for direct volume rendering. Note that for the continuous volume rendering integral, we are falling back to a non-discrete view of a volume for now.

For a ray r , that represents the way that light travels from a light source through the volume to the eye, we parametrise the position $r(x)$ on the ray with the distance x from the eye. The scalar value obtained from the volume at a position $r(x)$ is denoted by $v(r(x))$. Since we are using an absorption and emission model, we need to define the emission $c(v(r(x)))$ and absorption coefficient $\kappa(v(r(x)))$. For the sake of simplicity, in the following equations both are written as functions with distance x from the viewer as parameter.

As Figure 3.1 illustrates, the ray enters the volume at s_0 and exits it in the direction of the viewer at s_1 . To model the emission of the volume, we integrate over the distance between s_0 and s_1 and add it to the initial intensity $I(s_0)$ multiplied by the transparency of the volume along r . We model the transparency T between two points x_0 and x_1 with

$$(3.1) \quad T = e^{-\tau(x_0, x_1)}$$
$$\tau(x_0, x_1) = \int_{x_0}^{x_1} \kappa(x) dx$$

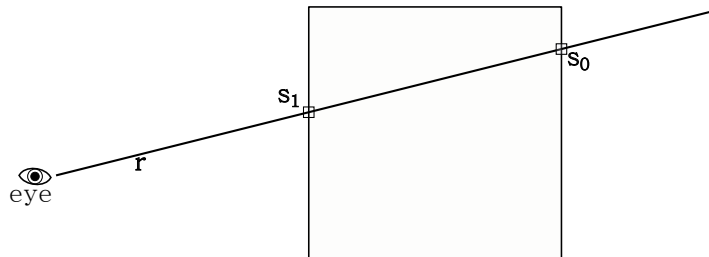


Figure 3.1: Top-down cross section of a Ray-Box intersection.

The integral over the absorption coefficient is called the optical depth. The exponential function for the transparency is deduced by solving the differential equation given by the emission-absorption model [Max95]. Finally, Equation 3.2 is the volume rendering integral.

$$(3.2) \quad I(s_1) = I(s_0) \cdot e^{-\tau(s_0, s_1)} + \int_{s_0}^{s_1} c(x) \cdot e^{-\tau(x, s_1)} dx \quad [\text{Max95}]$$

3.3.1 Discrete Volume Rendering Integral

As already stated in Section 3.1, in practical applications the volume is a discrete set of values. Therefore, the volume rendering integral is also evaluated using a discrete, numerical approximation. The discretisation is achieved by approximation of the integrals with Riemann sums [HLSR08].

For the optical depth τ in Equation 3.1 we obtain the discrete version

$$(3.3) \quad \tau(x_0, x_1) \approx \sum_{i=0}^{\lfloor (x_1 - x_0) / \Delta x \rfloor} \kappa(x_0 + i \cdot \Delta x) \cdot \Delta x$$

which leads to the following approximation of the exponential function

$$(3.4) \quad e^{-\tau(x_0, x_1)} \approx \prod_{i=0}^{\lfloor (x_1 - x_0) / \Delta x \rfloor} e^{-\kappa(x_0 + i \cdot \Delta x) \cdot \Delta x}$$

Accordingly, the emission in i -th ray segment is approximated by

$$(3.5) \quad C_i = c(s_1 + i \cdot \Delta x) \Delta x$$

Using the approximation for both the emission and absorption, leads to the discrete volume rendering integral:

$$(3.6) \quad I = \sum_{i=0}^{\lfloor (s_0 - s_1) / \Delta x \rfloor} C_i \prod_{j=0}^{i-1} e^{-\kappa(s_1 + j \cdot \Delta x) \cdot \Delta x}$$

3.4 Compositing

Volume rendering algorithms usually evaluate the discrete volume rendering integral by means of an iterative computation. The method shown here is also called alpha blending [HLSR08].

3.4.1 Back to Front Compositing

We introduce the opacity A , which can be defined as

$$A = 1 - T$$

Back-to-front composition is directly derived from the discrete volume rendering integral.

$$(3.7) \quad \begin{aligned} C'_i &= C_i + (1 - A_i)C'_{i+1} \\ A'_i &= A_i + (1 - A_i)A'_{i+1} \end{aligned}$$

C_i is the colour value of the i -th sample and C'_i is the color accumulated up to the i -th sample. Analogue, A_i is the opacity of the i -th sample and A'_i is the accumulated opacity up to the i -th sample.

3.4.2 Front to Back Compositing

Reversing the order of the summation leads to the front-to-back compositing scheme.

$$(3.8) \quad \begin{aligned} C'_i &= C_i(1 - A_{i-1}) + C'_{i-1} \\ A'_i &= A_i(1 - A_{i-1}) + A'_{i-1} \end{aligned}$$

3.5 Texture Based Volume Rendering

One of the most important algorithms for hardware accelerated volume rendering is a texture-based, object-order approach, that has been known since the early nineties [CN94][CCF94] and has since then repeatedly been used and improved [WE98][EKE01]. It is a method that doesn't require modern per fragment operations, but instead relies on rasterization, texturing and blending features, that even older graphics hardware supports.

The general idea is to slice the volume with several planes to create proxy geometry, which is then textured with values from the volume and blended together. The intersection of a plane with the volumes bounding box results in a polygon, that is used as proxy geometry.

Slicing is done either bounding box axis aligned, or view plane aligned. Depending on the slicing method the slices are textured using 2D textures or a 3D texture, meaning either bilinear or trilinear interpolation. The composition of the resulting, textured stack of slices is done in back-to-front order. Figure 3.2 illustrates the three major steps for both bounding box and view plane aligned slicing.

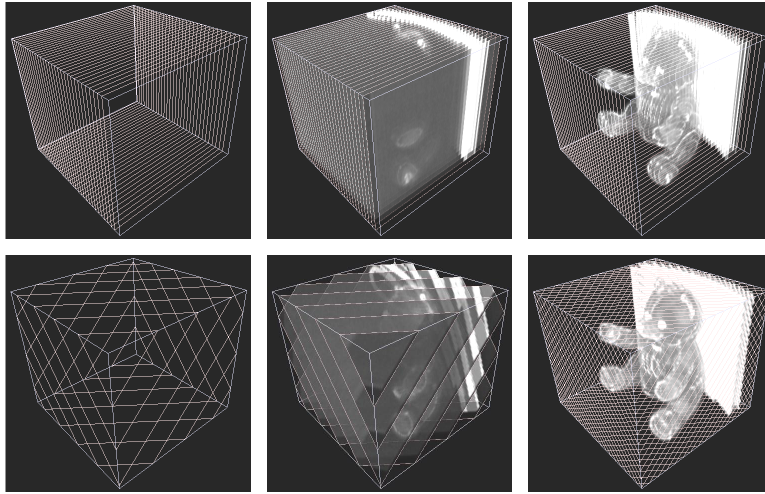


Figure 3.2: *Upper left:* Bounding box aligned slicing. *Upper middle:* The slices are textured using bilinear filtering. *Upper right:* Composition of the slices. *Lower left:* View plane aligned slicing. *Lower middle:* The slices are textured using trilinear filtering. *Lower right:* Composition of the slices. [Ert11].

3.6 Raycasting

Raycasting is an image-order algorithm that renders a frame by casting rays into the scene. A single ray is generated for every pixel on the screen, in order to retrieve the pixel's color from the scene. Unlike Raytracing, which shares some similarities, Raycasting only uses rays that originate at the viewer location, usually referred to as primary rays. The general idea is illustrated in Figure 3.3.

After their generation, the rays are tested against all scene objects for intersections. In volume rendering, the volume's bounding box is used as primitive for the intersection test. If a ray hits a bounding box, the volume is visible at the corresponding screen pixel, otherwise the screen pixel is set to the background color. The intersection points (when intersecting a straight line with a box, there are always two intersections) are sorted by distance from the origin to identify the entry and exit point. The screen pixel's color is then retrieved by stepping through the volume, starting at the entry point, along the ray. To get the final color, the sample points are composited in front to back order.

Since the rays are independent from each other and the operations performed for all rays are nearly identical, Raycasting is well suited for parallelization on the GPU. Casting a ray, performing the intersection test and stepping through the volume if necessary, are implemented as per fragment operations.

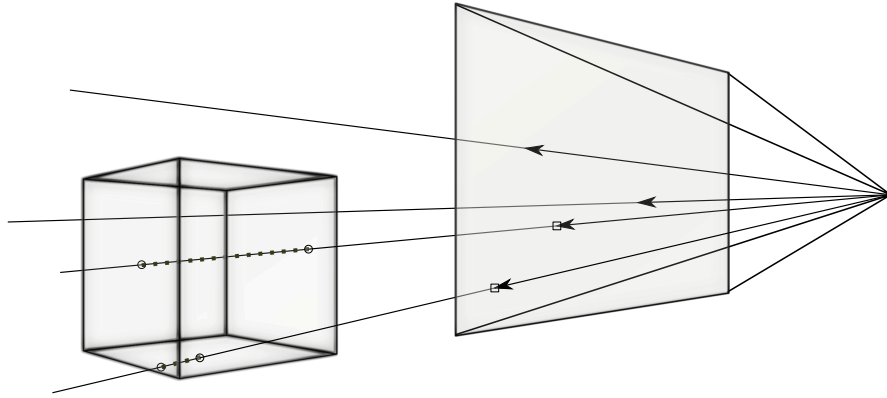


Figure 3.3: Illustration of the general idea of Raycasting.

3.6.1 Generating and Casting a Ray

There are different approaches at creating the fragments and generating the corresponding rays.

The first is a multi-pass approach [KW03], where the actual bounding box of the volume is rendered to determine the entry and exit points of a ray, that originates at the viewer's location and hits the bounding box. The faces of the box are color coded to reflect the coordinates of a surface point inside texture space. We refer to texture space as the three dimensional space where the bounding box dimensions are normalized in all three directions, meaning the bounding box expands from $(0.0, 0.0, 0.0)^T$ to $(1.0, 1.0, 1.0)^T$ despite what the actual dimension of the volume dataset is.

In the first renderpass the frontfaces of the bounding box are rendered to a texture. Due to the color coding, the color of a texel is equal to the coordinates of the entry point in texture space. The next renderpass starts with rendering the backfaces of the bounding box, thus obtaining the exit points for each fragment. Together with the output of the first renderpass, the ray direction is now calculated by subtracting the entry points from the exit point. The intersection of the ray with the bounding box is done implicitly by only rendering the bounding box in the first place. Hence, no fragments are created where a viewing ray does not hit the bounding box.

The approach can be modified, so that only a single render pass is required. Depending on whether or not the viewer location is inside or outside the bounding box, either the backfaces or frontfaces are rendered. The ray direction is then obtained by subtracting the camera position, which is transformed to texture space for this, from the entry or exit point. This variation is slightly superior, since it allows for camera positions inside the volume, but requires either an expensive if conditional statement in the fragment shader, or separate shader programs to decide the camera location.

Both variations are dropped in favour of a second, more generalised approach to ray generation. Instead of using the graphics pipeline to create only the relevant fragments, we want to create

a fragment for each screen pixel and then test the bounding box against all resulting rays. This is achieved by rendering a simple, screen filling plane.

The ray direction is then calculated by subtracting the camera position from the fragment position. If done in view space, this is especially easy as the viewer location is always the origin. Using the obtained ray direction as well as the ray origin, the ray and bounding box are now tested for intersections.

3.6.2 Sampling and Compositing

An important part of a Raycast volume renderer addresses how the rays are used to acquire the color and opacity information of a screen pixel from the volume. As mentioned above, this is done by stepping through the volume.

Essentially, this means that the volume is sampled at equidistant points along the ray. We explained earlier, that the volume's voxels form a three dimensional uniform grid. Sample points are rarely positioned at a voxel's center and therefore interpolation schemes are used to obtain the value at a sampling point.

Starting at the entry point, we gradually move in ray direction until we pass the exit point. During this process the color and opacity, obtained from the sample points via a transfer function (see Section 3.7), are accumulated using front to back compositing. Should the opacity pass a threshold of 1.0 the sampling the volume is stopped, because all sample point further ahead are occluded by the thus far accumulated sample points anyway. This simple speed up method is called early ray termination.

Once the last sample point is accounted for, the accumulated color and opacity is assigned to the screen pixel, that corresponds to the fragment.

3.7 Transfer Functions

The transfer function is essentially a transformation from the space \mathbb{R}^1 to the space \mathbb{R}^4 . The domain in both spaces is limited to the range $[0, 1]$, due to normalization of the values from the volume dataset.

It is used to assign a color and opacity to each sample point, which only holds a single, scalar value. While assigning an opacity is indispensable for the employed volume rendering technique, assigning a color, instead of using a grey-scale value, is a very important aspect of visualizing scientific data, as it helps to set apart certain areas. In less scientific applications (e.g. video games etc.), color remains an important aspect, simply because of its vital role in visual perception.

4 Volume Rendering with WebGL

4.1 Introduction to WebGL

In March 2011 the Khronos Group released version 1.0 of the WebGL specification [Khr11], as the first, finalized result of their initiative. WebGL offers client-side 3D rendering functionality for web applications, based on the OpenGL ES 2.0 API. For a list of the difference between WebGL and OpenGL ES 2.0 see [Khr]. The rendering pipeline is directly taken over from OpenGL ES 2.0 and contains two fully programmable stages, namely the vertex and fragment Shader. It is illustrated in Figure 4.1.

A WebGL application generally needs, apart from the HTML web page it is embedded into, two parts: The JavaScript code and a Shader program.

The JavaScript code controls the application and issues all necessary calls to the OpenGL API. Here, all WebGL objects (e.g. Vertex-Buffer-Objects, textures, Shader programs etc.) are created and administrated. It is executed on the CPU.

Shaders on the other hand are essentially programs written specifically for graphics hardware. They are written in the OpenGL Shading Language (GLSL). The Raycasting algorithm is implemented as a Shader program and executed on the GPU.

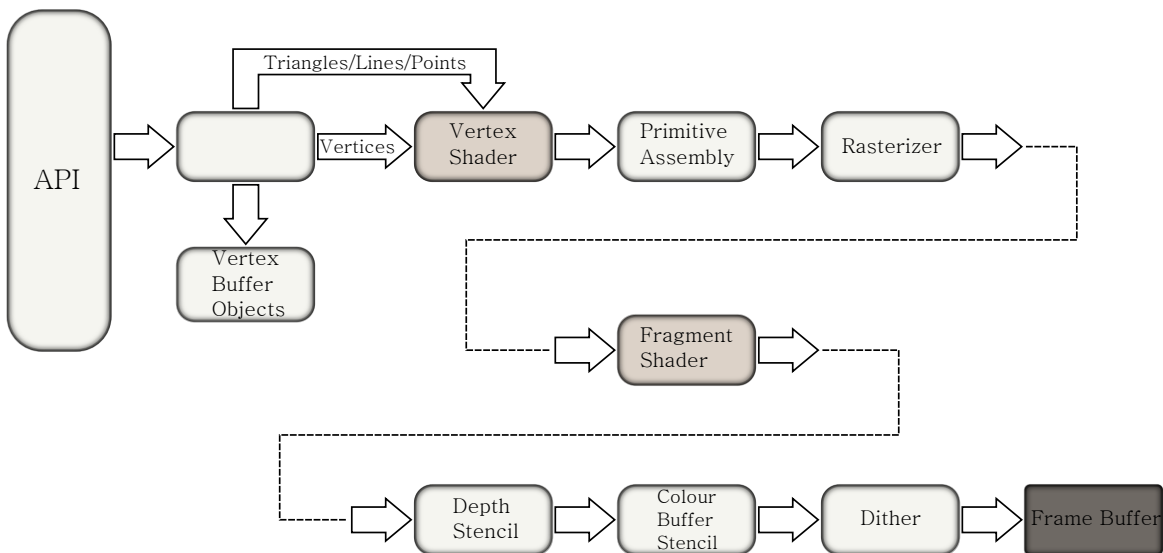


Figure 4.1: OpenGL ES 2.0 Programmable Pipeline.

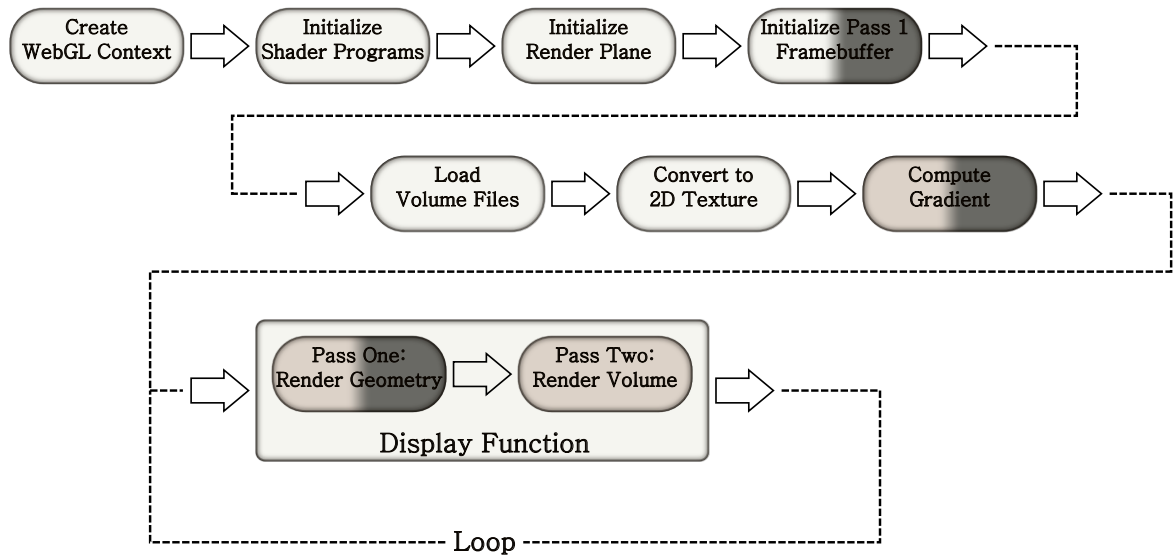


Figure 4.2: Overview of the application flow. Light colored nodes are part of the JavaScript code, the slightly darker nodes are Shader Programs. Nodes with a dark, right half are part of advanced features that are described in Chapter 5.

Writing a WebGL application is in quite a lot of ways very similar to writing a classic OpenGL program using any eligible language (e.g. C++). However, due to the limiting nature of the current WebGL specification, there are a couple of areas where some additional effort is necessary in order to implement the volume rendering techniques presented in the previous chapter.

This Chapter is divided into two areas. The Sections 4.2 to 4.9 cover the JavaScript code of the application, whereas the Sections 4.10 and 4.11 cover the Shader program.

4.2 Creating a WebGL Context

A WebGL context provides access to the WebGL functionality and is required to use any WebGL function e.g. compiling Shader program or creating WebGL Texture objects.

To create a context a canvas element is required. The canvas element was introduced with HTML5 to offer support for freely drawing graphical elements. In combination with WebGL, the canvas is used to draw the rendered images on the web page. Using the `getContext()` canvas function, an *experimental-webgl* context is created.

4.3 File Loading

It is possible to store Shader sourcecode as HTML script elements. It is more convenient though to store the Shader source code in separate files alongside the JavaScript files. Unlike the JavaScript files, which can be directly linked in the HTML code, the Shader files have to be loaded using the XMLHttpRequest API, a popular way to load server-side files.

The volume files are stored locally and have to be uploaded by the user. This is done using the File API, that was introduced with HTML5. It features easy-to-use access to local files and a file reader interface, that makes it possible to read binary data and store it in an array.

4.4 Initializing Shaders

Initializing a Shader program in WebGL is done analogously to *desktop* OpenGL. Unlike more recent OpenGL Versions (3.2+), which also offer the possibility to use geometry and even tessellation Shaders, a WebGL program consists only of a vertex and a fragment Shader. In short, the initialization is compromised of the following steps:

First, new vertex and fragment Shader objects are created. After the Shader source code is added, the Shaders can be compiled and are ready to be attached to a newly created program object. As soon as both Shaders are compiled and attached to the program, the program is linked to the WebGL context and is ready for use.

4.5 Emulating 3D textures

One of the essential parts of any volume renderer is, naturally, the volume data itself. Currently, 3D textures are the most popular and convenient way to handle the volume data. This is based on the earlier explained view of a volume as a uniform, three dimensional grid.

Unfortunately, there currently is no native support for 3D textures available in the WebGL Specification. A workaround has to be found, using the existing feature set in the best way possible.

A known solution for this issue is to emulate the 3D texture using the available 2D texture support [Khr]. This approach suggests itself, since one way to interpret a 3D texture is as a number of 2D textures. These single 2D textures, which together add up to the complete 3D texture or volume, are henceforth referred to as slices. The aim of this method is to find an optimal way to copy and arrange all slices of a given volume into a single 2D texture, often referred to as a texture atlas. Furthermore each slice should not be split into smaller parts, but remain as whole within the 2D texture space. Figure 4.3 illustrates the general idea of how the slices are arranged in a 2D texture.

Texture Resolution

A naive algorithm could simply copy the data of the 3D texture into a 2D texture with the same width as a single slice and the height of a slice multiplied by the number of slices as its height. Obviously this would not produce an optimal result, especially when considering the limited maximum texture resolution of graphic card hardware. Instead, a more optimal texture resolution can be found by trying to meet two conditions.

First, the width m multiplied by the height n of the texture has to be either equal or greater than the volume's width x multiplied with it's height y and depth z , so that there is a texel in the texture for every voxel of the volume.

Second, the width and height of the texture should be approximately the same, so that a maximum amount of image data can be fitted into a single texture file. Both conditions are expressed by Equations 4.1.

On a side note, the width and height of the 2D texture should both be a multiple of a slice's width and height respectively. Because slices are to remain as a whole, this simplifies the process of copying the data into the texture buffer, as well as accessing values from the texture later on.

$$(4.1) \quad \begin{aligned} m \cdot n &\geq w \cdot h \cdot d, \\ m &= a \cdot w \approx b \cdot h = n, \end{aligned}$$

The dimensions m, n, w, h and d are specified in the number of texels and voxels. The variables a and b are the number of slices in X and Y direction respectively. All variables are elements

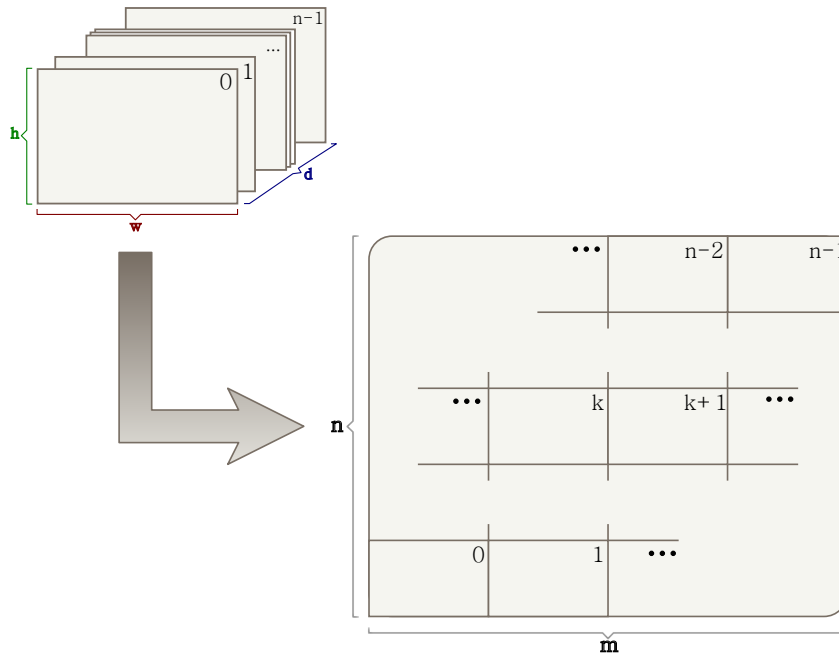


Figure 4.3: Illustrates how the slices of a volume are arranged inside a single 2D texture.

of N .

Using the Equations 4.1 a near minimum number of slice in x and y direction can be determined. This is expressed in the following Equations 4.2.

$$(4.2) \quad \begin{aligned} a &= \lceil \sqrt{\frac{h \cdot d}{w}} \rceil \\ b &= \lceil \frac{w \cdot a}{h} \rceil \end{aligned}$$

With a and b being known values, calculating the final texture width and height becomes easy.

Bilinear Filtering

WebGL features integrated bilinear filtering for 2D textures. Because each slice stays as a whole when copied into a 2D texture, we are able to benefit from this feature. However, there now are multiple areas where the boundary texels of one slice are located directly next to those of another slice. When accessing texels from those areas, binary filtering would interpolate between values from texels, that originally weren't neighbours. This can lead to clearly visible artefacts during rendering, which could be described as part of one of the volume's boundary surface 'bleeding' into another.

To avoid this effect, each slice is expanded by a one texel wide border. This results in the new width of a slice $w' = w + 2$ and the new height $h' = h + 2$, as well as the new width $m' = m + (2 \cdot a)$ and height $n' = n + (2 \cdot b)$ of the 2D texture. The values for the texels of this border are taken from the nearest texel of the original slice. Therefore, whenever a texture access falls into the vicinity of a slice's boundary, the binary filtering will only interpolate between appropriate values.

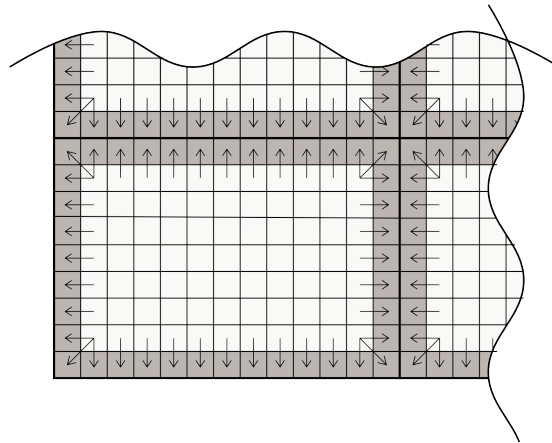


Figure 4.4: Illustration of the lower left slice and its neighbourhood. Dark *pixels* belong to the added border. Arrows indicate which values are used to fill the border pixels

Data Format

Before implementing an algorithm that meets all of the above listed requirements, the internal data representation of any texture, be it 3D, 2D or even 1D, has to be taken into consideration, since this is where the actual copying and rearranging of voxel to texel data takes place.

In our case, the volume data is compromised of a single float value per voxel. Even though the data represents a 3D texture, these values are stored slice by slice in a simple, one dimensional float array q , with each slice being stored in a row-major order. Similarly the texels of a 2D texture are simply stored in row-major order in a one dimensional array. Basically, all that needs to be done is to copy the data from the original input array q into a new array p , while making sure that the position inside p will result in the desired position within 2D texture space. Depending on the number of slices that are placed horizontally next to each other in 2D texture space, a row of the resulting 2D texture is made up from single rows of different slices. Because of the row-major order storage of 2D textures, texels from a single slice are no longer stored in a single connected block.

To copy the data of a voxel with given coordinates $x, y, z \in \mathbb{N}$, we need to know its index in q and p . The position inside the one dimensional array p is calculated by adding up a couple of offset values, that are defined in Equation 4.3. Both s_{offset} and t_{offset} are offset values that are necessary to reach the starting point of the slice, that the current texel belongs to. To reach the correct row inside the slice r_{offset} is needed and by adding x the position of the texel within the array p is finally found. Figure 4.5 illustrates where the texels, that the offsets are pointing at in the 1D array, are located in the 2D texture and how adding them up leads to the correct position.

On the other hand, finding the index within the array q is a little bit more straightforward. To reach the beginning of a slice within q , the voxels of all slices in front of it have to be skipped. We know that a slice contains $w \cdot h$ voxels, so we simply multiply that with the number of the slice z : $z \cdot (w \cdot h)$. To reach the correct row inside the slice, we again skip over the voxels of the rows in front of it: $y \cdot w$. Finally we can add the x value to obtain the final index.

The complete process is summed up in Equation 4.4. Variables w', h' as well as w, h and d are taken over from the two previous sections. It is important to note, that the equations hold for all x, y and z within the given domain.

$$\begin{aligned} s_{\text{offset}} &= w \cdot (z \bmod a) \\ (4.3) \quad t_{\text{offset}} &= h \cdot w \left\lfloor \frac{z}{a} \right\rfloor \\ r_{\text{offset}} &= m \cdot y \end{aligned}$$

(4.4)

$$\begin{aligned} P_{(t_{\text{offset}} + s_{\text{offset}} + r_{\text{offset}} + x)} &= q_{(z \cdot (w \cdot h) + (y \cdot w) + (x))} \\ x, y, z \in \mathbb{N}, 0 \leq x < w, 0 \leq y < h, 0 \leq z < d \end{aligned}$$

If we include the boundary around each slice, the equation gets more complicated. To correctly copy values for the boundary texels from the original array q , we have to handle several

different cases. This version, that is also used in the actual implementation, is shown in the Equations 4.6 and 4.5.

$$\begin{aligned}
 s_{\text{offset}} &= w' \cdot (z \bmod a) \\
 (4.5) \quad t_{\text{offset}} &= h' \cdot w' \lfloor \frac{z}{a} \rfloor \\
 r_{\text{offset}} &= m' \cdot y
 \end{aligned}$$

$$(4.6) \quad p^{(t_{\text{offset}} + s_{\text{offset}} + r_{\text{offset}} + x)} = \begin{cases} q_{(z \cdot (w \cdot h) + (y)(w) + (x))} & , (x = 0 \wedge y = 0) \\ q_{(z \cdot (w \cdot h) + (y-2)(w) + (x))} & , (x = 0 \wedge y = h' - 1) \\ q_{(z \cdot (w \cdot h) + (y)(w) + (x-2))} & , (x = w' - 1 \wedge y = 0) \\ q_{(z \cdot (w \cdot h) + (y-2)(w) + (x-2))} & , (x = w' - 1 \wedge y = h' - 1) \\ q_{(z \cdot (w \cdot h) + (y-1)(w) + (x-2))} & , (x = w' - 1) \\ q_{(z \cdot (w \cdot h) + (y-1)(w) + (x))} & , (x = 0) \\ q_{(z \cdot (w \cdot h) + (y-2)(w) + (x-1))} & , (y = h' - 1) \\ q_{(z \cdot (w \cdot h) + (y)(w) + (x-1))} & , (y = 0) \\ q_{(z \cdot (w \cdot h) + (y-1)(w) + (x-1))} & , \text{else} \end{cases}$$

$$x, y, z \in \mathbb{N}, 0 \leq x < w', 0 \leq y < h', 0 \leq z < d$$

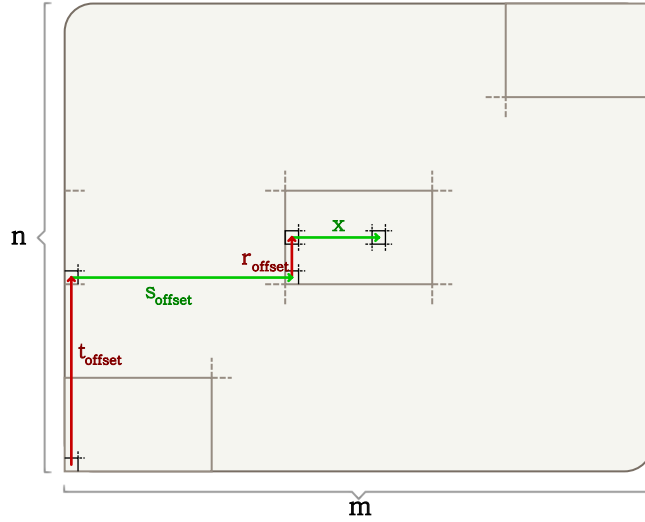


Figure 4.5: Illustrates where the texels, that the respective offsets are pointing at inside the 1D array, are positioned inside 2D texture space.

Multiple Textures

When converting a volume into a 2D texture representation, one last problem needs to be taken care of. That is, the size limitation imposed on 2D textures by the available graphics card hardware. Most modern GPUs support a texture resolution up to the size of 16384×16384 . Even though a volume dataset with a resolution of 512^3 can still be easily fitted onto a single 2D texture of that resolution, any considerable larger datasets require the usage of more than one texture.

Therefore, in case the maximum texture size would be exceeded, the volume has to be split apart. Should the resulting parts exceed the size restriction again, the number of overall parts is increased by one until an acceptable size for each part is reached. Furthermore no slice of the volume is to be split apart, so that each part only contains a set of complete slices. Depending on the overall amount of slices and the number of parts the volume is split into, the different parts do not contain the exact same number of slices.

Afterwards, as soon as the volume is split into parts that can be fitted into a single texture, each part is copied into a 2D texture data buffer and uploaded to the graphic card as a separate texture. The number of textures is again limited by the hardware and it's available texture units. This feature is especially useful on devices with much smaller maximum texture resolution.

4.6 Initializing Textures

In Chapter 3 we stated, that we assume volume datasets to contain only float values. It would be most convenient to simply keep these float values for the actually used textures as well. Therefore, at some point before the first texture is uploaded to the graphics card, preferably directly after the WebGL context is created, the float texture extension needs to be activated to be able to use float valued textures in first place.

After the raw volume data has been loaded and transformed into the required 2D representation, it can finally be send to the graphics card as one or more 2D texture. The following steps are repeated for each data buffer.

A new WebGL Texture object is created and set active. Next, the texture parameters are set. Because we are using non-power-of-two textures, some limitations are in place concerning texture filtering, where only linear interpolation or nearest-texel are legal modes, and texture wrapping, which has to be set to clamp to edge. Now, the data buffer is uploaded with the data format as well as the internal format set to LUMINANCE, as the texture only contains a single float value per texel.

4.7 Initializing Render Plane

As explained in Section 3.6 rendering a screen filling plane is the first step of our GPU Raytracing algorithm. To use it, the plane geometry has to be created and prepared for rendering. First, a new array is created to house the coordinates of the plane's four vertices.

To have the plane fill the complete screen, the coordinates are already stored in normalized device coordinates. Each vertex is placed in a screen corner, resulting in the coordinates of $(-1.0, -1.0, 0.0)$, $(-1.0, 1.0, 0.0)^T$, $(1.0, -1.0, 0.0)^T$ and $(1.0, 1.0, 0.0)^T$ respectively.

WebGL utilizes vertex buffer objects (VBO) to store the vertex data of a geometry object directly on the graphics hardware for non-immediate-mode rendering. So the next step is to generate a new VBO, set it active and finally upload the previously created vertex data.

4.8 Transfer Function

Instead of using a static transfer function, a user-defined transfer function, that can be changed in real time, offers a higher level of interaction and flexibility. The function itself is implemented with a one dimensional RGBA texture. Because 1D textures are not supported in WebGL, it is faked by simply using a 2D texture with a height of one. Using a texture is not only an elegant way to implement the $\mathbb{R}^1 \rightarrow \mathbb{R}^4$ transformation, but also offers linear interpolation to make up for the limited resolution.

All four channels of the texture can be manipulated individually. For that purpose, four HTML5 canvas objects are used, each controlling a channel. The canvas x-axis equals the position inside the texture, while the y-axis equals the value stored in a one channel of a texel. By creating and moving points on the canvas a number of support points for linear interpolation are supplied. A point's x-coordinate is mapped to the corresponding texel, whereas the y-coordinate is normalized to a range between zero and one and used to set the channel value. The texels in-between two support point are set according to the linear interpolation. The texture values, which are stored inside a array, are updated every time a change on a canvas is made and the texture in the graphics card's memory is updated with these new values.

4.9 Display Function

The rendering of a frame is handled by the display function. At the beginning of the function, the framebuffer is prepared for rendering, followed by the computation of the required transformations matrices. Afterwards the transformation matrices are passed on to the Shader program, just like some other relevant informations including texture dimension. Next, all required textures are bound to an active texture units.

Finally, we bind the render plane's VBO and make the draw call to start rendering a frame.

4.10 Vertex Shader

The vertex Shader is a simple pass-through Shader. Since the vertices of the render plane are already given in screen space coordinates, no transformations are necessary at this point.

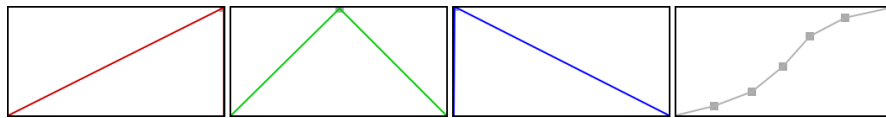
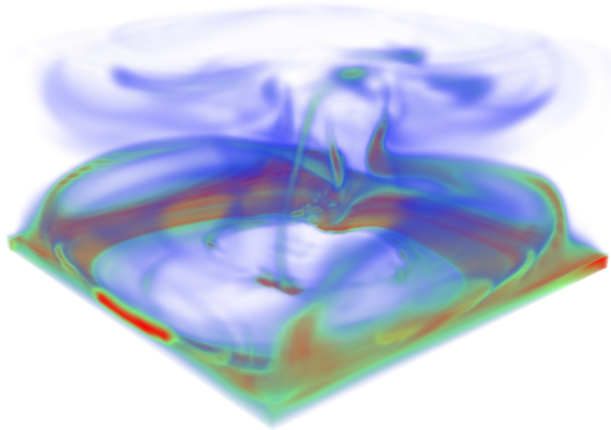


Figure 4.6: Example of a user defined transfer function and the resulting colour of the rendered volume.

4.11 Fragment Shader

Volume Texture Access

Section 4.4 discusses the problems associated with the lack of 3D texture support in WebGL. Accordingly, neither do WebGL Shader programs support the built-in access function for 3D textures, nor would the built-in function work after the rearrangement of the volume data. Therefore, custom functions are needed to access the volume textures. The basic problem is to find for a set of given 3D coordinates the correct texture (in the case of the usage of multiple textures) and for that texture a set of 2D coordinates pointing at the right texel. 3D and 2D Texture coordinates are generally given within a domain of $[0, 1]$.

Identifying the correct texture and determining the 2D texture coordinates for that texture are strictly separated functions. Using the z-coordinate and the number of overall textures the correct texture is easily identified. The z-coordinate is then normalized back to a range of $[0, 1]$ for the function that calculates the 2D access coordinates.

There, we first of all need to identify the slice that the current z-coordinate points at as well as its position inside the 2D texture. To that end the z-coordinate is first transformed to the

index number of the slice and then used for the calculation that determines the x- and y-offset of the slice's lower left corner within 2D texture space.

$$(4.7) \quad \begin{aligned} slice &= \lfloor (z \cdot d) \rfloor \\ x_{\text{offset}} &= \frac{slice \bmod a}{a} \\ y_{\text{offset}} &= \frac{\lfloor \frac{slice}{a} \rfloor}{b} \end{aligned}$$

A texel position within the slice is then calculated by adding the x- and y-coordinate, both scaled to down to fit to the slice's height and width inside the texture, to the corresponding offset. The one pixel wide border around each slice is also taken into account.

$$(4.8) \quad \begin{aligned} x &= x_{\text{offset}} + \frac{x}{a} + \frac{1}{m} \\ y &= y_{\text{offset}} + \frac{y}{b} + \frac{1}{n} \end{aligned}$$

Trilinear Interpolation

Sample points are interpolated between the eight nearest voxels, using a trilinear interpolation scheme, to achieve a smooth result. As explained in Section 4.4, bilinear interpolation is already achieved by using the built-in functionality, but interpolation in z-direction has to be done manually. Instead of accessing only a single value from the nearest slice, two values are accessed from the two nearest slices. Overall, the above described process for accessing a volume texture is done twice: Once with the original sample point coordinates, and a second time with the z-coordinate incremented by one.

Bounding Box Intersection

For the intersection test of a view ray with the volume bounding box, a fast and straightforward algorithm, developed by Kay and Kayjia, is used. A description of the algorithm can be found at [Sig98].

Accumulation Loop

The accumulation loop is where the sampling and compositing steps of the Raycasting are executed. In this context, accumulation refers to how the color and opacity of the sample points are accumulated over time with each cycle of the loop. WebGL does not support while-loops in a Shader, so instead a for-loop with static length and a conditional break is used.

These six basic steps make up the body of the loop:

1. The texture coordinates of the sample point within the volume are computed. We do this by adding the ray direction, multiplied with the thus far covered distance, to the coordinates of the starting point. This is, depending on whether the camera is in- or outside the volume, either the entry point of the ray or the camera position itself.
2. With the coordinates, the sample point's value is retrieved from the volume data by calling the volume texture access function.
3. The transfer function is called and returns a 4D vector, containing RGB color information and an opacity value.
4. The obtained color and opacity are added to the so far accumulated values, utilizing the front-to-back compositing scheme.
5. We increase the value that stores the distance from the start point in preparation of the next iteration of the loop.
6. Before jumping into the next cycle, we check the two conditions for exiting the loop: Reaching or going past the exit point, or reaching an accumulated opacity of one. If either one of these is fulfilled, a break command is issued and the accumulation loop is finished.

5 Advanced Features

Apart from the basic volume rendering functionality, several advanced features have been added to the application.

5.1 Geometry Rendering

Even though the focus of this work is set on volume rendering, the ability to render surface geometry is a useful addition. Since geometry rasterisation is the most classical application of hardware accelerated computer graphics, it is fairly easy to implement. It can be used both to directly complement the rendered volume (e.g. by displaying a frame around the volumes bounding box) or to add standalone elements to the scene.

To implement geometry rendering alongside volume rendering, a multi-pass renderer is necessary, consisting of a geometry and a volume pass. In the first pass, all geometry objects within the scene are rendered to a texture using a previously created framebuffer object. The texture attached to that framebuffer object contains RGBA float values, however the alpha channel is used to store depth information instead of opacity. To ensure that the geometry is correctly aligned with the volume, the transformation matrices for the first render pass are generated with the same camera parameters that are used for the Raycasting during the second render pass.

To correctly merge the rasterized geometry with the rendered volume, even in the case that some geometry objects are positioned within the volume, any simple composition of the first render pass and the following volume pass does not produce a satisfying result. Therefore, during the second pass the texture containing the output of the first pass needs to be made available. Within the fragment Shader, the fragments position in normalized screen space is calculated and used to access the corresponding values of the first render pass from the texture. The actual merging takes place during the accumulation loop. At each sample, the distance from the viewer's location is tested against the depth value stored in the texture's alpha channel. As soon as that distance exceeds the stored value, all remaining sample points along the view ray are occluded by a geometry object. Thus the accumulation of sample points can be stopped and the texture's RGB value, multiplied by the remaining opacity, is added to the accumulated color. The accumulated opacity is set to one and the accumulation loop is exited. Finally, the accumulated values are written to the Shader's output.

Currently, there only is limited support for loading geometry objects from files. Figure 5.1 shows a scene containing polygonal surface geometry that was loaded from a VTK file [Kit].

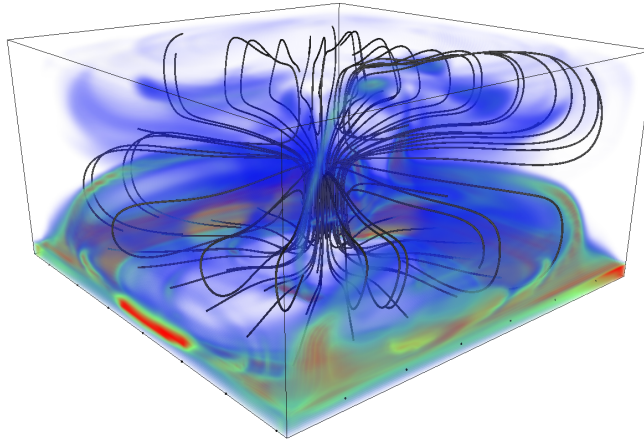


Figure 5.1: An example of the combination of volume rendering and surface rendering in the same scene, showing flow-lines added as geometry inside the volume. There is also a frame rendered around the bounding box.

5.2 Isosurface Visualization

Some regions inside the volume can be of special interest. This includes surfaces, which help to grasp the shape of a displayed object. One possibility to visualize the surfaces contained in volume data are isovalue contour surfaces [Lev88]. In this method, the surface is defined by voxels with the same value. To actually render the surfaces, the voxel's opacity has to be set based on whether they are part of a surface or not. Apart from simple approaches, where all voxels are either set opaquely or non-opaquely based on a threshold, the opacity can be set by using the local gradient of each voxel. The gradient of a voxel x_i is approximated by the following operator.

$$(5.1) \quad \nabla f(x_i) = \nabla f(x_i, y_j, z_k) \approx \begin{pmatrix} \frac{1}{2}(f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)), \\ \frac{1}{2}(f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)), \\ \frac{1}{2}(f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})) \end{pmatrix}, [\text{Lev88}]$$

In boundary regions, the central difference used to calculate the discrete differential, is replaced by either a forward or backward difference. This guarantees that the gradient of every voxel is well defined.

The gradient values are computed after the volume data has been written to 2D textures. For each texture that contains volume data, a texture containing the corresponding gradient data is created. Since it can be considered as image processing, it is easy to speed up the computation by doing it on the graphics card instead of the CPU. With a framebuffer object that matches the volume texture resolution, the gradient values for x, y and z direction are rendered to an RGB texture. In the case that a volume is made up out of more than a

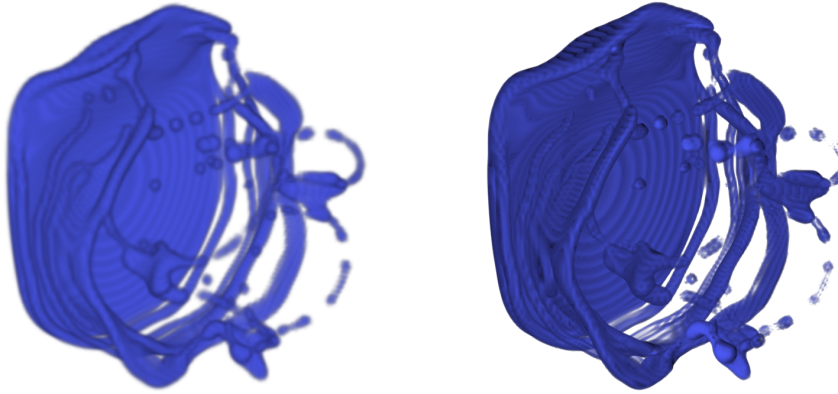


Figure 5.2: Left: Standard direct volume rendering. Right: Iso-surface mode with the following parameters: $f_v = 0.5$ $a_v = 1.0$ $r = w.0$.

single texture, as described in Section 4.5, up to three textures are required for the computation.

The opacity $a(x_i)$ of each voxel is now set according to the following expression [Lev88].

$$(5.2) \quad a(x_i) = \begin{cases} a_v, & \text{if } |\nabla f(x_i)| = 0 \text{ and } f(x_i) = f_v \\ a_v \cdot \left(1 - \frac{1}{r} \cdot \left|\frac{f_v - f(x_i)}{|\nabla f(x_i)|}\right|\right), & \text{if } |\nabla f(x_i)| > 0 \\ & \text{and} \\ & f(x_i) - r|\nabla f(x_i)| \leq f_v \leq f(x_i) + r|\nabla f(x_i)| \\ 0, & \text{otherwise} \end{cases}$$

The variables f_v , a_v and r are user-defined values. The value for f_v decides the displayed iso-surface, meaning that we are displaying the iso-surface, that is made up from voxels with value f_v . The opacity of that iso-surface is set with a_v .

To achieve a smoother final result, the opacity of voxels with a value unequal to f_v are set inverse proportional to their distance r (in voxel) from the nearest surface area.

5.3 Lighting

The visual quality of the rendered images can be increased by adding lighting and even a very simple model helps to lift the perceived quality of most volume data sets. Two examples are shown in Figure 5.3.

The Blinn-Phong [Bli77] reflection model was chosen for both its simplicity and good performance. It replicates the local illumination of a given surface point by combining an ambient I_a , diffuse I_d and specular term I_s .

$$(5.3) \quad \begin{aligned} I &= I_a + I_d + I_s \\ I_a &= k_a \\ I_d &= k_d \cdot (N \cdot L) \\ I_s &= k_s \cdot (N \cdot H) \end{aligned}$$

k_a, k_d and k_s are constant, implementation depended values used to control the contribution of the ambient, diffuse and specular term. To keep things simple, a directional light source is assumed, meaning that the light direction L is constant for all surface points. Furthermore, we assume that the light always comes from the direction of the camera V , meaning $L = V$. A gradient computation has already been added as a preprocessing step in the previous section, so the *surface* normal vector N can be easily obtained with a single texture access, followed by a normalization. The vector H is called the halfway-vector, and is used as an approximation of the reflection vector, that is used in the original Phong lighting. It is defined by

$$(5.4) \quad H = \frac{L + V}{|L + V|}$$

The lighting function is called for each sample point during the accumulation loop. A sample point's final color value is obtained by multiplying each of it's color channels with the light intensity I .

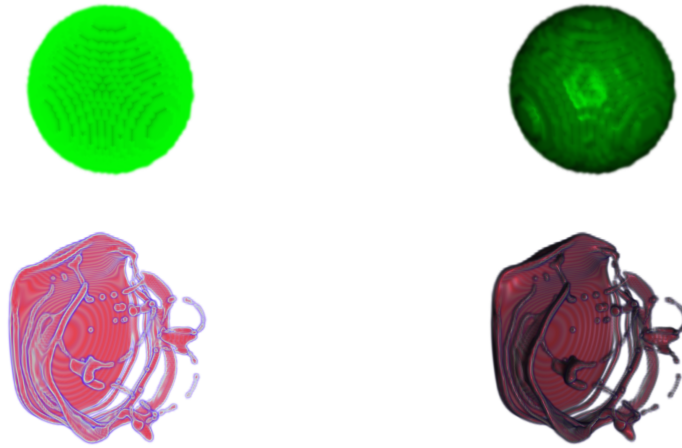


Figure 5.3: Left: Two different volume datasets, both unlit. Right: The same datasets, but with enabled lighting.

5.4 Animation

Volume data sources like simulations are often time dependent, meaning that for every point in time a volume file, containing the state of the simulation at that time, is created. Therefore it is a useful feature to be able to view different versions of a volume, or if possible to even *animate* it. To render such an animation, for each time step a volume file needs to be available, which is the main challenge of this feature. There are three different ways to solve this problem:

1. All relevant volume files are loaded into the main memory at the start of the application. Depending on the resolution of the volume, this is very memory consuming up to the point where there simply is not enough memory available to load the complete animation. Furthermore, due to the huge amount of data streamed and processed all at once, the application tends to freeze for a considerable time at startup.
2. Only the currently rendered volume is loaded from the hard drive into the system- and graphics card memory. Memory is not an issue with this version, however the hard drive speed is. Depending on the volume resolution, there will be a noticeable delay when switching between two volume files. If the delay is too long, a fluid animation would no longer be possible. Additional to the time needed to access the data on the hard drive, the preprocessing necessary for emulating 3D textures described in Section 4.5 increases the delay before a newly loaded volume file becomes available for rendering. Still, with reasonably fast hardware and a volume resolution below 128^3 , this version offers interactive framerates and is easy to implement.
3. A third solution requires a rather complicated streaming implementation. The volume files are streamed from the hard drive in advance, but at no point all volume files are present in system memory. This way, there are no delays when switching the rendered volume. However, it is difficult to perform the necessary preprocessing after streaming the volume files, without interrupting the rendering of the current volume and causing a considerable drop in framerates or even stuttering.

Within the scope of this thesis, the second variant was pursued, due to its simple implementation and reliability.

6 X3DOM

6.1 Introduction to X3DOM

X3DOM integrates the X3D standard into the HTML5 Document Object Model (DOM) [BEJZ09]. It aims to make 3D content available to web developers without previous experience in graphics programming. Because WebGL basically gives JavaScript access to the OpenGL ES 2.0 API, using it requires a certain knowledge of OpenGL programming.

X3DOM however only requires a declarative description of the 3D scene in X3D standard. For that purpose it offers a number of graphical primitives (e.g. boxes, spheres, light sources etc.) and options (e.g. material attributes, transformations) via so called nodes, that can be arranged in a scene.

6.2 Volume Rendering with X3DOM

X3DOM supports volume rendering with the *VolumeData* element. Thanks to the concept behind X3DOM, creating a scene that contains a volume is fairly easy and only involves a few lines of code. This is best demonstrated by simply showing the HTML body of a very simple X3DOM volume renderer:

```
<body>
  <X3D width='1024px' height='500px'>
    <Scene>
      <Background skyColor='0.0 0.0 0.0' />
      <Viewpoint description='Default' zNear='0.0001' zFar='100' />
      <Transform>
        <VolumeData id='volume' dimensions='4.0 2.0 4.0'>
          <ImageTextureAtlas containerField='voxels' url='room.png' numberOfSlices='91'
            slicesOverX='7' slicesOverY='14' />
          <OpacityMapVolumeStyle>
          </OpacityMapVolumeStyle>
        </VolumeData>
      </Transform>
    </Scene>
  </X3D>
</body>
```

Listing 6.1: Example of the minimum necessary code for volume rendering with X3DOM.

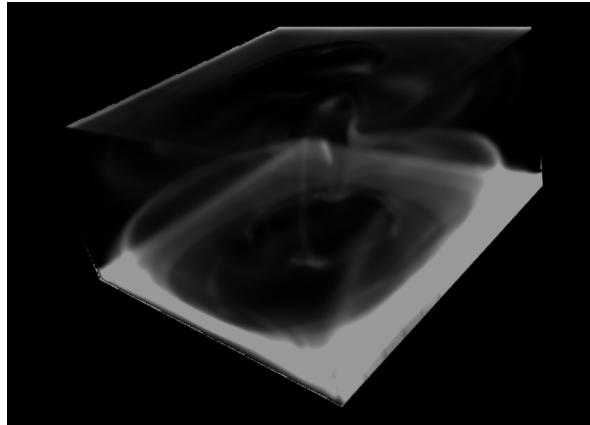


Figure 6.1: A screenshot taken with X3DOM’s volume renderer.

A requirement for volume rendering with X3DOM is to have a 2D image file, also called texture atlas, containing the volume’s slices in a very similar way to the 2D texture created in the preprocessing stage of our application. We were able to edit a texture atlas, that was created by our application and then saved as an image file, to match X3DOM’s expected layout and used it as input for the X3DOM application above. Figure 6.1 shows an image captured from the screen during testing of the application.

Apart from that, X3DOM’s volume renderer currently only works correctly when the number of slices in x-direction n_x equals the number of slices in y-direction n_y . We identified a small bug in the fragment Shader as the cause. The original implementation calculates the y-position d_y of a slice s as

$$d_y = \frac{\lfloor \frac{s}{n_y} \rfloor}{n_y} .$$

Correcting it to

$$d_y = \frac{\lfloor \frac{s}{n_x} \rfloor}{n_y}$$

makes it possible to use a texture atlas with any number of slices.

7 Evaluation

7.1 Stability

WebGL is still a relatively new standard and it can therefore be expected to run into stability issues every now and then.

During development and testing, the browser failed to compile perfectly fine Shader programs quite frequently. Most of the time, that behaviour was triggered by refreshing the web page (and therefore also restarting the WebGL application) numerous times, especially if changes were made to the code in mean time. It also occurred, even though much less frequently, if the browser had already been open for a while, possibly having run a WebGL application before. On the bright side however, we rarely experienced a crash of the web browser due to our application.

7.2 Performance

Benchmarks were conducted with an AMD HD7870 graphics card, combined with an AMD Phenom II X6 1045T and 8GB RAM. We ran the application using Mozilla Firefox version 16.0.1 and Windows 7 64bit as operating system. On Windows systems, Firefox uses ANGLE as backend for WebGL. ANGLE translates WebGL API calls to DirectX9 API calls, trying to avoid compatibility issues with OpenGL and Windows. Yet, for our application, Firefox is explicitly configured to use native OpenGL. Otherwise the Shader programs usually fail to compile correctly if the accumulation loop exceeds a certain amount of cycles.

The benchmark was conducted with a canvas resolution of 1024×512 and a volume dataset of the size $91 \times 46 \times 91$. Table 7.1 shows the average frames per second (fps) achieved with varying sample rates. The fps are compared between four different render modes: Transfer texture only (TT), transfer texture in combination with lighting (TTwL), iso-surface visualization (ISO) and finally iso-surface visualization with lighting (ISOwL). Since geometry rendering is currently not optional, in all four modes a bounding box frame is rendered as well.

It is important to note, that the sampling is done in texture space, where the bounding box is normalized in all directions. Thus, the greatest distance a ray can travel through the volume is $\sqrt{3}$. Together with the distance between samples, referred to as step-size, the maximum amount of samples per fragments can be calculated.

The function used for redrawing the frame as often as possible is browser dependent and deviations from the expected maximum framerate are not uncommon. We observed, that

Table 7.1: Benchmark - Shows the average fps while rendering a volume dataset of the size $91 \times 46 \times 91$ with a resolution of 1024×512 .

Step-Size	TT	TTwL	ISO	ISOwL
0.01	67.0	66.8	66.9	59.7
0.0075	67.0	60.2	63.9	53.1
0.005	67.0	50.5	54.3	42.8
0.0025	51.0	33.3	37.3	27.3
0.001	28.9	16.3	19.0	12.9

Firefox’s `requestAnimationFrame()` function seems to limit the fps to an odd-valued 67 frames per second. According to the official documentation, a redraw occurs up to 60 times per second.

The achieved framerate is satisfying as far as the interactivity of the application is concerned. Heavily noticeable stuttering that starts below 20 fps, only sets in for a very small step-size. The worst case amount of sample points collected for a single fragment is up to 1400 in that case.

7.3 Comparison with X3DOM

7.3.1 Features

Compared to our implementation, X3DOM lacks certain comfort and quality features.

First of all, the employed bilinear filtering does not accommodate for the border regions. As described earlier, not doing so results in visual artefacts, best described as one boundary surface of the volume ‘bleeding’ into the opposite side. This can be clearly seen in Figure 6.1. The X3DOM volume renderer uses a ray-casting algorithm, that uses the first approach described in Section 3.6.1 for the ray generation. Of course this means that it is not possible to move the camera into the volume, a problem we avoided by using a more generalised, if slower, approach.

While X3D offers several different render modes for volumes, X3DOM currently only seems to support standard direct volume rendering with a transfer function. Neither lighting nor iso-surface visualization are available

Just like our application, X3DOM utilizes a 1D texture for the transfer function. But unlike our implementation, the texture is read from an image file and cannot be interactively changed in real time without expanding the code.

In X3DOM the volume is loaded using an image file that contains the 2D representation of the volume. If the source of the volume data does not already output such an image file, it has to be created from the raw volume data by an external application. While there are certainly some requirements concerning the raw data that can be read with our application,

Table 7.2: Benchmark Comparison of average fps with X3DOM.

Max Cycle	X3DOM	Step-Size	TT
60(default)	60	-	-
280	60	0.005	67.0
560	50	0.0025	51.0
1400	35	0.001	28.9

we integrated the conversion to a 2D texture, which is rather specific to WebGL because of the lacking 3D texture support, as a preprocessing step.

7.3.2 Performance

As a result of the rather small feature set, X3DOM's volume renderer runs pleasantly fast. In fact, it is difficult to see framerates below the upper limit of 60 frames per second on a dedicated graphics card. Depending on the sample rate and as long as most of the advanced features are deactivated, our implementation 'suffers' from the same problem on our benchmark system. Hence, it is difficult to make a comparison of the implementations. On a less powerful system, our implementation drops below 67 FPS even with iso-surface rendering and lighting disabled, while X3DOM still remains at the 60 fps limit. This is expected since we have some additional overhead, due to the shiftable features, and a higher default sampling rate. However, a meaningful performance comparison is still simply impossible as long as one application runs faster than the fps-limiter would allow.

That is why we are manually adjusting the sample rate of X3DOM's volume renderer to be both high enough to push the fps below 60 as well as having it match the sample rate of our implementation. This is done by manually setting a higher value for the maximum amount of loop cycles. The result is illustrated in Table 7.2.

We compare X3DOM's performance to our basic display mode, with both iso-surfaces and lighting disabled. This mode resembles X3DOM's volume renderer the most, even though it still renders the additional geometry pass. For both applications, the size of the rendered window is again set to 1024×512 and the same dataset as before is used.

The overall performance is still competitive in a comparable scenario, even tough at the cost of disabling most of the advanced features.

8 Conclusion and Future Work

We presented a web-based volume renderer, using GPU acceleration to achieve interactive framerates. To that end, the Raycasting algorithm for direct volume rendering was successfully implemented using WebGL. In addition, a number of advanced features could be added to the application.

To our knowledge, our application is possibly the most feature rich web-based volume renderer at this time. Nevertheless, interactive framerates are sustained in most scenarios, given a reasonably fast system.

Even though WebGL already produces quite impressive results both performance- and feature-wise, some stability issues still occasionally occur in the current version. During the development and testing of the application, we also experienced some difficulties with browser dependent functionality. For the affected functions, some extra care has to be exercised to guarantee cross-browser compatibility.

Some aspects of the application could be further improved in the future. This includes the basic volume rendering algorithm, which still lacks some optimizations. The accumulation loop in particular is prone to unnecessary, redundant operations, that can heavily afflict the performance. For example, a possible speed-up could be achieved by using a look-up array or 1D texture, instead of calculating the 2D coordinates for a set of 3D coordinates in real-time for each sample point.

Apart from that, additional acceleration techniques could be implemented. Early ray termination is already in use, but empty space skipping could still improve the performance in some scenarios.

Another obvious choice for future improvements is upgrading the advanced feature set. A more sophisticated lighting system, featuring shadows and other effects, would certainly raise the visual quality. The ability to display more than one iso-surface at once, would also be beneficial, just as improved support for loading common geometry files would be.

In the scope of this thesis, little to none effort was put into the creation of a comfortable user interface. This definitely could be improved in the future to guarantee a better user experience.

9 Acknowledgements

I wish to thank my professor for giving me the opportunity to perform this work in his institute. I would also like to express my gratitude to both of my advisers for their continued support and guidance during the length of this work. Furthermore I would like to extend my thanks to the ITLR Stuttgart for supplying a volume dataset.

A Appendix

A.1 Zusammenfassung

Web-basierte Anwendungen erfreuen sich zunehmend großer Beliebtheit in einer Vielzahl von Einsatzgebieten. Entsprechend werden auch auf dem Gebiet der web-basierten 3D-Grafik zahlreiche Fortschritte erzielt.

Im Rahmen dieser Bachelorarbeit wird eine Implementierung zur Darstellung von Volumengrafik auf Webseiten mithilfe der WebGL API vorgestellt. Es wird zunächst ein Überblick über die theoretischen Grundlagen der Volumengrafik, sowie über die üblichen Ansätze einer GPU-Implementierung, gegeben. Dies umfasst sowohl die zugrundeliegenden optischen Modelle und das Volume-Rendering-Integral, als auch das Raycasting Verfahren und ein Textur-basiertes Verfahren der Volumengrafik. Anschließend werden die einzelnen Teile der Implementierung im Detail erläutert. Dies umfasst sowohl all jene Teile, die dem in JavaScript geschriebenen Grundgerüst der Anwendung zugehörig sind, als auch die auf der Grafikkarte ausgeführten Shader Programme. Darüber hinaus werden in einem weiteren Kapitel die fortgeschrittenen Techniken näher behandelt.

Ziel dieser Arbeit ist es die grundlegenden, sowie einige fortgeschrittene Methoden der Volumengrafik zu implementieren und darüber hinaus eine interaktive Bildwiederholungszahl zu erreichen. In diesem Sinne wird der Erfolg der Implementierung anhand ihrer Lauffähigkeit und Leistung diskutiert und zudem eine alternative Möglichkeit zur Volumen-Darstellung auf Webseiten zum Vergleich herangezogen.

Es folgt abschließend die Feststellung, dass die Implementierung im Rahmen dieser Arbeit erfolgreich war und es wird zudem auf einige, mögliche zukünftige Arbeitsgebiete sowie Verbesserungsmöglichkeiten hingewiesen.

Bibliography

- [BEJZ09] J. Behr, P. Eschler, Y. Jung, M. Zöllner. X3DOM: a DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, Web3D '09, pp. 127–135. ACM, New York, NY, USA, 2009. doi:10.1145/1559764.1559784. URL <http://doi.acm.org/10.1145/1559764.1559784>. (Cited on pages 10, 11 and 39)
- [Bli77] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '77, pp. 192–198. ACM, New York, NY, USA, 1977. doi:10.1145/563858.563893. URL <http://doi.acm.org/10.1145/563858.563893>. (Cited on page 36)
- [CCF94] B. Cabral, N. Cam, J. Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings of the 1994 symposium on Volume visualization*, VVS '94, pp. 91–98. ACM, New York, NY, USA, 1994. doi:10.1145/197938.197972. URL <http://doi.acm.org/10.1145/197938.197972>. (Cited on page 16)
- [CN94] T. J. Cullip, U. Neumann. Accelerating Volume Reconstruction With 3D Texture Hardware. Technical report, Chapel Hill, NC, USA, 1994. (Cited on pages 10, 11 and 16)
- [CSK⁺11] J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, O. Ruiz. Interactive visualization of volumetric data with WebGL in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology*, Web3D '11, pp. 137–146. ACM, New York, NY, USA, 2011. doi:10.1145/2010425.2010449. URL <http://doi.acm.org/10.1145/2010425.2010449>. (Cited on page 11)
- [EKE01] K. Engel, M. Kraus, T. 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, pp. 9–16. ACM, New York, NY, USA, 2001. doi:10.1145/383507.383515. URL <http://doi.acm.org/10.1145/383507.383515>. (Cited on pages 10 and 16)
- [Ert11] T. Ertl. Visualization Course Material Lecture 8, 2011. URL [http://www.vis.uni-stuttgart.de/nc/lehre/details.html?tx_visteaching_pi1\[type\]=tx_visteaching_lecture&tx_visteaching_pi1\[uid\]=76&tx_visteaching_pi1\[back\]=773](http://www.vis.uni-stuttgart.de/nc/lehre/details.html?tx_visteaching_pi1[type]=tx_visteaching_lecture&tx_visteaching_pi1[uid]=76&tx_visteaching_pi1[back]=773). (Cited on page 17)

- [HLSR08] M. Hadwiger, P. Ljung, C. R. Salama, T. Ropinski. Advanced illumination techniques for GPU volume raycasting. In *ACM SIGGRAPH ASIA 2008 courses*, SIGGRAPH Asia '08, pp. 1:1–1:166. ACM, New York, NY, USA, 2008. doi:10.1145/1508044.1508045. URL <http://doi.acm.org/10.1145/1508044.1508045>. (Cited on pages 10, 13 and 15)
- [Jia12] P. Jiarathanakul. Ray Marching Distance Fields in Real-time on WebGL. Technical report, University of Pennsylvania, 2012. (Cited on page 11)
- [JKD⁺12] H. Jacinto, R. K echichian, M. Desvignes, R. Prost, S. Valette. A web interface for 3D visualization and interactive segmentation of medical images. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pp. 51–58. ACM, New York, NY, USA, 2012. doi:10.1145/2338714.2338722. URL <http://doi.acm.org/10.1145/2338714.2338722>. (Cited on page 11)
- [Joh07] T. Johansson. Taking the canvas to another dimension, 2007. URL <http://my.opera.com/timjoh/blog/2007/11/13/taking-the-canvas-to-another-dimension>. (Cited on page 9)
- [Khr] Khronos. Khronos WebGL Wiki - WebGL and OpenGL Differences. URL http://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences#No_3D_Texture_support. (Cited on pages 21 and 23)
- [Khr09] Khronos. Khronos Launches Initiative to Create Open Royalty Free Standard for Accelerated 3D on the Web, 2009. URL <http://www.khronos.org/news/press/khronos-launches-initiative-for-free-standard-for-accelerated-3d-on-web>. (Cited on page 9)
- [Khr11] Khronos. WebGL Specification 1.0, 2011. URL <https://www.khronos.org/registry/webgl/specs/1.0/>. (Cited on page 21)
- [Kit] Kitware Inc. VTK File Formats. URL <http://www.vtk.org/VTK/img/file-formats.pdf>. (Cited on page 33)
- [KW03] J. Kruger, R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pp. 38–. IEEE Computer Society, Washington, DC, USA, 2003. doi:10.1109/VIS.2003.10001. URL <http://dx.doi.org/10.1109/VIS.2003.10001>. (Cited on pages 10, 11 and 18)
- [Lev88] M. Levoy. Display of Surfaces from Volume Data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, 1988. doi:10.1109/38.511. URL <http://dx.doi.org/10.1109/38.511>. (Cited on pages 9, 34 and 35)
- [LL94] P. Lacroute, M. Levoy. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, SIGGRAPH '94, pp. 451–458. ACM, New York, NY, USA, 1994. doi:10.1145/192161.192283. URL <http://doi.acm.org/10.1145/192161.192283>. (Cited on pages 10 and 11)

-
- [Max95] N. Max. Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995. doi:10.1109/2945.468400. URL <http://dx.doi.org/10.1109/2945.468400>. (Cited on pages 13 and 15)
- [MHE01] M. Magallón, M. Hopf, T. Ertl. Parallel Volume Rendering Using PC Graphics Hardware. In *Proceedings of the 9th Pacific Conference on Computer Graphics and Applications*, PG '01, pp. 384–. IEEE Computer Society, Washington, DC, USA, 2001. URL <http://dl.acm.org/citation.cfm?id=882473.883442>. (Cited on page 11)
- [MW08] M. Moser, D. Weiskopf. Interactive Volume Rendering on Mobile Devices. In *Workshop on Vision Modelling and Visualization VMV '08*, pp. 217–226. 2008. (Cited on page 11)
- [Sig98] Siggraph. Ray - Box Intersection, 1998. URL <http://www.siggraph.org/education/materials/HyperGraph/raytrace/rtinter3.htm>. (Cited on page 31)
- [Vuk07] V. Vukićević. Canvas 3D: GL power, web-style, 2007. URL <http://blog.vlad1.com/2007/11/26/canvas-3d-gl-power-web-style/>. (Cited on page 9)
- [WE98] R. Westermann, T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pp. 169–177. ACM, New York, NY, USA, 1998. doi:10.1145/280814.280860. URL <http://doi.acm.org/10.1145/280814.280860>. (Cited on page 16)

All links were last followed on October 23, 2012.

Declaration

All the work contained within this thesis, except where otherwise acknowledged, was solely the effort of the author. At no stage was any collaboration entered into with any other party.

(Michael Becher)