Konputazio Zientziak eta Adimen Artifizialaren Saila
Departamento de Ciencias de la Computación e Inteligencia Artificial
Computer Science and Artificial Intelligence Department

eman ta zabal zazu

Universidad del País Vasco
Euskal Herriko Unibertsitatea

Informatika Fakultatea
Facultad de Informática
Computer Science Faculty

Ubiquitous volume rendering in the web platform

A dissertation submitted for the degree of Doctor in Computer Science

Ander Arbelaiz Aranzasti

Advisors
Alejandro García-Alonso
Aitor Moreno Guerrero

Donostia/San Sebastián, 2019

Konputazio Zientziak eta Adimen Artifizialaren Saila
Departamento de Ciencias de la Computación e Inteligencia Artificial
Computer Science and Artificial Intelligence Department

eman ta zabal zazu

Universidad del País Vasco   Euskal Herriko Unibertsitatea

Informatika Fakultatea
Facultad de Informática
Computer Science Faculty

Ubiquitous volume rendering in the web platform

A dissertation submitted for the degree of Doctor in Computer Science

Ander Arbelaiz Aranzasti

Advisors
Alejandro García-Alonso
Aitor Moreno Guerrero

Donostia/San Sebastián, 2019

Acknowledgment

En primer lugar, me gustaría expresar mi más sincera gratitud a mi director de tesis el Prof. Alejandro García Alonso por su gran apoyo, el cual no solo ha abarcado el transcurso del proceso doctoral, sino que empezó en mi etapa de estudiante durante la licenciatura. Su guía y su infinita paciencia durante todo este tiempo han sido indispensables para poder llevar a cabo este trabajo de investigación. No puedo pensar en un mejor director para el doctorando. Espero que sus consejos y buen hacer hayan quedado reflejados en la escritura de esta tesis.

También me gustaría agradecer a mi co-director de tesis el Dr. Aitor Moreno por todos los consejos, sugerencias y lecciones recibidas en el día a día. Éstas son incontables e invaluables. Su tutela y dedicación hacia la dirección de esta tesis han ayudado en gran medida ha llevarla a buen puerto.

Además de a mis directores, me gustaría dar las gracias en especial al Dr. Luis Kabongo y al Dr. Nicholas Polys. A Luis por todos sus valiosos comentarios y su labor de interlocucíon con los miembros del Web3D. Su ayuda en los trabajos de investigación han sido esenciales para poder completar esta tesis. A Nicholas por sus comentarios y su invitación para participar en el X3D Medical Working Group.

Agradezco a todos mis compañeros de Vicomtech los ánimos recibidos así como sus estimulantes conversaciones en los descansos del café que han vitaminado esta tesis.

Por último, pero no por ello menos importante, me gustaría agradecer a mi familia y amigos todos los ánimos recibidos. En especial a mis padres que me han dado su apoyo desde el inicio y siempre han estado presentes para ayudarme.

Contents

List of Figures

List of Tables

Listings

Resumen

La principal hipótesis de la tesis es que se puede lograr el renderizado de datos volumétricos de forma ubicua utilizando WebGL. La tesis enumera los desafíos que se deben enfrentar para lograr dicho objetivo. Los resultados obtenidos permiten a los desarrolladores de contenido web la integración de visualizaciónes interactivas de datos volumétricos dentro de páginas web estándar HTML5.

Los desarrolladores de contenido web solo necesitan declarar los nodos X3D que proporcionan el renderizado de las características que desean. A diferencia de los sistemas que distribuyen programas específicos de GPU. La arquitectura presentada crea automáticamente el código de GPU requerido para el proceso de renderizado con WebGL. Este código se genera directamente desde los nodos X3D declarados en la escena virtual. Por lo tanto, los desarrolladores de contenido no necesitan saber sobre la GPU.

La tesis amplía la investigación previa sobre estructuras de datos de volumen compatibles con la web, renderizado híbrido de objetos 3D y volúmenes, renderizado de volúmenes progresivo y algunos problemas específicos relacionados con la visualización de conjuntos de datos volumétricos en el ámbito médico.

Finalmente, la tesis contribuye al estándar ISO/IEC X3D con algunas propuestas para extender y mejorar el componente de renderizado de volumen. Las propuestas se encuentran en un estado avanzado, previo a su aceptación por parte del consorcio Web3D.

Laburpena

WebGL erabiliz bolumenen renderizazio nonahikoa lortu egin daitekela tesiaren hipotesi nagusia da. Tesi honek helburu hori lortu ahal izateko erronkak zerrendatzen ditu. Lortutako emaitzen bitartez, esfortzu gutxirekin, web edukien garatzaileek HTML5 web orrialdeetan bolumen renderizazio interaktiboak txertatu ditzakete.

Eduki garatzaileek soilik X3D nodoak adierazi behar dituzte nahi dituzten renderizazio ezaugarriak lortu ahal izateko. GPU programa finkoak banatzen dituzten sistemak ez bezala, aurkeztutako arkitektura automatikoki sortzen du WebGL behar duen GPU kodea. Kode hau zuzenean sortzen da eszena birtualean adierazitako X3D nodoen arabera. Hori dela eta, eduki garatzaileek ez dute GPU-ari buruzko jakintza izan behar.

Tesi honek, web-arekin bateragarria den bolumen data egitura, bolumenen eta 3D objetuen renderizazio hibridoa, bolumenen renderizazio progresiboa eta medikuntza arloan, datu bolumetrikoak bistaratzearekin lotutako arazo zehatz batzuei buruzko aurreko ikerketak zabaltzen ditu.

Bukatzeko, tesiak ISO/IEC X3D estandarraren bolumen renderizazio atala zabaltezko eta hobetzeko proposamenak aurkezten ditu. Proposamenak egoera aurreratuan daude, Web3D partzuergoaren onarpenaren zain.

Summary

The main thesis hypothesis is that ubiquitous volume rendering can be achieved using WebGL. The thesis enumerates the challenges that should be met to achieve that goal. The results allow web content developers the integration of interactive volume rendering within standard HTML5 web pages.

Content developers only need to declare the X3D nodes that provide the rendering characteristics they desire. In contrast to the systems that provide specific GPU programs, the presented architecture creates automatically the GPU code required by the WebGL graphics pipeline. This code is generated directly from the X3D nodes declared in the virtual scene. Therefore content developers do not need to know about the GPU.

The thesis extends previous research on web compatible volume data structures for WebGL, ray-casting hybrid surface and volumetric rendering, progressive volume rendering and some specific problems related to the visualization of medical datasets.

Finally, the thesis contributes to the ISO/IEC X3D standard with some proposals to extend and improve the volume rendering component. The proposals are in an advance stage towards their acceptance by the Web3D Consortium.

Chapter 1

Introduction

This chapter introduces the objectives and challenges of this thesis. Firstly, Section 1.1 describes the volume data type. Section 1.2 presents the problem of volumetric visualization, the hypothesis to overcome it and the challenges addressed in this thesis. Finally, Section 1.3 describes how the rest of the chapters of this thesis are organized.

1.1 Volume data

Traditional 3D objects are created using surface based representations such as polygonal meshes and NURBS patches. For these data objects, all properties of the model are evaluated at certain surface points. Rendering of the object's characteristics such as materials, colours and shading are based on those points. Real-world objects are approximated by a geometric model that simplifies their content at the surface level. For volumetric visualization, this is different. The volume data in the inside is as important as the volume data that encloses the volume (surface).

Volume data refers to scalar data which has a 3D nature, a discrete 3D scalar field. Usually, it is obtained by measuring the natural phenomena or by numerical simulation. Thus, its main applications are oriented towards scientific visualization. For instance, volume data can be usually found in one of the following domains:

- **Medical imaging.** 3D Image acquisitions of the inside of the human body with Computed Tomography (CT) or Magnetic Resonance Imaging (MRI) (see Figure 1.1).

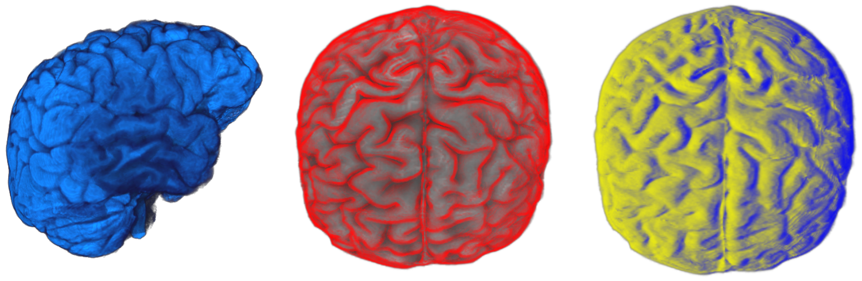


Figure 1.1: Non-photorealistic volume rendering of the `brain` MRI dataset.

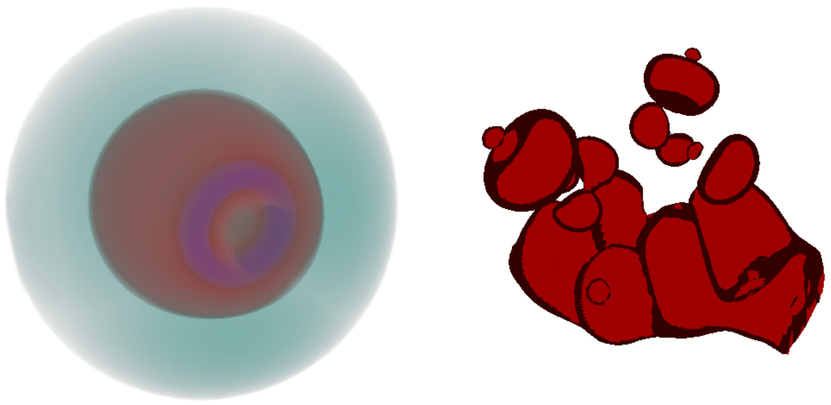


Figure 1.2: Non-photorealistic rendering of simulated 3D volumetric data.

- **Engineering simulation.** Simulation of natural phenomena, computational fluid dynamics (CFD), finite element Analysis (FEA), etc.
- **Mathematics and physics.** Numerical computation of probabilistic 3D distributions, physics simulation, etc. (see Figure 1.2).
- **Industrial manufacturing.** Reverse engineering, manufacturing quality control, etc. (see Figure 1.3).
- **Games**. 3D fuzzy objects or elements like fog, fire and clouds.

Generally speaking, volume data is a representation of scalar data, measured from a continuous signal, which has been discretized in a 3D space. It can be seen as a three dimensional array of cubes evenly spaced, where each unitary cube is called voxel. Each voxel represents an scalar value. Volume data can also be considered as the output values from a function of a 3D signal, where for each input (X, Y, Z) an scalar value output is generated.

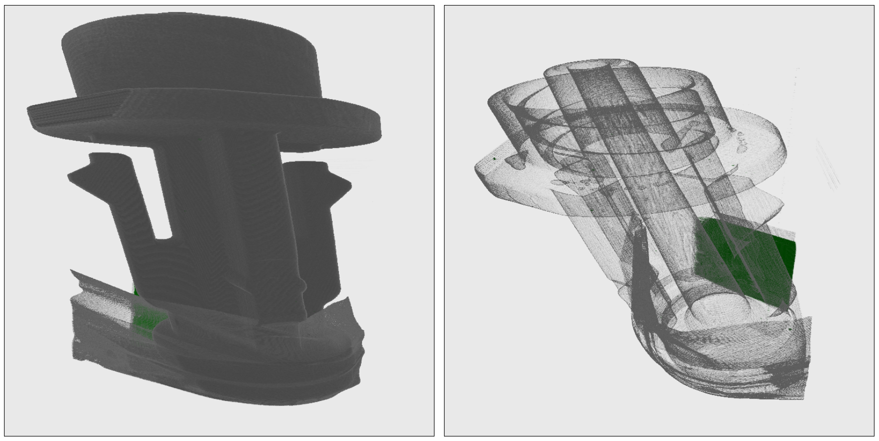


Figure 1.3: Volume rendering of a manufactured plastic part.

Volume data contains useful information on the inside of the object. Therefore, the traditional rendering algorithms which are based on surface based representations are not suitable for its visualization. For all the mentioned domains, volume rendering algorithms allow to visualise the whole volume data. They benefit from a true rendering of the represented object taking into account all the information it is composed of. In this case, inside data is as important as surface data.

1.2 Thesis objectives and challenges

As previously stated, volumetric visualisation is employed in several scientific fields. Volumetric visualization software is often proprietary, distributed as part of expensive volumetric data acquisition machines. There are also some open source solutions, like VTK (Kitware, 2019). In both cases volume visualisation software is tailored to specific applications and hardware due to each specific application domain need. The high computing requirements of volumetric visualization algorithms has encouraged researchers and software developers to pursuit the use of High Performance Clusters (HPC) or take the most of computing capabilities from dedicated graphics computing hardware (GPU).

These factors have aggravated the problem of sharing volumetric visualization content among users and peers, creating a segmented market of different volume data formats and the need of expensive equipment to visualise this type of content.

The **goal** of this thesis is to propose new algorithms and to evaluate architectures that work in a ubiquitous cross-platform solution. The main motivation of this thesis is that volume visualisation should be available in multiple devices, opening the access to volumetric content to a much wider audience.

Nowadays, the web platform with the uprising of social media is the most used environment to share multimedia content among users; it reaches the widest range of usable devices (PCs, tablets and smartphones). So, the web is the targeted platform in this thesis. Hardware acceleration is a requirement to reach real-time frame rates in volume rendering. OpenGL ES 2.0 API is the 3D graphics API available for the Web and better supported by all vendors. For these reasons the research goal takes WebGL (Khronos, 2016), that is built on top of OpenGL ES 2.0, as a foundation stone. This is the main design **hypothesis**: the thesis research challenges are met finding and testing solutions around this hypothesis.

The following points summarize the **challenges** met by this thesis in the context of volume rendering in the *ubiquitous web platform*.

i) To improve the volume data representation structure (see Chapter 3).

ii) To make easier to content developers the integration of volume visualization (see Section 10.2, Chapters 4 and 9)

iii) To provide a solution for hybrid rendering: volume data and surface data (see Chapter 5).

iv) To achieve a high-quality rendering with large datasets (see Chapter 6).

v) To solve problems related to the visualization of medical datasets: segmented visualization and patient data privacy, among them (see Chapter 7).

vi) To find and propose new extensions to X3D standard (see Chapter 8).

Section 10.2 describes the contributions of this thesis. Reading that section provides a deeper insight of the challenges met and solved in this thesis. The contribution described in Section 10.2, explains an important addressed challenge in this thesis: The component developed in this thesis makes it easier to create volumetric content for developers without specific knowledge on computer graphics rendering. The required GPU shader code is automatically generated with the proposed component whose architecture and implementation has been carried out in this thesis.

1.3 Thesis structure

This thesis is structured as follows. Chapter 2 briefly presents the background literature including the classic volume rendering techniques. It also includes the state of the art for volume rendering in the web platform.

The following chapters present research background at the beginning of each chapter as a matter of introducing the specific research work which is being extended or to make context for the presented contributions.

Chapter 3 presents improvements to a web compatible volume data structure: *ImageTextureAtlas*. Chapter 4 presents an architecture for the automatic generation of the volume rendering visualization programs and the provided implementation: the X3DOM volume rendering component.

Chapter 5 describes a novel hybrid volume rendering for the mixed visualization of volume and surface mesh data. Chapter 6 presents a progressive volume rendering ray-casting algorithm for the interactive high quality rendering of large datasets. Chapter 7 focuses in contributions for the visualization of volumetric data in the web browser for the medical domain.

Chapter 8 proposes extensions to the ISO/IEC X3D standard based in feedback received by the Web3D community. Chapter 9 validates the contributions of this thesis. Finally, Chapter 10 summarizes the contributions of this thesis.

Chapter 2

Literature Review

This chapter reviews some relevant literature about volume rendering and covers the necessary background concepts to introduce the reader in the topic. Volume rendering has been researched thoroughly in the past. The main initial contributions are covered as well as recent research works. For a more detailed revision of the state of the art in volume rendering several surveys are referenced.

The work of this thesis is focused in the rendering of volumetric data in the web platform. In this regard, the main contributions related to the web based volumetric visualization are presented.

In the following chapters of this thesis, previous research works by other authors are also presented. In those cases, their research work is described in more detail, in order to establish the necessary background to present the contributions of those chapters.

The chapter is structured as follows. Section 2.1 briefly presents the volume rendering optical model. Section 2.2 presents the most popular direct volume rendering techniques. Finally, Section 2.3 reviews the volume rendering literature focused in the area of research of this thesis: the web platform.

2.1 Volume rendering

Generally speaking, volume rendering algorithms provide a representation of the physical properties of a participating medium. These physical properties are used to compute light transport for the image generation. Every scalar value in the data distribution of the volume data represents a light emitting particle and these particles are mapped to coloured (RGBA) pixels in a projected image. Because the solution of the light transport equation is too complex, simplified models are used. Max (1995) reviewed the different optical models used nowadays in volume rendering.

1. **Absorption**. Particles absorb all the light. They do not emit nor scatter light.

2. **Emission**. Particles only emit light. It is assumed that the absorption is negligible.

3. **Absorption and emission**. Particles emit light and also occlude (absorb) light. There is no scattering or indirect illumination.

4. **Scattering and shading**. Includes scattering of illumination from an external source from the voxel under consideration. Scattered light can either illuminates the voxel or it can be shadowed by particles between the light and the voxel under consideration.

5. **Multiple scattering**. The complete illumination model is evaluated, including all of the previous models.

The basic model and most commonly used in volume rendering is the *Absorption and emission* model. This model is given by a differential equation that describes light transport by differential changes in radiance. It can be solved by integration along the direction of light flow. Given a single ray that traverses the volume, such that the light enters the volume at $s = 0$ and exits the volume at $s = D$, the radiance of the light emitted from the volume is defined by Equation 2.1 where I_0 is the radiance of light as it enters the volume from the background at the position $s = s_0$. I_D is the radiance of the light as it exits the volume at the position $s = s_D$.

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(t)dt} + \int_{s_0}^{D} q(s) e^{-\int_{s}^{D} \kappa(t)dt} ds \tag{2.1}$$

The main objective of volume rendering is to compute the volume rendering integral Equation 2.1. Numerical methods are applied to find an approximation to the solution as shown in Equation 2.2. This equation divides the integral into intervals where T is the transparency (*optical depth* of the model) and C_i is the colour contribution.

$$I_D = \sum_{i=0}^{n} C_i \prod_{j=i+1}^{n} T(j), \text{with } C_0 = I(s_0) \tag{2.2}$$

The discretized volume rendering integral can be solved with iterative computation using front-to-back (from the camera towards the volume) or back-to-front compositing. Equation 2.3 splits the summations and multiplications of Equation 2.2 into separate simpler operations that are executed sequentially.

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst}) * C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst}) * \alpha_{src} \end{aligned} \tag{2.3}$$

In following chapters, the term *blending and accumulation of opacity and colour* is used. In those cases, it refers to the compositing Equation 2.3. The work of this thesis focuses only in the *absorption and emission* model. For advanced illumination models with scattering, shading and multi-scattering, the survey by Jönsson et al. (2014) covers the state of the art in advanced volumetric interactive illumination techniques.

2.2 Direct volume rendering techniques

In the state of the art, volume rendering techniques can be classified into two categories: direct or indirect. Indirect methods try to extract the surface data in a pre-processing step by fitting geometry. Then, the reconstructed surfaces are rendered. In contrast, Direct Volume Rendering (DVR) methods map directly into screen space the volumetric content without using geometric primitives as an intermediate representation.

Several of the DVR methods take advantage of GPU acceleration due to the fact that they can be parallelized. These method are usually classified in either object-order methods or image-order methods. Object-order methods determine for each data sample, iterating over the object in the scene, how it affects the pixels on the image plane. In contrast, image-order methods iterate over the pixels in the image to be produced, rather than in the objects to be rendered.

The **shear-warp factorization** is an object-order technique presented by Lacroute and Levoy (1994) and Lacroute (1995). The volume data is *sheared* perpendicular to a base plane, aligned with two axes of the data set. Then the slices are projected and composed (in front-to-back or in back-to-front order) into the base plane. The final composed image is transformed with a *warp* operation. Because of its memory access pattern, it has not popularized in the GPU hardware. This technique has been improved and revisited by Sweeney and Mueller (2002); Schulze et al. (2003) and Li et al. (2010)

Splatting is another object-order technique. Originated by Westover (1990a), the data is traversed in 3D object space and the volumetric elements are projected into the image plane. Kernels are used for the projection 3D reconstruction. Westover (1990a,b) used the term *splat* to describe the flattening of the kernel in the image plane. Improvements on this technique and GPU acceleration were researched by Laur and Hanrahan (1991); Zwicker et al. (2002); Ren et al. (2002); Botsch et al. (2004); Botsch et al. (2005) and Neophytou and Mueller (2005)

Cell projection is an object-order technique focused in the rendering of unstructured volume data. Shirley and Tuchman (1990) presented the first work with the projected tetrahedra algorithm. The *cells* of a grid are traversed and *projected* into the image plane. Variations of this algorithm where made to accommodate for graphics hardware (Wylie et al., 2002; Weiler et al., 2003; Maximo et al., 2010). It is important to sort cells before traversing them. There are many approaches to cell sorting, but the most popular is the MPVO algorithm by Williams (1992). Over the years, researchers have made many improvements to the cell projection algorithm (Silva et al., 1998; Comba et al., 1999; Callahan et al., 2005). Further references can be found in the survey done by Silva et al. (2005).

Texture slicing is another object-order rendering technique, also known as *texture mapping* and first employed by Cabral et al. (1994). This technique is intended to be used with graphics hardware. In the context of 3D texture mapping, multiple planes parallel to the viewing plane are clipped against the parametric texture domain. Then the GPU hardware is used to interpolate 3D texture coordinates in the clipped polygon vertices. Finally, the interpolated texture coordinates are sampled and pixel values are blended into the frame buffer. The texture sampling is either trilinear if the 3D texture mapping hardware is available, or bilinear if only 2D texture are supported. Multi-pass, shading, image quality and performance improvements have been studied by Westermann and Ertl (1998) and Meißner et al. (1999, 2002).

The most often used volume visualization algorithm for the production of high-quality images is **ray-casting**. This image-order technique was presented by Kajiya and Von Herzen (1984a) and formalized by Levoy (1988). Rays are cast from the view-point through the view-plane into the volume. While the ray traverses the volume, data is sampled with interpolation and blended using Equation 2.1.

To reduce the high computational overhead of integrating viewing rays through the volume, Rottger et al. (2000) presented **pre-integration**. This technique performs the ray integration through a texture lookup. Pre-integrated classification assumes a piecewise linear, continuous reconstruction of the scalar field along a viewing ray. With this assumption, the contribution of a ray segment can be pre-computed in a pre-processing step. Therefore, since ray segments can be pre-computed before-hand, fewer samples along the ray are required to reconstruct the view image. This technique has been further improved by Engel et al. (2001); Roettger and Ertl (2002); Kye et al. (2008) and Guetat et al. (2010).

For a more detailed review of the state of the art in volume rendering techniques, further detailed descriptions and research references can be found in books and surveys by Elvins (1992); Kruger and Westermann (2003a); Hadwiger et al. (2006); Weiskopf (2007) and Zhou et al. (2018). Among the presented techniques, this thesis is based on the well-known direct method: *volume ray-casting.*

2.2.1 Volume Ray-casting

The ray-casting technique was presented as an special case of the generalized ray-tracing technique (Kajiya and Von Herzen, 1984a). Rays are cast from the viewer position through the volume data and sampled at regular intervals along these rays. Each sampled point is blended by accumulating colour and opacity in front-to-back or back-to-front order. Ray-casting differs from ray-tracing in that it does not spawn secondary rays like ray tracing does. Initial approaches and research work was oriented towards CPU ray-casting (Levoy, 1988; Tuy and Tuy, 1984; Upson and Keeler, 1988; Levoy, 1990).

The utilization of ray-casting became more popular when Kruger and Westermann (2003b) used the graphics hardware computational power and presented a GPU-based ray-casting algorithm, achieving real-time frame rates with nowadays consumer hardware.

Several modifications have been addressed to gain performance with ray-casting, such as early ray termination (Kruger and Westermann, 2003b), which finishes the accumulation process when the contribution of the sample is irrelevant; and empty space skipping (Li et al., 2003), which optimizes the ray traversal through empty regions. Generally, ray-casting is a technique that can obtain higher quality renderings than other direct methods. The flexibility and performance of ray-casting against slice-based algorithms were introduced by Stegmaier et al. (2005) when they presented a single-pass volume rendering framework for GPU-based ray-casting.

Other research works have focused in the acceleration of the ray traversal and in the space subdivision of the volume data for out-of-core rendering of large datasets. The survey by Beyer et al. (2014) covers the acceleration techniques for the rendering of large datasets.

2.2.2 Non-photorealistic rendering

Volume visualization can be enhanced by visual effects or feature extraction (Yang et al., 2014; Zhou et al., 2015). Originally conceived for traditional image rendering and artistic effects, illustrative and non-photorealistic renderings can be adapted for volume rendering. They enhance the perception for features within the volume data. Decaudin (1996) introduced cartoon style rendering for 3D scenes and Gooch et al. (1998) presented a tone-based non-photorealistic lighting model for automatic technical illustration. Applied to volume rendering, a set of non-photorealistic styles were collected by Ebert and Rheingans (2000). Cluster-based GPU hardware accelerated non-photorealistic renderings were studied by Lum and Ma (2002).

More recently, in the illustration of volume data by means of transfer functions, Bruckner et al. (2005) and Bruckner and Gröller (2005, 2007) presented a novel technique to apply illustrative styles. In this regard, Ljung et al. (2016) produced a state of the art about transfer functions applied in direct volume rendering.

The Medical Working Group of X3D (Web3DConsortium, 2014) defined a volume rendering component specification with support of non-photorealistic renderings for the declarative scene definition of volume rendering content. Polys and Wood (2012) and Polys et al. (2011a, 2013b) evaluated the specification in several domains and researched its use in immersive environments.

2.3 Volume rendering in the Web platform

Initially, when there was no access to accelerated GPU graphics in browsers, third-party plug-ins where used for rendering 3D graphics. These plug-ins were not provided by browser vendors. But after installation and configuration, they could access the same APIs as other desktop approaches and applications, like OpenGL or DirectX APIs. However, their support were discarded over time due to browsers third-party software sand-boxing policies and for security reasons.

With this situation in mind, third-party plug-ins were not a viable solution. Approaches to bring volume rendering to the web ecosystem used either *i*) rendering on the server side or *ii*) rendering on the client side with WebGL.

Server side rendering is an effective way of rendering big volumetric datasets using high-performance servers (Kaspar et al., 2013; Gutenko et al., 2014)). However, this approach is not always suitable for real-time applications, due to the connection lag between client and server. Also, it is not a scalable solution and requires more investment on the server computational power and network infrastructure.

With the arrival of the WebGL API to web browsers, accelerated GPU graphics within the browser context was enabled. In recent years, there has been a development of web-oriented real-time 3D graphics engines motivated by the objective of making easier the creation and delivering of Web-based games and interactive content. Several popular frameworks already take advantage from the latest capabilities of JavaScript, HTML5 and WebGL, enabling the interactive visualization of traditional polygonal meshes on the Web e.g. *Three.js* (Cabello, 2018), *Babylon.js* (Catuhe et al., 2014) and *OSG.js* (Pinson, 2014).

Volume rendering is a computationally expensive rendering technique that can be implemented with different algorithms. However, the translation of the volume visualization to an ubiquitous platform such as the Web is a challenge. Currently, among the 3D graphics frameworks available for the Web, only *goXTK* (Hähn et al., 2012), *X3DOM* (Arbelaiz et al., 2016b), *Three.js* (Cabello, 2018) and *VTK.JS* (Kitware, 2019) support volume rendering for scientific data visualization.

Initial approaches used indirect methods, polygonal surface data were extracted from the volumetric data, until Congote et al. (2011) presented a WebGL based real-time volume rendering ray-casting algorithm that enabled client-side rendering in the browser. Their WebGL volume ray-casting algorithm was based on Kruger and Westermann (2003b) multi-pass approach. They also used this approach for the visualization of air quality models combined with GIS data (Congote et al., 2012) and weather radar data visualization (Moreno et al., 2014).

Later, Mobeen and Feng (2012b,a); Movania and Lin (2012); Movania et al. (2014) improved the multi-pass ray-casting algorithm with a single-pass version based on the algorithm presented by Stegmaier et al. (2005) and compared it with a texture slicing method in mobile devices.

Also, with WebGL's ubiquitous characteristic, Noguera et al. (2012) and Noguera and Jiménez (2012, 2016) have analysed volume rendering on mobile devices with the OpenGL ES 2.0 API, the same API in which WebGL 1.0 API is based on. They have also compared ray-casting with the texture slicing rendering technique for volume rendering with WebGL in mobile devices.

Hybrid approaches of server and client rendering have been studied. Wangkaoom et al. (2015) and Chandler et al. (2015) have used WebGL as a light rendering client by using a client and server based hybrid rendering solution.

Arbelaiz et al. (2016b) presented a volume rendering component for X3DOM based in the approach presented by Congote et al. (2011) and Congote (2012). This thesis focuses solely in the client side volume rendering with WebGL based ray-casting. X3DOM is a DOM-based implementation of X3D (Fraunhofer IGD, 2014) that enables declarative X3D in the Web. Later, Arbelaiz et al. (2016a) extended the WebGL ray-casting algorithm for the hybrid rendering of surface and volume data.

Additional works have addressed different challenges related to the volume rendering component, presenting contributions and advances in the interaction and exploration of volumetric datasets. Yang et al. (2015) have contributed to X3DOM for weather data visualisation in conjunction with terrain data. Tabor et al. (2018) have also used this component to create a zebrafish brain browser tool that facilitates the design of intersectional genetic experiments and for the study of brain circuit-mapping. Arbelaiz et al. (2017a,c) have explored the use of DICOM medical data exchange format in combination with X3DOM for medical volume visualization.

This component implementation provided by Arbelaiz et al. (2016b) offers X3D's volume visualisation reproducible and declarative features and it has been the reference to obtain feedback from the community (X3DOM Community, 2015a, 2016b,a, 2015b, 2017). The feedback received was taken into account to define new extension proposals for the X3D standard (Arbelaiz et al., 2017b).

Recently, the WebGL 2.0 API has been officially released to the public. Its support by modern browsers and platforms is not yet ubiquitous as its predecessor. However it exposes new hardware capabilities that can be exploited for volume rendering (Mwalongo et al., 2018; Lesar et al., 2018).

Figure 2.1 shows a time-line with the presented contributions related to WebGL based DVR (above) and some of the community requested enhancements (below). The time-line is divided in three periods: *i)* initial contributions of DVR algorithms applicable in the Web (orange area); *ii)* elapsed time in which initial developments of the X3DOM volume rendering component were made (green area) and *iii)* contributions to web based DVR, including contributions to the X3DOM volume rendering component in response to community feedback (blue area).

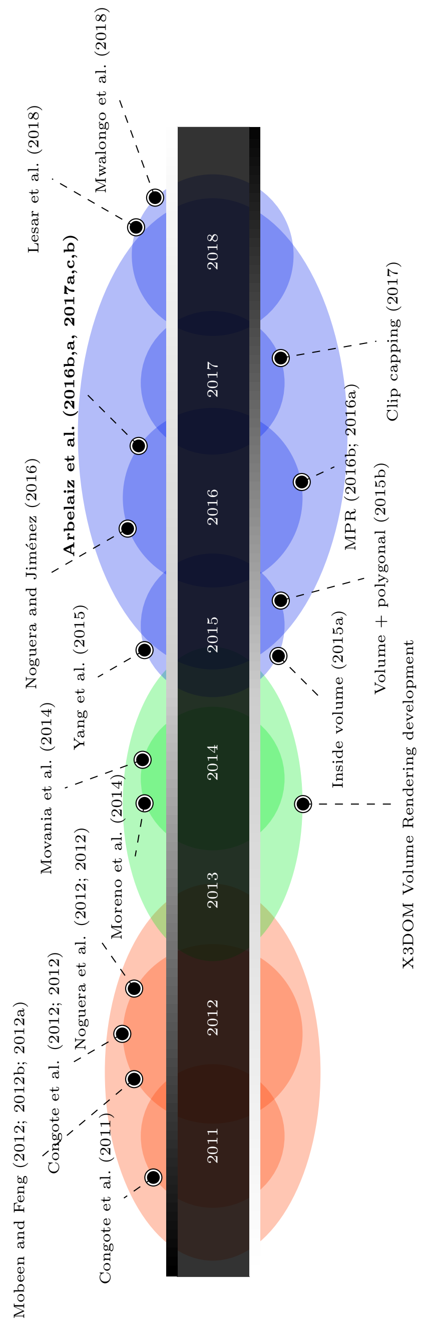

Mobeen and Feng (2012; 2012b; 2012a)
Congote et al. (2011)
Congote et al. (2012; 2012)
Noguera et al. (2012; 2012)
Moreno et al. (2014)
Movania et al. (2014)
Yang et al. (2015)
Noguera and Jiménez (2016)
Arbelaiz et al. (2016b,a, 2017a,c,b)
Lesar et al. (2018)
Mwalongo et al. (2018)
2011
2012
2013
2014
2015
2016
2017
2018
X3DOM Volume Rendering development
Inside volume (2015a)
Volume + polygonal (2015b)
MPR (2016b; 2016a)
Clip capping (2017)

Figure 2.1: Time-line of contributions to WebGL-based volume rendering and community feedback.

Chapter 3

Web compatible volume data texture

The Web has become a valuable entry point to access information and interactive multimedia content. In particular, applied to 3D real-time rendering, web browsers postulate as a new opportunity to extend 3D advanced graphics virtues and interactive experiences to all users. These users demand access to content from a multitude of different devices, and therefore, this new paradigm must be taken into account.

From an ubiquitous perspective, the Web is the only platform that facilitates to reach the highest number of devices and operating systems due to the utilisation of common standards supported by multiple web browsers. However, this benefit does not come without a few caveats.

WebGL is a W3C proposed standard that is based in the OpenGL ES 2.0 API by Khronos (Khronos, 2016). Although WebGL is based on an API originally created for embeded devices (OpenGL ES), WebGL does not support all OpenGL ES features. This situation was intentionally done by the W3C consortium in order to ensure major compatibility and vendor support among graphics architectures and devices.

On one hand, it can be assumed that 3D ubiquitous computer graphics is possible using the WebGL API. On the other hand, there are technical constraints that must be solved in order to create advanced rendering algorithms. The representation of volumetric datasets using ray-casting methods is an example of such rendering algorithms.

This chapter describes methodologies behind the web compatible volume data structures and how this thesis contributes to extend their usage to complex volume rendering scenarios. The presented methodology proves its compatibility with the web platform with new applications, such as 4D volume rendering and improves its performance from previous approaches.

The chapter is structured as follows. The volumetric data problem in the web is introduced at Section 3.1. Afterwards, Section 3.2 describes the contributions to the web compatible volume data structure.

3.1 Volumetric data problem in WebGL

Due to the high computational power required for visualisation of volume data, volume rendering is mainly applied in the scientific and engineering domains. Current approaches try to propose computationally efficient algorithms, but one of the core problems of volume rendering is the amount of memory samples required to render the whole volume. Specially in direct volume rendering approaches, a high number of the voxels must be sampled in real-time in order to reconstruct the final 3D render accurately. This implies that the problem is memory bound.

Furthermore, volume data is usually an approximation of a real phenomena discretized into a finite number of voxels. Essentially, volume data is a discretization of a continuous signal that can be infinitesimally sampled. Thus, the amount of voxels or the volume grid size defines the resolution of the model. For very accurate acquisitions coming from new MRI and CT scan technologies, they can generate too large volume datasets that would require a large quantity of memory, in the order of several gigabytes (GB) or even terabytes (TB) of data.

When the volume is discretized into a grid of scalar values, there is an step value difference between neighbourhood voxels. This step must be considered when sampling the volume for visualization, as it can produce a visual artefact in the rendering outcome similar to aliasing. An interpolation phase can be used in order to smooth the transition of the value changes, when sampling between voxels data.

In real-time volume rendering, the memory problem is solved by using specialized hardware, i.e, graphics processing unit (GPU). Current GPUs have texture units, a specialized image data memory storage, designed to obtain better performance with sequential and non-sequential memory access thanks to additional specialized cache memory and an efficient scheduling to hide memory latency. In addition to this, they offer hardware based interpolation when fetching texture data. Modern GPUs have both 2D and 3D texture units, even in mobile devices.

The hardware capabilities of GPUs are exposed by graphics APIs (DirectX, OpenGL, Metal, etc.) or General-Purpose computing (GPGPU) APIs like OpenCL and CUDA. With the exception of WebCL at certain browsers in concrete versions, when the web platform is used as a medium for ubiquitous volume rendering, WebGL is the only graphics API that allows to access the underlying GPU hardware.

The WebGL API specification is based on the OpenGL ES API, the embedded version of OpenGL, which offers a graphics programmable pipeline supported in a great number of devices. Unfortunately, to favour the adoption of the standard, WebGL 1.0 does not support all the features of its reference, OpenGL ES 2.0. For volume data, the WebGL 1.0 API does not support the 3D texture capabilities present on the underlying GPUs. The lack of support of 3D textures is a major drawback to allow the visualisation of volume data in the web platform. A solution for a web compatible structure was firstly presented by Congote et al. (2011) and later extended to larger datasets by Noguera and Jiménez (2012).

3.1.1 Congote et al. contribution: tiled volume texture

Congote et al. (2011) proposed a solution to emulate 3D textures. Their approach is divided into two parts: (a) The first part consists in the generation of a 2D texture that contains the volume data. This is created by tiling each slice (8-bit single channel) that composes the volume data in one axis (Z) direction into a matrix configuration in row-major order. In this manner, the 3D volume data is transformed into a 2D representation (2D texture) that can be managed with WebGL. (b) The second part allows to fetch the volume data in the fragment shader at the rendering stage by sampling the previously generated 2D texture.

Figure 3.1 shows how each slice that composes the volume data is tiled into a matrix configuration.

In order to maximize the amount of data that can be stored in a single 2D texture unit, the generated texture must match the squared size dimensions supported by GPUs: 4096×4096, 2048×2048, etc. When tiling the slices of the volume data, the number of slices, their dimensions and the supported texture resolution size of the targeted GPU must be taken into account. GPU resources will be optimally used when the width and height of the generated texture are equal, the smallest possible dimension is used and at the same time, the required amount of data is stored into the texture.

Once the tiled volume data is loaded into the GPU as a 2D texture, it is sampled in the fragment shader using Equation 3.1 to get the actual 3D volume data value. This equation uses GLSL 1.10 shader built-in math

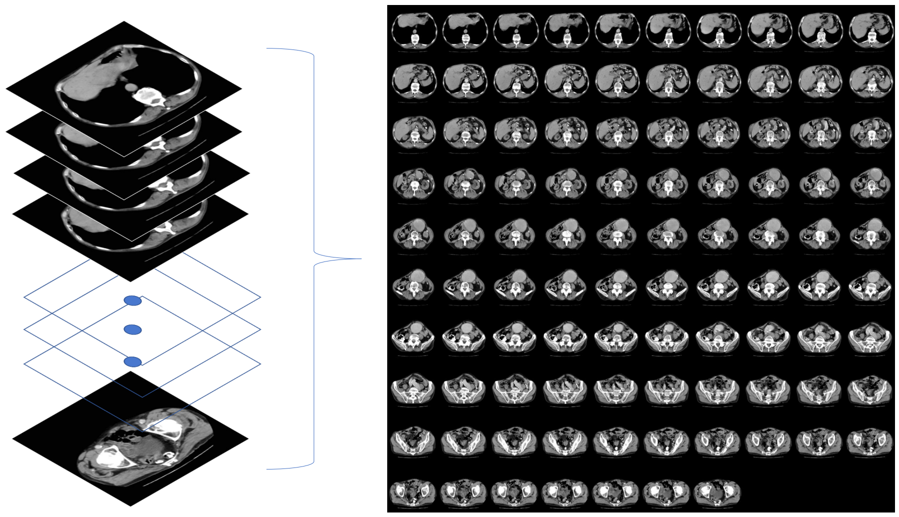


Figure 3.1: Tiling slices into a matrix configuration in row-major order.

methods: the matrix configuration is defined by n_s (number of slices), n_x (number of slices over X direction), and n_y (number of slices over Y direction).

$$
\begin{aligned}
s_0 &= \lfloor z \cdot n_s \rfloor \\
s_1 &= s_0 + 1 \\
\vec{d_1} &= \left[\mathbf{fract}\left(\frac{s_1}{n_x}\right),\ \frac{\lfloor \frac{s_1}{n_x} \rfloor}{n_y} \right] \\
\vec{d_2} &= \left[\mathbf{fract}\left(\frac{s_2}{n_x}\right),\ \frac{\lfloor \frac{s_2}{n_x} \rfloor}{n_y} \right] \\
\vec{t_1} &= \vec{d_1} + \left[\frac{x}{n_x},\ \frac{y}{n_y} \right] \\
\vec{t_2} &= \vec{d_2} + \left[\frac{x}{n_x},\ \frac{y}{n_y} \right] \\
\vec{v_1} &= \mathbf{texture2D}(\vec{t_1}) \\
\vec{v_2} &= \mathbf{texture2D}(\vec{t_2}) \\
r &= \mathbf{mix}(\vec{v_1}_x, \vec{v_2}_x, (z \times n_s) - s_0)
\end{aligned}
\tag{3.1}
$$

For each volume sample (x, y, z) (3D coordinate), two texture fetches of consecutive slices are performed. Those values are linearly interpolated in the Z axis to obtain the final volume data value. In this manner, a trilinear interpolation for each volume sample is emulated since texture sampling in a 2D texture unit is automatically bilinearly interpolated by hardware. Multiple samples and a interpolation are required to generate a smooth visualisation of the discretized volumetric data.

3.1.2 Noguera et al. contribution: large volume data

In order to accommodate bigger datasets, Noguera and Jiménez (2012) extended the approach by Congote et al. (2011) using multiple colour channels or as an alternative, multiple atlases in multiple texture units.

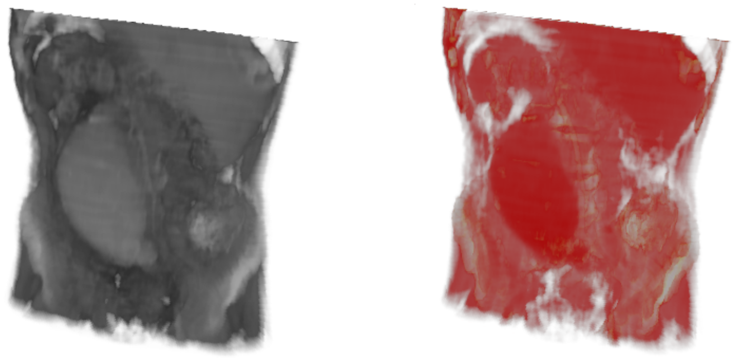
On their first method, they rearrange slices into a matrix configuration in one colour channel using a row-major order until the dimension criteria for the given texture atlas is met. Then, they continue rearranging slices in the next colour channel also in row-major order, repeating this process per colour channel until all colour channels are filled with volume data. This method allows to store up to 4 times as much data in a single ($RGBA$) texture.

As an alternative, on their second method, they use multiple single colour channel (R) textures of same dimensions and store them into multiple sequential texture units. Then, in the fragment shader they sample the required texture unit accordingly with the method presented by Congote et al. (2011).

3.2 A new proposal: ImageTextureAtlas

The tilling approach to compose a 2D volume data image allows to implement ray-casting algorithms in web browsers. However, both contributions from Congote et al. (2011) and Noguera and Jiménez (2012) can be extended to accommodate more advanced volume renderings such as illustrative, non-photorealistic and segmented visualizations.

Different structure variations and algorithms are needed to accommodate such renderings. The data structure that includes all these variations by composing a 2D texture of volume data slices is called *ImageTextureAtlas*. In this section, the different contributed types of *ImageTextureAtlas*es are described.



(a) Without colour classification (b) With colour classification

Figure 3.2: Comparison of the volume rendering with the `aorta` dataset after colour classification and without colour classification.

3.2.1 Coloured volume data

Usually volume data represents the measured scalar data acquired from a scan acquisition or simulation. These data does not describe any additional property such as colour, just the values of the measured property. In order to aid the user to discern the value changes and patterns within the data, colour can be applied to illustrate the volume rendering outcome.

Figure 3.2 shows a direct volume rendering visualisation of the `aorta` dataset. In Figure 3.2a no colouring is applied, while in the Figure 3.2b, colour has been used to illustrate the volume data.

The rendering in Figure 3.2 shows how colouring in function of the different density values at each voxel allows to easily discern between tissue and bones. The mapping of colour values to data values (classification) can vary in function of the use case. Additionally, it can also be applied globally or to localized regions of the data. This functionality will later be shown in Subsection 3.2.3.

To store coloured volume data, a *RGBA* data structure is needed. In the desktop platform with full OpenGL support, a *RGBA* 3D texture can be easily declared with the `glTexImage3D` API function. Listing 3.1 shows the input arguments required for the declaration of a multi-channel *RGBA* 3D texture. Each voxel is composed from a triplet of 8-bit unsigned values that represents a colour value.

Listing 3.1: 3D texture declaration in OpenGL for RGBA 3D texture.

```
glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA8, WIDTH, HEIGHT
    , DEPTH, 0, GL_RGBA, GL_UNSIGNED_BYTE, texels);
```

However, there is no such equivalent function in the WebGL 1.0 API. Instead, the approach presented by Congote et al. (2011) can be extended to be used with multiple channels. In this case, rearranging the coloured volume data slices (*RGBA*) in row-major order a *RGBA ImageTextureAtlas* is generated. This step is performed before computing the rendering of the volume. Instead of using a single colour channel *ImageTextureAtlas* for each voxel of the volume data, multiple colour channels are used. Figure 3.3 shows a representation of this process.

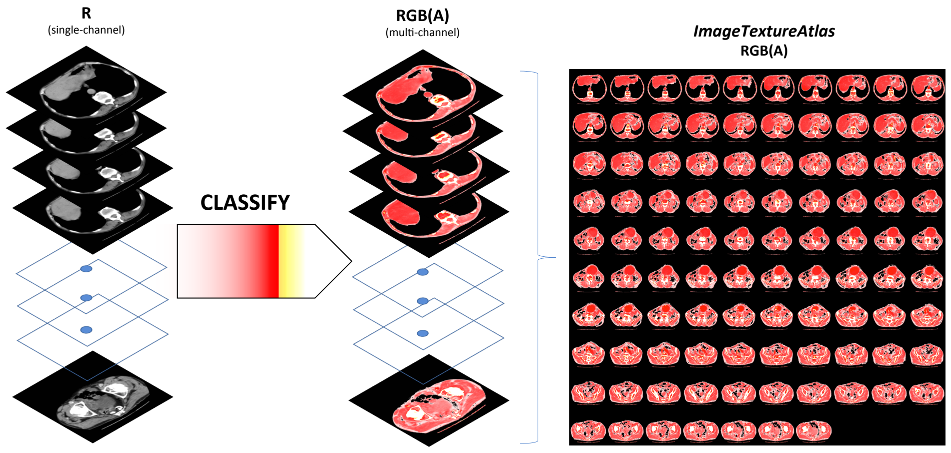


Figure 3.3: Pre-classification to compose a coloured *RGBA ImageTextureAtlas*.

In contrast to Equation 3.1, where only one colour channel is used when sampling the texture, two samples of a four component vector are obtained ($\vec{v}_1, \vec{v}_2$) at the coloured *ImageTextureAtlas*. These vectors represent pre-classified volume data values. They are linearly interpolated to compute the output value $\vec{r}$ (see Equation 3.2).

$$
\begin{aligned}
s_0 &= \lfloor z \cdot n_s \rfloor \\
s_1 &= s_0 + 1 \\
\vec{d_1} &= \left[\mathbf{fract}\left(\frac{s_1}{n_x}\right), \ \frac{\lfloor \frac{s_1}{n_x} \rfloor}{n_y} \right] \\
\vec{d_2} &= \left[\mathbf{fract}\left(\frac{s_2}{n_x}\right), \ \frac{\lfloor \frac{s_2}{n_x} \rfloor}{n_y} \right] \\
\vec{t_1} &= \vec{d_1} + \left[\frac{x}{n_x}, \ \frac{y}{n_y} \right] \\
\vec{t_2} &= \vec{d_2} + \left[\frac{x}{n_x}, \ \frac{y}{n_y} \right] \\
\vec{v_1} &= \mathbf{texture2D}(\vec{t_1}) \\
\vec{v_2} &= \mathbf{texture2D}(\vec{t_2}) \\
\vec{r} &= \mathbf{mix}(\vec{v_1}, \vec{v_2}, (z \times n_s) - s_0)
\end{aligned}
\tag{3.2}
$$

From a performance point of view, as the colour is already encoded in the volume data atlas, no additional colour mapping operations are required at the fragment shading stage. Therefore, there is no additional computation time penalization. This pre-classified coloured *ImageTextureAtlas* has the advantage of being an efficient approach to render of illustrated volume data. Additionally, it is a good approach in cases where the colour mapping is not intended to be changed interactively in a real-time rendering. Interactive colour mapping approaches and more advance renderings will be presented in the following chapters.

3.2.2 Gradient volume data

Surface normals are used to compute the incidence angle of a light vector from a light source. This allows computing advanced rendering styles, in order to enhance features, illuminate or illustrate the volume data.

For the surface representation using polygonal mesh data, a surface normal at given point is a vector perpendicular to the tangent plane to the surface in that point. These normals can be computed for each face or for each vertex of the mesh, whereas for the volume data, multiple iso-surfaces can be considered and the normal vectors are associated per voxel, i.e, like in a 3D vector field per point in space. These normals are obtained by computing the derivative of the original volume data signal function. Since the volume data is discretized and stored as image data, the derivative of the data is approximated by the gradient computation using a neighbourhood operator. As a result, a three component (x, y, z) vector is generated for each voxel.

To avoid the computation of gradient data in real-time, it can be computed in a pre-processing step (offline). Additionally, using this approach, different gradient operators such as sobel, gauss, etc. can be used. These methods are not suitable for real-time rendering as a high number of neighbour voxels are required to be sampled.

At the desktop platform with full OpenGL support, a *RGB* 3D texture could be used, as previously shown in Listing 3.1, to store the gradient data. However, such support is not available in the WebGL 1.0 API. As a solution, the computed gradient data is encoded into a multi-channel *RGB ImageTextureAtlas*.

The computed gradient data is encoded into a multi-channel *RGB ImageTextureAtlas*. Each gradient vector is mapped to colour channels ($R : X$, $G : Y$, $B : Z$) using a row-major order to compose the matrix of slices that represents the *ImageTextureAtlas*. Figure 3.4 shows a single slice of volume data and its computed gradient slice.

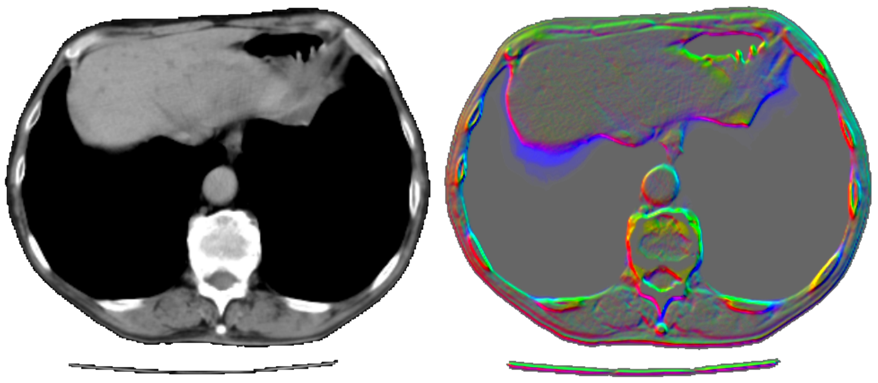

(a) Volume data slice (b) Gradient data slice

Figure 3.4: Comparison of volume data slice of the `aorta` dataset and a colour-enhanced version of a computed gradient data slice (background removed).

The alpha channel can also be used to store the gradient magnitude, i.e, the length of the gradient vector at each voxel. For this case, a *RGBA ImageTextureAtlas* is generated.

On the visualisation process, every time the volume data is sampled at given 3D texture coordinate, both the volume data and the gradient data *ImageTextureAtlas*es are sampled on their respective texture units. In the first case, at the fragment shader, Equation 3.1 is used to extract the volume data value. In the later case, for the gradient data the same equation is used, but with modifications that take into consideration the vectorial nature of the information, as presented in Equation 3.2 for the coloured *ImageTextureAtlas*.

After obtaining the gradient value, since the texture stores the gradient values as positive values in 8-bit [0-255] range, they must be transformed back into the [-1,1] range. For that purpose, the GLSL code at Listing 3.2 can be used.

Listing 3.2: Sampling gradient vector from texture.

```
vec4 getNormalFromTexture(sampler2D sampler, vec3 pos) {
  vec4 n = (2.0 * sampleImageTextureAtlas(sampler, pos)-1.0);
  return vec4(normalize(n.xyz), length(n.xyz));
}
```

Therefore, the gradient vector is obtained performing an additional sampling (see Equation 3.1) and transform operation (see Listing 3.2). A limitation of this approach is the extra GPU memory space required, as gradient data uses three times more space than 8-bit single-channel scalar data.

An alternative to store the gradient volume data into a texture is to compute this data in real-time at the fragment shader. Equation 3.3 shows the central-differences neighbourhood operator ($\bigtriangledown$) usually employed in real-time volume rendering. It shows that for each 3D sample, six additional neighbour texel fetches are required.

$$\bigtriangledown f(x,y,z) = \begin{cases} \frac{f(x+1,y,z)-f(x-1,y,z)}{2}, \\ \frac{f(x,y+1,z)-f(x,y-1,z)}{2}, \\ \frac{f(x,y,z+1)-f(x,y,z-1)}{2} \end{cases} \tag{3.3}$$

For the *ImageTextureAtlas* approach, each 3D texture fetch (*sampleImageTextureAtlas*) requires two 2D texture fetches. The Listing 3.3 shows a GLSL implementation of Equation 3.3.

Listing 3.3: 3D gradient computation in the fragment shader.

```
vec4 gradient(sampler2D sampler, vec3 pos){
  float v0 = sampleImageTextureAtlas(sampler, pos + vec3(↩
     offset.x, 0, 0)).x;
  float v1 = sampleImageTextureAtlas(sampler, pos - vec3(↩
     offset.x, 0, 0)).x;
  float v2 = sampleImageTextureAtlas(sampler, pos + vec3(0, ↩
     offset.y, 0)).x;
  float v3 = sampleImageTextureAtlas(sampler, pos - vec3(0, ↩
     offset.y, 0)).x;
  float v4 = sampleImageTextureAtlas(sampler, pos + vec3(0, ↩
     0, offset.z)).x;
  float v5 = sampleImageTextureAtlas(sampler, pos - vec3(0, ↩
     0, offset.z)).x;
  vec3 grad = vec3(v0-v1, v2-v3, v4-v5)*0.5;
  return vec4(normalize(grad), length(grad));
}
```

The forward-difference method fetches only the next neighbour voxel in the ray direction, which translates in half amount of texture fetches (see Equation 3.4).

$$\triangledown f(x,y,z) = \begin{cases} \frac{f(x+1,y,z)-f(x,y,z)}{2}, \\ \frac{f(x,y+1,z)-f(x,y,z)}{2}, \\ \frac{f(x,y,z+1)-f(x,y,z)}{2} \end{cases} \tag{3.4}$$

Figure 3.5 shows the volume rendering visualization of the `aorta` dataset illustrated with the gradient vector encoded as *RGB* colours.

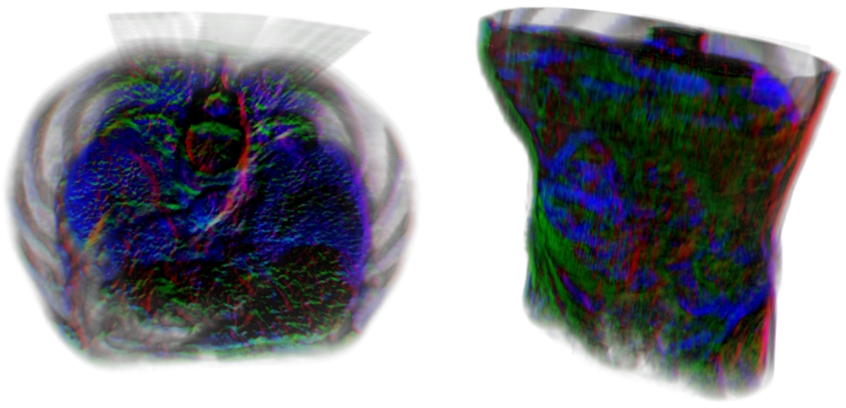


Figure 3.5: Volume rendering of the `aorta` dataset illustrated with the computed gradient data as colour.

The use of the gradient texture depends on the targeted hardware. In

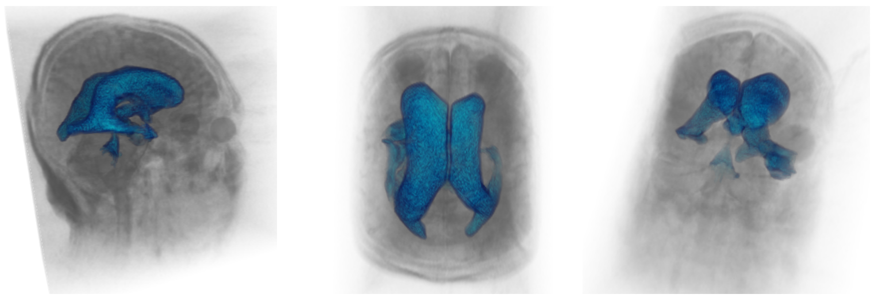


Figure 3.6: Segmented volume rendering of the `Head MRI` dataset from different points of view.

some cases, the latency of additional texture fetches will be hidden along the compute operations by GPU instruction scheduling. In other cases, the high memory bandwidth and transfer time required for the gradient *ImageTextureAtlas* will be a burden for the performance.

3.2.3 Segmented volume data

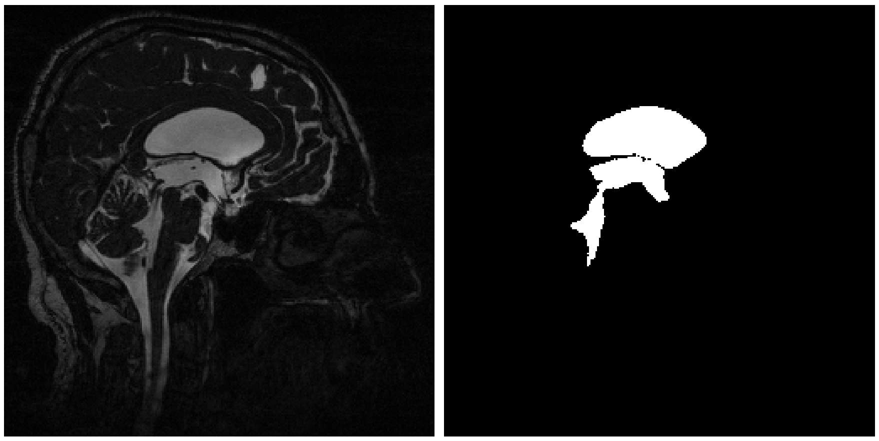
There are use cases where regions within the volume data are required to be rendered in a different manner. For instance, in medicine, tissues or bones can be segmented based in the acquired density values to apply different colour mappings to each region.

For this purpose, firstly, each voxel in the volume data must be identified into a given region. Once regions of data are assigned, in the rendering process this information can be used to enhance the region of interest or interactively interact with the visualisation in separate regions of the data. Figure 3.6 shows a segmented visualisation of the ventricles of the brain with the `Head MRI` dataset.

At the desktop platform with full OpenGL support an additional 3D texture can be used to store the segmented data. In function of the number of regions, different data types (bool, uint8, uint16, etc.) could be used to minimize the required memory space. Listing 3.4 shows an example declaration for such data structure in OpenGL.

Listing 3.4: 3D texture declaration in OpenGL for segmented data

```
glTexImage3D(GL_TEXTURE_3D, 0, GL_R8, WIDTH, HEIGHT,
    DEPTH, 0, GL_R, GL_UNSIGNED_BYTE, texels);
```



(a) Volume data slice (b) Segmented data slice

Figure 3.7: Comparison of a volume data slice and the segmented binary mask of the same slice from the `Head MRI` dataset.

Such declaration cannot be made in WebGL 1.0. However, an additional *ImageTextureAtlas* of the same dimensions can be used to store the segmented data in a compatible manner.

The segmented data is a mask that is applied during the rendering. In the web platform, two type of masks have been applied:

1. **Binary mask**: For each voxel a binary value (true or false) is assigned to specify if the given voxel belongs to the region of interest or not.

2. **Identifier mask**: For each voxel an scalar value is given. This identifier specifies to which region belongs the voxel so that, at the rendering process it will mapped accordingly.

Identifiers are assigned into [0-255] range 8-bit single-channel images. Figure 3.7 shows an slice of the `Head MRI` dataset on the left and the extracted binary mask on the right, obtained from the Volvis `Head MRI` dataset (University of Tübingen WSI/GRIS, 2014).

The extracted segmented slices are composed into a matrix configuration in row-major order to compose an *ImageTextureAtlas* (see Figure 3.8).

In the rendering process, once the segmented data is loaded into GPU memory, the segmented data is sampled along the volume data with the same 3D texture coordinates. In function of the identified region, the rendering will vary, as it will be described later on, in Chapter 7.

The process of region extraction or segmentation algorithms is out of the scope of this thesis. In the medical domain, segmentation is an important

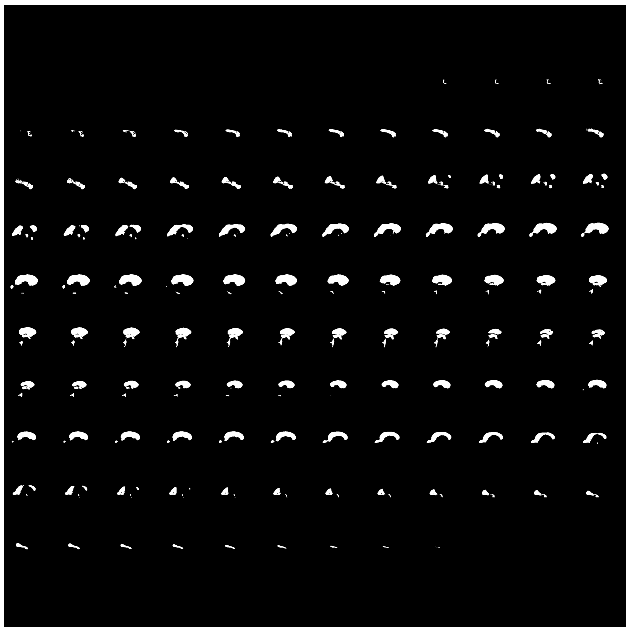


Figure 3.8: Segmented *ImageTextureAtlas*. Binarized mask of the ventricles of the brain from the `Head MRI` dataset

.

research topic, and several tools and applications that can be used to generate the segmented data. The output of these tools can be converted into the proposed *ImageTextureAtlas*. In Chapter 9 a segmentation pipeline is presented to generate a segmented *ImageTextureAtlas*.

Figure 3.9 shows the ubiquitous segmented visualization of different volumetric datasets which use the proposed structure.

3.2.4 Time-varying volume data (4D)

In natural phenomena 3D data also changes over time. Therefore, data should not be considered as static. Data evolves over time is also known as time-varying data. Thanks to advances in MRI and CT technology, capturing of sequential scans at higher speed is now possible, and therefore, datasets of volume data that evolves over time (4D) can be considered for study. Currently, applications in the medical domain target 4D ultrasound imaging acquisition or, in the geosciences domain, 4D data simulations in a long period of time of a system are studied (Ho and Jern, 2008).

In the desktop platform there is no 4D texture structure available. In-

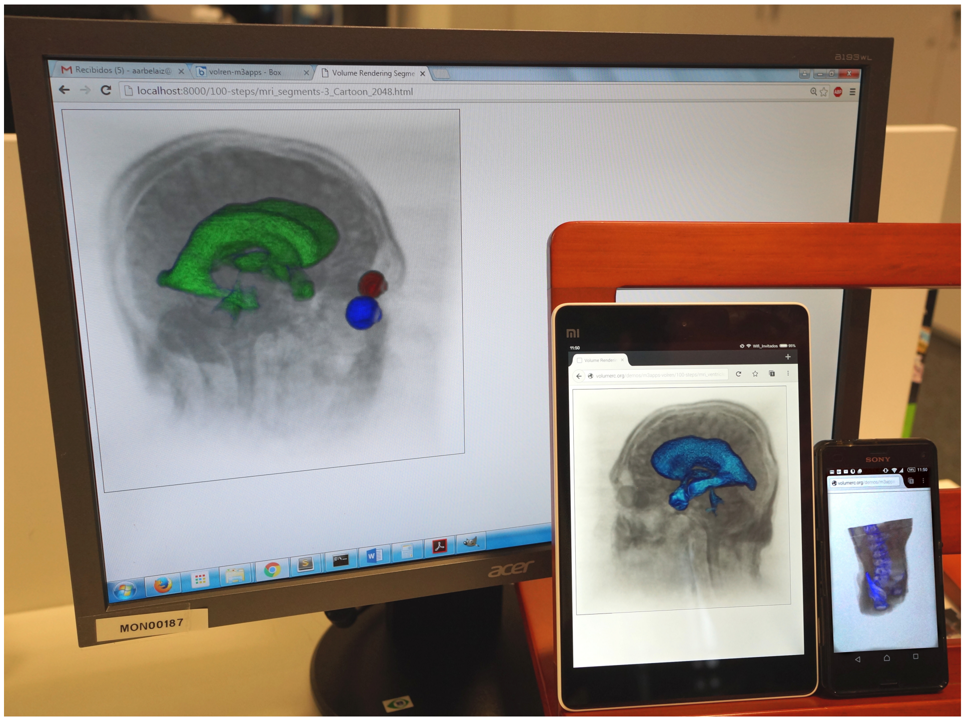


Figure 3.9: Ubiquitous segmented volume visualization of the `Head MRI` and `aorta` datasets in multiple devices.

stead, it is represented as an array of 3D textures. To perform the rendering, 3D textures must be uploaded on demand to the GPU memory and this can be a complex process to handle. This approach is not applicable in the Web platform. However, there is a solution that exploits the HTML5 capabilities of web browsers.

The video container is natively supported by modern web browsers and video files can be easily distributed from a web server. A new extended version of *ImageTextureAtlas* is used to accommodate the time (4D). In this case, 4D volumetric data is stored as a video instead of an static *ImageTextureAtlas*

Figure 3.10 illustrates the composition of the *ImageTextureAtlas* video container, where the fourth axis (time) hold the frames of the video that will be reproduced.

The video generation must be performed in the server side or in a preprocessing step. This approach exploits the native video reproduction and 2D canvas API features of HTML5 in modern browsers. An atlas for each

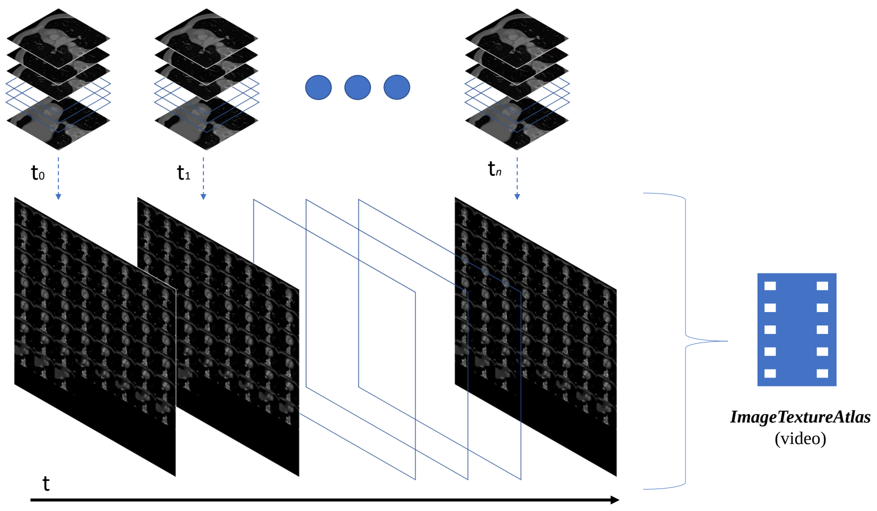


Figure 3.10: Illustration of the composition of the time-varying *ImageTextureAtlas* video. A data structure to represent 4D volume data in the web.

time step of the 4D volume data is created. Then, all the atlases are converted and encoded into a video which will play in a continuous loop.

Listing 3.5 shows the definition of the video declaration in HTML5. This atlas sequence is being played in the background hidden to the user.

Listing 3.5: HTML5 video declaration for an *ImageTextureAtlas*.

```
<video autoplay loop style='width:2048px;height:2048
   px;display:none'>
  <source src='atlas_video.mp4' type='video/mp4'>
  </source>
</video>
```

The hidden video is linked through JavaScript with a dynamic *ImageTextureAtlas*. A 2D canvas of the same resolution of the video is created and its content is updated with the canvas API using a fixed timer (refresh time). Since the 2D canvas data can be transferred to the GPU as a texture, its continuous changes over time are reflected in real-time in the visualization.

There are some considerations to take into account like the potential loss of accuracy due to the employed video encoding or the refresh time of the texture data. Nevertheless, this approach demonstrates the current capabilities of the Web technological stack and the proposed web-centred

ImageTextureAtlas approach.

3.2.5 Single-channel *ImageTextureAtlas* resolution discussion

There is one consideration that must be taken into account regarding the presented approach: downscaling needed to fit the atlas within the texture size limit of the client device GPU. Typically, for in-core GPU volume rendering, resolution of datasets vary from $128 \times 128 \times 128$ to $512 \times 512 \times 512$. Bigger datasets will require downscaling or out-of-core algorithms as researched by Gobbetti et al. (2008) and Crassin et al. (2009).

Using WebGL Stats (Bösch, 2019), it is safe to assume that in 2018 the majority of desktop computers support 8192×8192 texture size *ImageTextureAtlas*es, and 75% among them support a 16384×16384 size atlas. Whereas in mobile devices, the established threshold is set on a 4096×4096 texture size.

Table 3.1 shows the texture size required to store the volumetric data into an *ImageTextureAtlas*. Common resolution of volumetric datasets are taken as a reference (columns): $128 \times 128 \times 128, 256 \times 256 \times 256$ and $512 \times 512 \times 512$. Rows indicate the 2D texture sizes supported by GPU hardware. If the dataset fits the targeted texture resolution without downscaling a check symbol (✓) is used, otherwise a percentage of the overall downscaling required is given (size reduction).

	Dataset		
Atlas	$128 \times 128 \times 128$	$256 \times 256 \times 256$	$512 \times 512 \times 512$
1024×1024	33%	75%	91%
2048×2048	✓	50%	83%
4096×4096	✓	✓	65%
8192×8192	✓	✓	30%
16384×16384	✓	✓	✓

Table 3.1: Texture size and downscaling required for the single-channel *ImageTextureAtlas*.

3.2.6 Multi-channel large volume data atlas

With the technological advances of scanners, the resolution and accuracy are expected to grow continuously. Therefore, the visualization of large volume datasets should be considered.

There is a limitation with this approach in the amount of volume data that can be packed into an *ImageTextureAtlas* (see Section 3.2.5). The maximum number of slices that can be rearranged into a matrix composition is determined by the resolution of each slice and the maximum dimension supported by the 2D texture unit of the targeted GPU device. From an ubiquitous perspective, not all devices share the same memory capabilities.

The new approach uses multiple colour channels like the approach proposed by Noguera and Jiménez (2012) to store up to four times as much data in a single *ImageTextureAtlas*. However, the volume data slices are rearranged in a different order that makes better use of spatial coherence to optimize the usage of GPU memory. Slices are positioned across colour channels in depth order first, by stacking sequential slices into groups (eg. 4 for $RGBA$, 3 for RGB, etc.) and continue rearranging grouped slices in row-major order until all the texture is filled (see Figure 3.11).

Samples of each ray will fetch texture data from neighbour texels or even at the same position using the proposed ordering, specially when the camera is located to visualize the volume from the front and back view angles. This happens because sequential slices are often spatially closer in the composed atlas using a depth first order, i.e, in the next colour channel but at the same 2D texture coordinate. The proposed ordering will improve the use of GPU texture cache.

Two optimization strategies are proposed depending on the required colour channels. Figures 3.11 and 3.12 shows the difference orders employed for the two strategies: RGB and $RGB + A$ *ImageTextureAtlas*es.

i) **Storage capacity**. When the dataset is large, the storage capacity may be prioritized. In this case all the colour channels ($RGBA$) are used to store the whole volume data. Figure 3.11 shows a diagram of the order used to compose a 2×2 $RGBA$ *ImageTextureAtlas* with 16 slices of volume data. Please note that at the bottom right corner of each slice, the index in Z axis direction of the original data slice is given. For the $RGBA$ case a depth first sequence order is used.

ii) **Look-up optimization**. When all the colour channels are not needed to store the whole volume data (the targeted GPU has larger texture size capacity than required), a look-up optimization can be exploited. The free (not used) colour channel are used to store repeated data. The depth order in which sequential slices are stacked together is still maintained. However, the last slice of a stacked slice group will be the

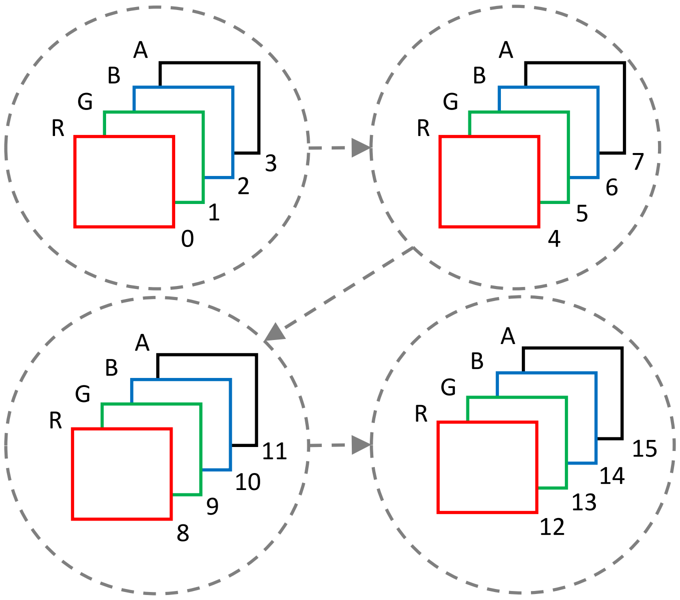


Figure 3.11: *RGBA* multi-channel order composition for a 2×2 *ImageTextureAtlas*.

same as the first slice of the next stacked group (following the row-major order matrix in groups). This composition is referred as *RGB* + *A ImageTextureAtlas*. Figure 3.12 shows the order sequence for the *RGB* + *A* case. Please note how the indices of the last slice in each multi-channel group *A* repeats with the first slice *R* in the next multi-channel group in row-major order. As data is repeated, in this diagram a 2×2 *RGB* + *A ImageTextureAtlas* is composed with 12 slices of volume data. The last slice at the last group is empty (contains no data). The motivation to use the redundant data is to avoid multiple 2D texture fetches to the *ImageTextureAtlas* per 3D volume sample when the trilinear interpolation is emulated in the fragment shader.

The sampling function that takes a 3D texture coordinate (x, y, z) and fetches the data from a *RGBA ImageTextureAtlas* is presented in Equation 3.5. The objective of this function is to return the voxel scalar value of a given 3D texture coordinate (r).

$$\vec{k}_1 = [1, 2, \ldots, n_c]$$
$$\vec{k}_2 = [0, 1, \ldots, n_c - 1]$$

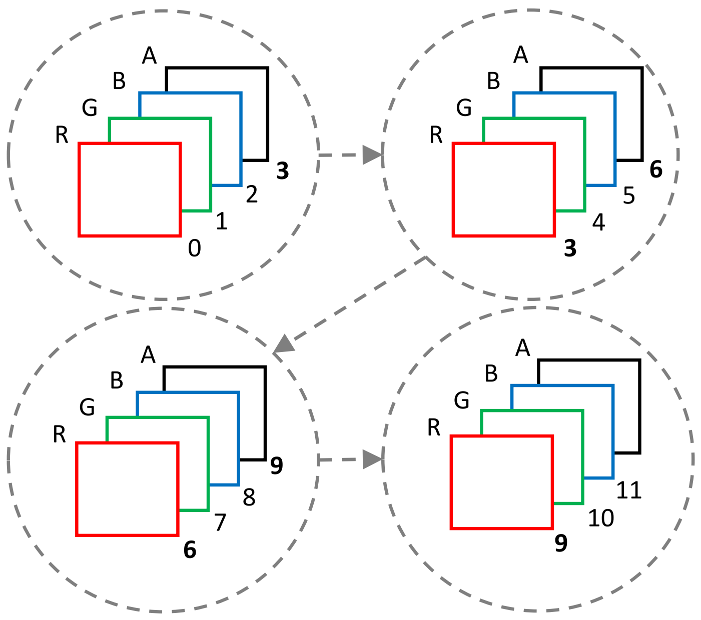


Figure 3.12: *RGB + A* multi-channel order composition for a 2 × 2 *ImageTextureAtlas*.

$$
\begin{aligned}
s_0 &= \lfloor z \cdot n_s \rfloor \\
s_1 &= s_0 + 1 \\
\vec{c_1} &= \mathbf{mod}(s_0, n_c) \\
\vec{c_2} &= \mathbf{mod}(s_1, n_c) \\
s_2 &= s_1 \cdot \frac{1}{n_c} \\
s_1 &= s_0 \cdot \frac{1}{n_c} \\
d_{x1} &= s_1 \cdot n_x - (\lfloor s_1 \cdot n_x \rfloor) \\
d_{y1} &= (\lfloor s_1 \cdot n_x \rfloor) \cdot n_y \\
d_{x2} &= s_2 \cdot n_x - (\lfloor s_2 \cdot n_x \rfloor) \\
d_{y2} &= (\lfloor s_2 \cdot n_x \rfloor) \cdot n_y \\
\vec{t_1} &= (d_{x1}, d_{y1}) + (x \cdot n_x, y \cdot n_y) \\
\vec{t_2} &= (d_{x2}, d_{y2}) + (x \cdot n_x, y \cdot n_y) \\
\vec{d_1} &= \mathbf{texture2D}(\vec{t_1}) \\
\vec{d_2} &= \mathbf{texture2D}(\vec{t_2})
\end{aligned}
\tag{3.5}
$$

$$
\begin{aligned}
\vec{a_1} &= \mathbf{step}(\vec{k}_1, \vec{c_1}) \\
\vec{a_2} &= \mathbf{step}(\vec{k}_1, \vec{c_2}) \\
\vec{b_1} &= \mathbf{step}(\vec{k}_2, \vec{c_1}) \\
\vec{b_2} &= \mathbf{step}(\vec{k}_2, \vec{c_2}) \\
\vec{f_1} &= \vec{b_1} - \vec{a_1} \\
\vec{f_2} &= \vec{b_2} - \vec{a_2} \\
r &= \mathbf{mix}(\vec{d_1} \cdot \vec{f_1}, \vec{d_2} \cdot \vec{f_2}, (z \times n_s) - s_0)
\end{aligned}
$$

Equation 3.5 uses GLSL 1.10 shader built-in math functions *mod, texture2D, step and mix. texture2D* fetches 2D texture data given a 2D vector. *step* method generates a step function by comparing two inputs and *mix* performs a linear interpolation between two values. $\vec{k_1}$ and $\vec{k_2}$ are pre-defined constant vectors which define a sequence delimited by the number of colour channels: *i)* when 4 colour channels are used, $RGBA \rightarrow \vec{k_1} = [1, 2, 3, 4]$. *ii)* When 3 colour channels are used $RGB \rightarrow \vec{k_1} = [1, 2, 3]$.

The *ImageTextureAtlas* configuration is defined by n_s (number of slices), n_x (1 / number of slices over X direction), n_y (1 / number of slices over Y direction) and n_c (number of colour channels).

The final scalar value r is returned as the data value for a given x, y, z 3D texture coordinate. All the other variables are temporal variables that are executed sequentially to compute the final value r.

Note that $\vec{c_1}$ is a 4-component vector, whose coordinates are initialized with the result of the scalar modulus operator. The same applies to $\vec{c_2}$.

The GLSL 1.10 version of WebGL does have some limitations such as dynamic indexing of a vector. This means that if a component of a vector is required to be accessed, the index of this component must be stated at compile-time. To obtain the final scalar value r with the proposed multi-channel $RGBA$ texture, the function in Equation 3.5 requires to fetch individually the colour channels (vector components) based in a computed index at run-time. To overcome this issue, math operations (dot product) and the GLSL *step* function are used to emulate this functionality and unpack the required vector component which contains the actual scalar value.

When the $RGB + A$ *ImageTextureAtlas* is used, the function to extract the final scalar value r given a 3D texture coordinate (x, y, z) is similar to Equation 3.5. However it has a major difference: only one *texture2D* instruction is used instead of two.

Equation 3.6 shows the mapping function required for the $RGB + A$ *ImageTextureAtlas* case. It uses the same notation as Equation 3.5. For this configuration $(RGB + A)$ n_c equals 3.

$$\begin{aligned}
\vec{k}_1 &= [1, 2, \ldots, n_c] \\
\vec{k}_2 &= [0, 1, \ldots, n_c - 1] \\
s_0 &= \lfloor z \cdot n_s \rfloor \\
s_1 &= \lfloor s_0 \cdot \frac{1}{n_c} \rfloor \\
\vec{c} &= \mathbf{mod}(s_0, n_c) \\
d_x &= s_1 \cdot n_x - (\lfloor s_1 \cdot n_x \rfloor) \\
d_y &= (\lfloor s_1 \cdot n_x \rfloor) \cdot n_y \\
\vec{t} &= (d_x, d_y) + (x \cdot n_x, y \cdot n_y) \\
\vec{d}_{rgba} &= \mathbf{texture2D}(\vec{t}) \\
\vec{f} &= \mathbf{step}(\vec{k}_1, \vec{c}) - \mathbf{step}(\vec{k}_2, \vec{c}) \\
\vec{v} &= \mathbf{mix}(\vec{d}_{rgb}, \vec{d}_{gba}, (z \times n_s) - s_0) \\
r &= \vec{v} \cdot \vec{f}
\end{aligned} \tag{3.6}$$

Since it is ensured that the next colour channel always contains the next data slice (due to data repetition) with one 2D sample (*texture2D*) we fetch the data from both required slices at the same time and after interpolation, using a dot product operation the final data value r can be extracted from the correct colour channel (vector component). This has a beneficial impact in devices like tables and mobile devices where the memory latencies of their GPUs are not as capable as their desktop counterparts.

The same strategies can be used with small volumetric datasets to compose RGB, $RG + B$, RG or $R + G$ multi-channel *ImageTextureAtlas*es.

Using this approach more data can be packed in a single graphics texture unit, making better use of the memory resources of the GPU. Table 3.2 shows the texture size required to store volumetric datasets up to $512 \times 512 \times 512$ using the presented multi-channel $RGBA$ approach.

	Dataset		
Atlas	$128 \times 128 \times 128$	$256 \times 256 \times 256$	$512 \times 512 \times 512$
1024×1024	✓	50%	83%
2048×2048	✓	✓	67%
4096×4096	✓	✓	33%
8192×8192	✓	✓	✓
16384×16384	✓	✓	✓

Table 3.2: Texture size and downscaling required for the multi-channel *RGBA ImageTextureAtlas*.

Chapter 4

Automated shader composition and generation

In several fields, volumetric data and multi-variate data needs to be analysed, evaluated and interpreted. To achieve these goals, the capability to visualise volumetric data is mandatory as visualisation is a natural way to interpret and understand data.

The scientific domain is where volumetric visualisation is more frequently used. In fields like biology, geology, medicine and physics volumetric data that needs to be visualised, processed and inspected can be found. Nonetheless, in other domains such as engineering, volumetric rendering can be used in fluid and heat transfer simulations.

As the specialization field changes, the needs and requirements for each visualisation also change. From a practical point of view, a direct volume visualisation of the raw data is not enough to interpret the data. Additional enhancements to the visualization algorithms are required in order to target the use cases and necessities of each field. For instance, the ability to discern segmented regions in the volume by the use of colour mapping is frequently used in the medical field.

Unfortunately, the use and modification of the rendering algorithms require either great knowledge of computer graphics or the use of specialized software designed for each specific use case. Volumetric visualisation software is designed to fulfil requirements or goals specific to a certain field. Essentially, each application modifies the rendering process to achieve a certain visualisation style. The non-standardization of this rendering algorithms has negative consequences in the visualisation of the content: few devices have the software to render this information. Developing tailored software does not impose a problem to specialized software products, but it is a real problem for general purpose visualisation applications.

In this chapter an architecture for the automated generation of GPU volume rendering shader code generation is presented. The proposed architecture implements the X3D ISO volume rendering set of nodes and composable styles. These contributions overcome three main challenges: *i*) facilitate the creation of graphics shader code to non-expert users, *ii*) allow the interoperability and visualisation of volume rendering content over the web, and *iii*) provide a cross-domain ubiquitous volume rendering solution within the web platform.

This chapter is organized as follows. Section 4.1 explains the shader generation process. Using this approach, Section 4.2 presents an implementation of the ISO/IEC X3D volume rendering nodeset for the web.

4.1 Automatic shader generation

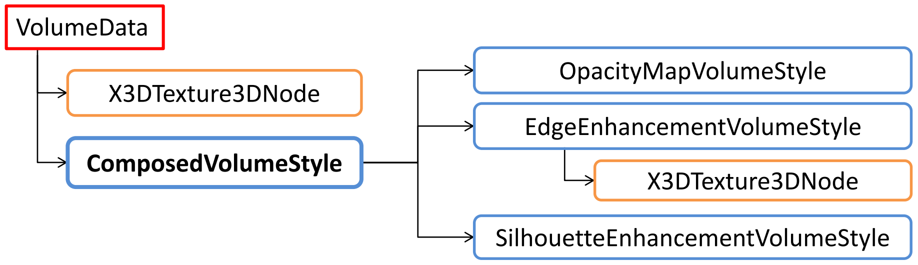
A novel architecture is proposed and implemented for the X3D scene declaration schema and node sets. Based on the nodes of this scene, the necessary code is created on run-time during the scene tree traversal. The shader generation and composition uses a template based approach. Section 4.1.1 presents the X3D scene declaration and Section 4.1.2 presents the composition architecture for the web.

4.1.1 Declarative approach for the scene definition

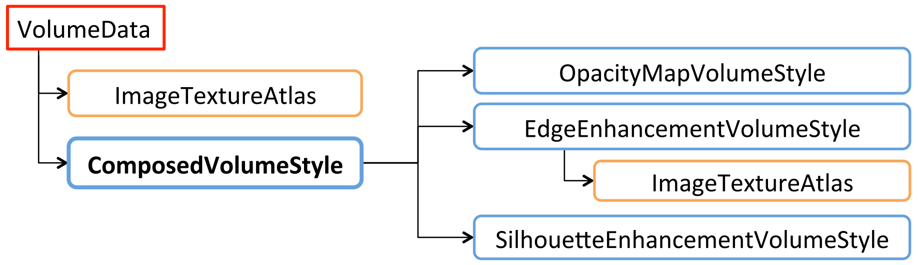
X3D is a royalty-free and matured ISO standard (Web3DConsortium, 2017b). Conceived for interchangeable 3D content on the Web, it aims to represent a 3D real-time scene with a standard XML-based file format. X3D defines several profiles, each of them is composed of a set of components. Some of these components are extensions added by collaborating committees. There are desktop implementations of X3D (non-web). This chapter addresses the web implementation of the volume rendering nodes of X3D.

The scene graph is the basic entity of the X3D run-time environment. It contains the objects and relations that define the scene. The X3D standard defines a set of nodes for volume rendering, along with the definition of its fields and expected output behaviour (rendering output). Following this convention different applications can use the same volume rendering scene definitions.

Figure 4.1 shows an example of a volumetric scene tree for the `backpack` (University of Tübingen WSI/GRIS, 2014) dataset, where each node type is shown with a distinctive colour.



(a) Non-web

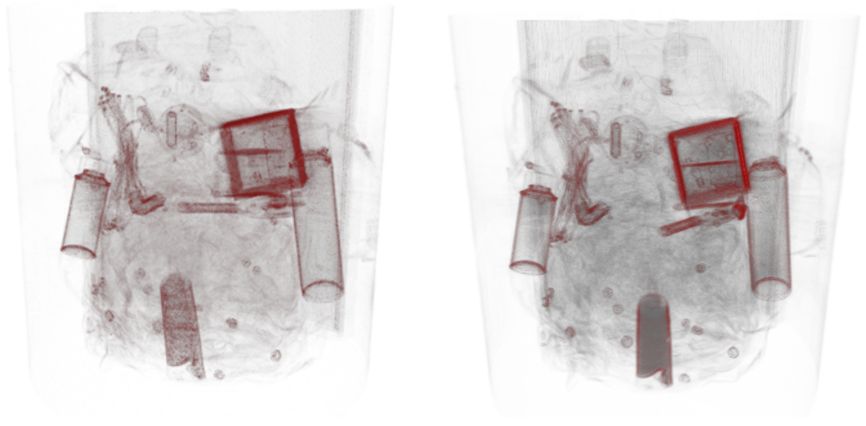


(b) Web

Figure 4.1: Partial tree of a composed volume scene. a) scene tree for standard desktop X3D and b) scene tree for X3DOM in the web.

In this example, a scene is defined with a *ComposedVolumeStyle* which includes two rendering styles. Firstly, edges are enhanced with red colour using the *EdgeEnhancementVolumeStyle* and then, the *SilhouetteEnhancementVolumeStyle* highlights the areas where the surface normals are perpendicular to the view direction. Figure 4.2 shows the rendering output of the X3DOM scene tree and Listing 4.1 shows the user declaration of the web scene. The novel automatic web implementation generates the GPU shader code that renders the nodes as they are specified by a web content developer (see Section 4.1.2).

The *ImageTextureAtlas* is an additional node, currently not defined by the X3D standard. This node is used by the component to provide the volume data or the gradient data and it has been defined in Chapter 3. Depending on the scene, the gradient data can be provided as an additional *ImageTextureAtlas* texture and declared as a child node of the *X3DVolumeDataNode*; or as a child node of the *X3DComposableVolumeRenderStyleNode*.



(a) Non-web (b) Web

Figure 4.2: Two rendering outputs of the `backpack` dataset using the *Composed-VolumeStyle* with the *SilhouetteEnhancementVolumeStyle* and the *EdgeEnhancementVolumeStyle*. a) Non-web rendering output taken from the X3D standard. b) Rendering output from the novel web implementation.

In a volume rendering scene, multiple volume data nodes can be declared. A custom shader will be generated for each declared volume data type derived node (*X3DVolumeDataNode*). The volume data node contains the initial shader template code. This template is completed by the rendering styles declared as child nodes in the scene. In the case of Listing 4.1 the *EdgeEnhancementStyle* and *SilhouetteEnhancementVolumeStyle* nodes contains functions that are inlined in the ray traversal to modify the rendering output according the declared parameters.

4.1.2 Web composition novel architecture

The volume rendering component generates on-the-fly the necessary shaders to be used by the programmable graphics pipeline available through WebGL. Therefore, the workload of the volume rendering ray-casting method is done by shader programs running in the GPU. Shader programs are a set of text strings that are passed to the graphics hardware driver for compilation and execution. This approach is based on the previous works by Congote et al. (2011) and Mobeen and Feng (2012b). A single shader program (vertex and fragment shader) is generated for each volume data declared on the scene.

The declarative nature of X3D allows to nest multiple rendering styles in a hierarchically constructed node scene graph. Using a given volume data, Listing 4.1: Composed scene declaration.

```
<X3D width="500px" height="500px">
<Scene>
 <Background skycolour="1 1 1" transparency="0"></Background>
 <Viewpoint zNear="0.0001" description="Default" zFar="100"></
     Viewpoint>
 <VolumeData id="vol" dimensions="4 4 4">
  <ImageTextureAtlas containerField="voxels" url="backpack.png"
      numberOfSlices="373" slicesOverX="20" slicesOverY="20">
  </ImageTextureAtlas>
  <ComposedVolumeStyle>
   <OpacityMapVolumeStyle lightFactor="0.7" opacityFactor="20">
   </OpacityMapVolumeStyle>
   <EdgeEnhancementVolumeStyle gradientThreshold="0.6" edgecolour="1
       0 0">
    <ImageTextureAtlas containerField="surfaceNormals" url="backpack
       -g.png" numberOfSlices="373" slicesOverX="20" slicesOverY="20
       ">
    </ImageTextureAtlas>
   </EdgeEnhancementVolumeStyle>
   <SilhouetteEnhancementVolumeStyle silhouetteBoundaryOpacity="1.0"
       silhouetteRetainedOpacity="0.1" silhouetteSharpness="1.2">
   </SilhouetteEnhancementVolumeStyle>
  </ComposedVolumeStyle>
 </VolumeData>
</Scene>
</X3D>
```

content developers can define a X3D scene with the desired rendering styles. As a result, they will get the desired visualisation of such dataset. For instance, they could use different illustrative styles in two segments within a volume or they could compose a selection of styles to enhance the contours of the volume.

Thus, the number of possible scene declarations is unbounded. Each scene requires its specific shader to implement the volume rendering. In this regard, to fulfil the dynamic requirements of X3D, the component avoids storing pre-defined shaders. Instead, shaders are created on-the-fly by composing a set of strings which are collected during the traversal of the X3D volume rendering nodes. Each node defines its own shader strings, which are added to a common template defined at the root level (see Figure 4.3).

This process starts when a new web page with X3D content is loaded.

The ray-casting loop is implemented in a fragment shader. The component generates automatically the shader code required for each X3D scene. It uses a fixed step size and a fixed maximum number of steps in the ray-casting loop, because the GLSL shading language requires the number of instructions sent to the GPU to be known at compiling time.

Unlike the fragment shader, the vertex shader is common to all scenes. When the HTML document is loaded on the browser, a scene traversal is triggered to load the X3D scene. Once the traversal has parsed the child nodes of the root volume data node and they are attached to the DOM, the shader generation begins. This shader generation is performed in two steps: an *initialization phase* and a *shader code generation phase*.

During the initialization phase, shader uniforms and texture variables are collected from the child nodes, initializing their values so that they can be handled by X3DOM. The initialization of these variables is needed as they must be declared on both the JavaScript (CPU) and shader (GPU) sides. There are several factors that are taken into account for this phase:

i) The uniforms data types must be specified before compilation.

ii) The name of the uniforms and texture variables must not be the same to avoid name collision problems when the same style is applied more than once.

iii) The assignment of a free texture unit to each texture sampler must be managed.

iv) Each variable must be correctly linked to its node parameter at the DOM tree to generate dynamic changes on the shader variables.

In the shader code generation phase, the complete shader code is composed, compiled and linked in the GPU. The volume data type node (*X3D-VolumeDataNode*) defines the base template of the fragment shader (see Figure 4.3, on the right). The missing parts of this template are filled with the strings collected from its child nodes in several traversals. In general terms, a render style node (*X3DVolumeRenderStyleNode, X3DComposable-VolumeRenderStyleNode*) defines a set of strings where the *uniforms* and *textures*, the *lighting equation*, the *style functions* and the *inline code* are stored.

The *uniforms* and *textures*, marked as red on Figure 4.3, declare the input parameters that are used by the style. Therefore, they have to be located at the top of the template. The code generated by the template to

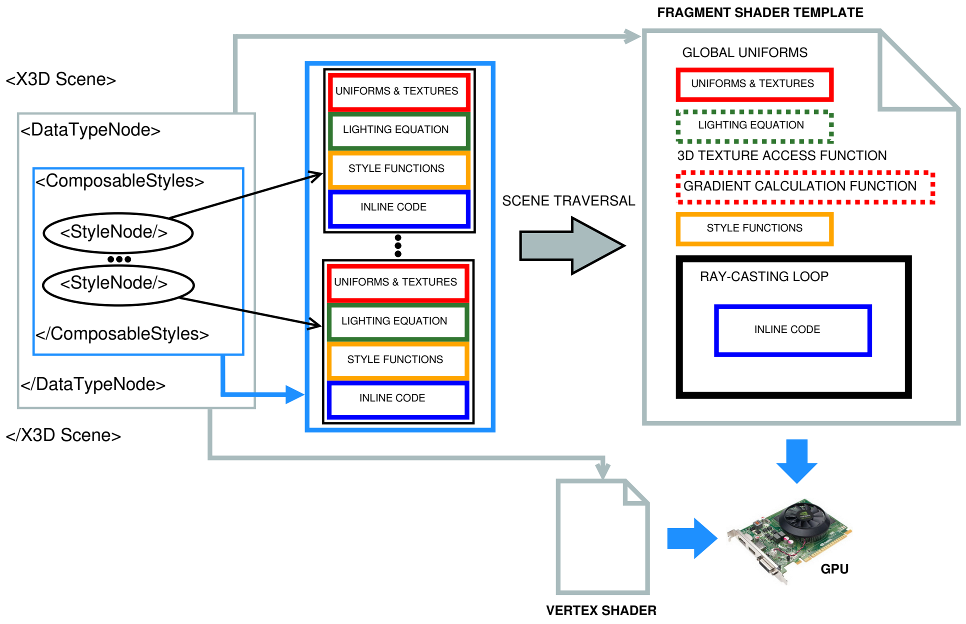


Figure 4.3: Template-based shader code generation.

calculate or access the gradient data is conditioned by whether the gradient data is provided by a render style node through a texture or not. When no texture gradient is provided, a function to calculate the gradient is generated on the template. In the opposite case, a function to access the gradient atlas is generated.

The *lighting equation* marked as green is an optional function which if the user declares a light on the scene id added to the template or if it is mandatory to the style, e.g. *ShadedVolumeStyle* (Section 4.2.3) it will also be added.

The *style functions* marked as yellow are strings composed of functions that modify the ray accumulation according to the style logic.

The *inline code* marked as blue is code to be located within the ray-casting loop. It consists mainly of function calls to the *style functions*, but it can also contain code to serialize or blend results of several styles and code that can not be separated on *style functions*, such as temporal variables.

Some render styles are composable, so there can be several rendering styles applied to one dataset. Each of the styles defines their own strings following the same described structure. They are collected and appended one after the other on their corresponding part of the template. An exception is the *lighting equation* string, which is not appended. As defined in the X3D standard, the lighting equation is only collected from the first style node. By filling each part of the template with the collected strings, the shader is completed and ready to be compiled.

4.2 X3DOM volume rendering component

X3DOM is a DOM based model that allows the integration of the X3D nodes into the HTML5 DOM content (Behr et al., 2009). It adds the capability of declaring 3D scenes under the X3D format and directly modify the X3D tree through DOM events. This chapter, proposes the first implementation for the Web of the volume rendering component defined by the X3D. The set of volume rendering styles which composes the component, in combination with X3DOM, provides a suitable framework for real-time volumetric visualisation under any WebGL compatible device. This approach benefits from the ubiquity of the Web, as it is deployable even on mobile devices.

The camera defined in a volume rendering scene can be interactively manipulated by the web page viewer: rotation, zoom, pan are the basic camera manipulation methods. X3DOM connects the X3D scene with the DOM. Changes on the scene can be done with the addition of JavaScript event handlers and listeners that change the attributes of a volume rendering node tag at the DOM tree.

The `<x3d>` tag element is the initial statement to embed a 3D canvas. Each X3D node matches with a corresponding tag under the `<x3d>` namespace. Input parameters of the rendering style nodes are usually shader uniform variables (see Figure 4.3). Once they have been compiled at run-time, an update in a style parameter will dynamically modify the uniform value. As a result, the output rendering will be updated in real-time without the need of regenerating and compiling the shader again. Textures are also linked as uniforms on the shaders. Thus, an update on the input textures (such as the volume data, gradient data or any transfer function) will be directly reflected on the rendering output.

This approach creates custom shaders when the scene is loaded. When needed, shader code is generated based on the provided parameters, possibly affecting the *style function*, *inline code* or the base *template* (see Figure 4.3). The modification of such parameters will require the regeneration of the shaders again. For these cases, the scene must be reloaded (complete scene traversal) to compile and link the new updated shader program.

The scene graph is the basic entity of the X3D run-time environment. It contains the objects and relations that define the scene. The X3D standard

Abstract nodes	**Implementation nodes**
X3DVolumeDataNode	VolumeData SegmentedVolumeData IsoSurfaceVolumeData
X3DVolumeRenderStyleNode	ProjectionVolumeStyle
X3DComposableVolume- RenderStyleNode	ComposedVolumeStyle OpacityMapVolumeStyle EdgeEnhancementVolumeStyle BoundaryEnhancementVolumeStyle SilhouetteEnhancementVolumeStyle ToneMappedVolumeStyle BlendedVolumeStyle CartoonVolumeStyle ShadedVolumeStyle

Table 4.1: X3D volume rendering component nodes for v3.3.

defines a set of nodes for volume rendering, along with the definition of its fields and expected output behaviour. The Medical Working Group of X3D is the responsible for the definition of the volume rendering component nodes.

The node hierarchy defined for the volume rendering nodes is composed of three abstract node types. The root node describes the volume data to be rendered and it is defined as *X3DVolumeDataNode*. A volume rendering style node defines how the volume data is rendered, producing illustrative and non-photorealistic renderings to enhance the visual output. The style nodes derive from a *X3DVolumeRenderStyleNode* or a *X3D-ComposableVolumeRenderStyleNode*, and they are declared as children of the *X3DVolumeDataNode*. Style nodes that inherit from the *X3DComposable-VolumeRenderStyleNode* can be composed: the output of a style can be the input of the next applied style.

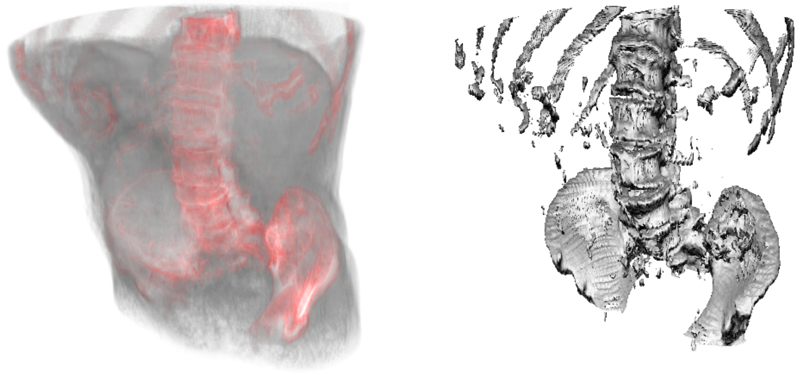
Table 4.1, lists all the nodes defined in the X3D volume rendering component (version 3.3).

There was no prior implementation of these nodes for the Web platform in the literature. Following subsections describe the implementation of each of these nodes.

4.2.1 X3DVolumeDataNode

The *X3DVolumeDataNode* has three derived nodes: *VolumeData*, *Segmented-VolumeData* and *IsoSurfaceVolumeData*.

The *VolumeData* specifies a non-segmented volume. It is the most basic node. The styles attached to this node will be applied to the whole volume data. By default an *OpacityMapVolumeStyle* is used.



(a) *SegmentedVolumeData* (b) *IsoSurfaceVolumeData*

Figure 4.4: Two rendering outputs of the `aorta` dataset using the *X3DVolume-DataNode*. a) Two segments tissue and bones, the first rendered using a *BoundaryEnhancementVolumeStyle* and the second with an *EdgeEnhancement-VolumeStyle*. b) An isosurface (scalar value 0.92) rendered with the *Cartoon-VolumeStyle*.

The *SegmentedVolumeData* takes a segmented volume data as input. The segment identifier assigned to each voxel is not stored in the volume data. Therefore, when required in the rendering process, a segment identifier is assigned using Equation 4.1.

$$id = \lfloor f(x) \times maxSegment - 0.5 \rfloor \tag{4.1}$$

In Equation 4.1, $f(x)$ is the voxel value and $maxSegment$ is the number of segments considered (by default, 10). Each segment is mapped to a render style in strict order of declaration (see Figure 4.4a). In addition to standard attributes, a $maxSegment$ parameter has been added. This attribute allows to adjust the way the segment identifiers are computed from the input segmented volume data.

The *IsoSurfaceVolumeData* allows the visualisation of one or more surfaces extracted from the volume data (see Figure 4.4b): *"An isosurface is* *defined as the boundary between regions in the volume where the voxel values are larger than a given value (the isovalue) and smaller on the other side and the gradient magnitude is larger than a given surface tolerance"* (Web3DConsortium, 2014).

$$\begin{cases} C_g = styleNode(C_v, O_v) \wedge O_g = 1, \\ \qquad \text{if } (f(x) \geq iso_v \vee f(x-1) < iso_v) \wedge \left\| \bigtriangledown f(x) \right\| \geq s_t \\ C_g = styleNode(C_v, O_v) \wedge O_g = 1, \\ \qquad \text{if } (f(x) \leq iso_v \vee f(x-1) > iso_v) \wedge \left\| \bigtriangledown f(x) \right\| \geq s_t \\ C_g = C_v \wedge O_g = 0, \qquad \text{otherwise} \end{cases} \tag{4.2}$$

Multiple isovalues can be given as parameters to the style. Equation 4.2 shows the conditional statement used to check if a voxel belongs to a given isosurface. C_v and O_v are the original voxel colour and opacity. C_g and O_g are the generated output colour and opacity. When multiple isovalues are given, a rendering style is associated to each isovalue, following the rules of the X3D specification.

4.2.2 X3DVolumeRenderStyleNode

The *X3DVolumeRenderStyleNode* has only one derived node: the *ProjectionVolumeStyle*.

The *ProjectionVolumeStyle* allows three types of rendering methods: *max*, *min* and *average*. Each method outputs a colour based on the voxels values traversed by a ray.

Maximum Intensity Projection (MIP) stores the greatest value along the ray (see Figure 4.5) and it is widely used in the medical field. It was originally proposed by Wallis et al. (1989) for its use in Nuclear Medicine. It can be used for lung nodules detection in lung cancer for computed tomography data and for magnetic resonance angiography studies (Perandini et al., 2010).

Minimum Intensity Projection outputs the minimum value along the ray. *Average Intensity Projection* outputs the average value along the ray traversal and the resultant rendering is an approximation of an X-Ray.

4.2.3 X3DComposableVolumeRenderStyleNode

Nodes derived from the *X3DComposableVolumeRenderStyleNode* can be composed resulting in richer renderings. The *X3DComposableVolumeRenderSty-*

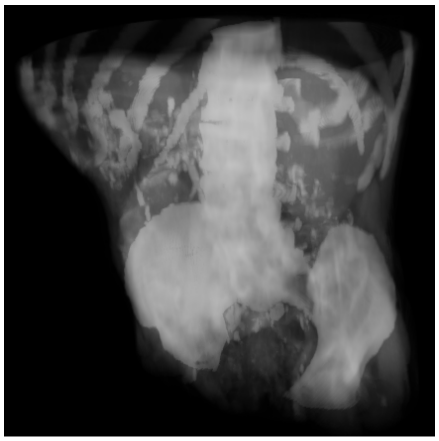


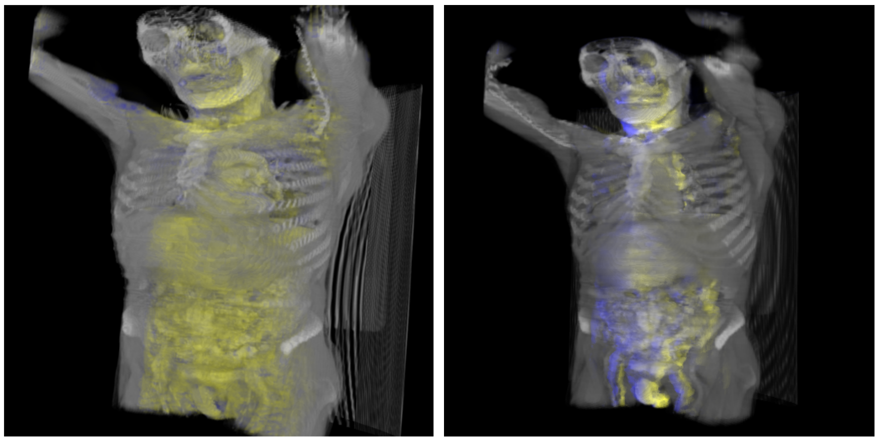
Figure 4.5: The rendering output of the `aorta` dataset using the *Projection-VolumeStyle* with the *max* method (MIP).

leNode has the following derived nodes: *ComposedVolumeStyle*, *Blended-VolumeStyle*, *CartoonVolumeStyle*, *OpacityMapVolumeStyle*, *BoundaryEn-hancementVolumeStyle*, *EdgeEnhancementVolumeStyle*, *SilhouetteEnhance-mentVolumeStyle*, *ToneMappedVolumeStyle* and *ShadedVolumeStyle*.

The *ComposedVolumeStyle* allows compositing multiple *X3DComposa-bleVolumeRenderStyleNode* rendering styles under a single render pass. This is done by serializing the styles; the output of a style is the input of the next style. The component applies the styles in the same order as declared. There is no order restriction for the styles, i.e. the order in which the styles are declared is decided by the X3D designer. But the order is important, as the X3D standard defines, the equation for the lighting is always taken from the first rendering style node.

The *BlendedVolumeStyle* allows blending two volume datasets with a weight function (see Figure 4.6). The main dataset is the parent *X3D-VolumeDataNode* and the second dataset is passed as a parameter to the *BlendedVolumeStyle* using an *ImageTextureAtlas* node. The X3D standard defines several options for the weight function: it can be a constant value, a value dependent on the opacity of one of the datasets or it can be a texture. When a texture is provided as a weight function, each opacity value from the dataset is mapped to a weight value from the texture. The use of a *ComposedVolumeStyle* is mandatory when the X3D designer wants to apply a rendering style to each of the datasets.

The *CartoonVolumeStyle* takes two colours as input parameters. The final rendering will depend on the local surface normals and the view direc-



(a) X3D example (b) Our implementation

Figure 4.6: Two rendering outputs using the *BlendedVolumeStyle* with the `body` (Web3DConsortium, 2014) and `internals` (Web3DConsortium, 2014) datasets. (a) Rendering output taken from the X3D standard. It uses the *OpacityMapVolumeStyle* on the `body` and the *ToneMappedVolumeStyle* on the `internals`. (b) Rendering output from our implementation. Also, it uses the *OpacityMapVolumeStyle* on the `body` and the *ToneMappedVolumeStyle* on the `internals`.

tion. The result is a cartoon-style non-photorealistic rendering (Decaudin, 1996). The alpha channel of the input colours is not taken into accout in the component. Instead, the opacity values are obtained from the volume data.

The *OpacityMapVolumeStyle* maps the opacity and colour values for each voxel from a function stored as a texture. This texture is called transfer function (see Section 7.4). The creation of this transfer function is delegated in the designer and is created in an offline preliminary step. Extensive work has been done regarding this topic. Kniss et al. (2002) denoted the use of multi-dimensional transfer functions. Bruckner and Gröller (2007) implemented illustrative styles through transfer functions. Only the support of a 1D transfer function texture is mandatory to be compliant with the X3D standard.

The *BoundaryEnhancementVolumeStyle* modifies the opacity of the volume. This approach, based on the gradient magnitude, enhances boundaries. A volume is usually composed of several densities. The gradient magnitude is low in areas of constant density, and it is large when density varies.

$$O_g = O_v \times (K_{gc} + K_{gs} \times (\left\| \bigtriangledown f(x) \right\|^{K_{ge}})) \tag{4.3}$$

The Equation 4.3 is used to enhance the opacity of boundaries. K_{gc} is the amount of initial opacity to retain, while K_{gs} and K_{ge} adjust the darkness of the boundary.

The *EdgeEnhancementVolumeStyle* highlights the edges of the volume with an input colour parameter. Edges are volume data values where the gradient is perpendicular to the view direction. The input colour is blended with the volume data colour in function of the angle between both vectors (see Equation 4.4).

$$C_g = \begin{cases} C_v \times \left|\bigtriangledown f(x) \cdot V\right| + edgeColour \times (1 - \left|\bigtriangledown f(x) \cdot V\right|), \\ \quad \text{if } \left|\bigtriangledown f(x) \cdot V\right| > \cos(gradThreshold) \\ C_v, \quad \text{otherwise.} \end{cases} \tag{4.4}$$

$$O_g = O_v \tag{4.5}$$

The edge enhancement can be more or less noticeable with the threshold parameter *gradThreshold*. It is used to adjust the edge detection. The *edgeColour* is the input colour and the normalized view direction is denoted by V.

The *SilhouetteEnhancementVolumeStyle* is similar to the *EdgeEnhancementVolumeStyle*: both enhance the voxels where the gradient is perpendicular to the view direction. In this case, only the opacity is enhanced, but not the colour.

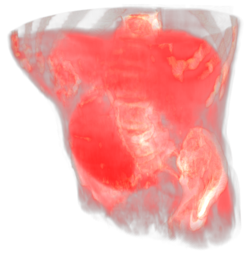
$$O_s = O_v \times (K_{sc} + K_{ss} \times (1 - \left|\bigtriangledown f(x) \cdot V\right|^{K_{se}})) \tag{4.6}$$

The Equation 4.6 is used to enhance the opacity of the volume. K_{sc} is the base opacity factor to retain. It regulates the non-silhouette areas. K_{ss} represents the silhouette enhancement factor and K_{se} is an exponent to control the sharpness of the silhouette. The three factors are the input parameters of this style.

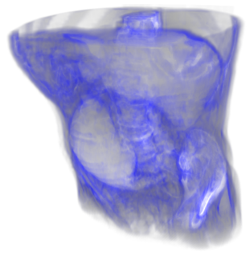
The *ToneMappedVolumeStyle* illustrates the volume based on the orientation towards the light. Gooch et al. (1998) were the first to propose an illumination model following this approach. This tone shading technique defines two colours: warm and cool. The warm colour represents surfaces facing towards the light direction, and the cool colour is used for surfaces facing away the light. The interpolation between these colours is assigned using the dot product between the angles of the gradient and the light direction to each sampled voxel.

Currently, the volume rendering component supports local illumination following the Blinn-Phong illumination model (Blinn, 1977). Additionally, the *ShadedVolumeStyle* node allows to specify the fog and material properties.

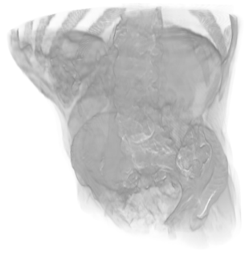
When gradient data is passed as parameter to one of the child nodes of the *ComposedVolumeStyle*, it is loaded just once, being available for the rest of the style nodes. The memory consumption is reduced by avoiding multiple instantiation of the texture. Any other gradient data defined on the styles will be ignored, except if it is defined with the *BlendedVolumeStyle*, where a second gradient data texture can be provided. Figure 4.7 shows the rendering output of our implementation for each described *X3DComposableVolumeRenderStyleNode*.



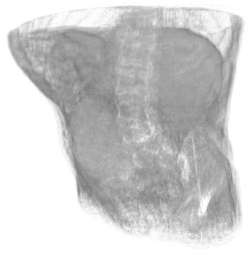
(a) *OpacityMapVolumeStyle*



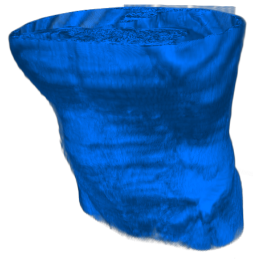
(b) *EdgeEnhancementStyle*



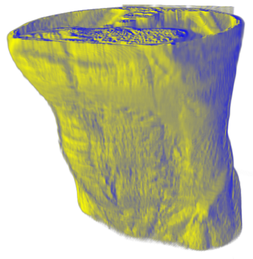
(c) *BoundaryEnhancementVolumeStyle*



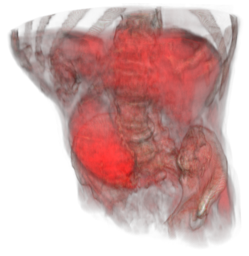
(d) *SilhouetteEnhancementVolumeStyle*



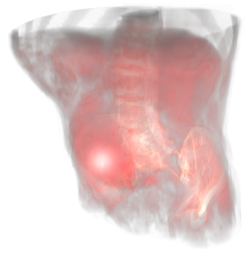
(e) *CartoonVolumeStyle*



(f) *ToneMappedVolumeStyle*



(g) *ComposedVolumeStyle*



(h) *ShadedVolumeStyle*

Figure 4.7: The rendering output of each *X3DComposableRenderStyleNode* using the `aorta` dataset

Chapter 5

Hybrid volume rendering

Volume rendering deals with the visualisation of 3D scalar data. Thus, rendering volumetric datasets requires a different approach than rendering surface data. Traditional rendering techniques focus on rendering of surface data. Polygonal mesh models have many advantages over volumetric models which make them more appropriate for 3D real-time applications: they are faster to render with common hardware. Current rendering pipelines optimize triangle-based rendering which target only surface data representation.

In contrast to surface data, volumetric content has relevant information in the inside of the enclosed volume. However, sometimes the combination of both volumetric and polygonal meshes is required to achieve the desired visualization. This is called hybrid volume rendering.

This chapter presents a hybrid volume rendering algorithm that targets all WebGL compatible devices. Additional WebGL extensions that could also be used to achieve better performance are also briefly discussed. Web 3D rendering is ubiquitous, which is a great advantage for a great variety of domains. However, web platforms have also some shortcomings: quite different rendering power of different devices, not every hardware feature available, etc. These drawbacks are needed to be considered in order to create application with hybrid rendering, i.e, with volumetric datasets and polygonal meshes in the same virtual scene.

The integration of 3D polygonal meshes in volume rendering is a key function in medical simulations and in other fields like engineering process simulation. In this chapter the `aorta` dataset is used to visually validate the presented hybrid volume rendering algorithm. The chapter is organized as follows. Section 5.2 presents the two-pass basic volume rendering algorithm. Section 5.3 describes the multi-pass hybrid volume rendering approach and

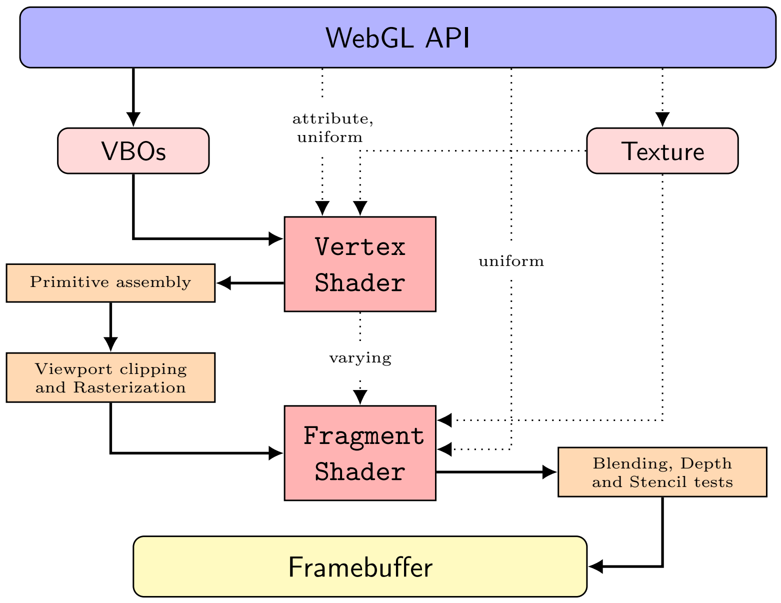


Figure 5.1: Simplified overview of the WebGL programable graphics pipeline.

Section 5.4 discusses available WebGL extensions that could be used to improve the performance.

5.1 WebGL graphics pipeline

A programmable graphics pipeline allows to apply different type of rendering effects because some steps of the graphics pipeline can be customized or programmed. It allows the definition of the calculations for computing colour, position, texture coordinates and the lightning model to be applied in a geometric model. The pipeline is composed of several steps: some are fixed and some are programmable. For the programmable steps, two main programs are defined by developers, these are known as shaders. These *vertex and fragment shaders* run on the GPU.

Figure 5.1 shows a simplified diagram of the WebGL 1.0 graphics pipeline. A summary of the steps to perform in the WebGL 1.0 pipeline is as follows:

1. Set-up of the geometric data. Vertex data is uploaded as an array that is placed into a Vertex Buffer Object (VBO). Additional data, such as normals, texture coordinates and indexing of vertices are provided with calls to WebGL API methods.

2. The *vertex shader* computes the screen position of each vertex and optionally performs any additional calculation at a per-vertex basis.

3. The output of the *vertex shader* continues into the primitive assembly step (geometry assembly) so that the GPU computes geometric primitives (triangles).

4. The rasterization discards primitive parts outside the viewport. Parts within the viewport are divided into pixels and grouped into fragments.

5. Per-vertex associated values (colour, coordinates, etc.) are interpolated.

6. Fragments with the interpolated values are passed into the *fragment shader* program.

7. The *fragment shader* program is executed to set the colour value for each pixel and any additional computation, such as lightning effects.

8. Fragments are either discarded or passed as colour values into a framebuffer object (FBO).

Any WebGL application requires a pair of *vertex shader* and *fragment shaders* and the WebGL API gives the data required for their computation to these shaders: vertex data (position, colour, etc.), shader input values (attribute, uniform) and bitmap data (textures). The *vertex shader* program is executed for every vertex to determine the coordinates of triangles in the rendering canvas. Then, the computed triangles are rasterized by the GPU. With the rasterization, the pixels to be drawn in the canvas are determined. The *fragmet shader* is run for every pixel to compute the colour of the pixel that will be written to the final framebuffer. The main framebuffer is the screen buffer. Additional framebuffers can be defined and linked between shaders as texture inputs to concatenate rendering passes (full cycle of the graphics pipeline).

5.2 Volume ray-casting with the programable graphics pipeline

Ray-casting is a direct volume rendering technique that generates rays from the camera position which traverse the volumetric data, mimicking the physical model of light (see Section 2.1 in Chapter 2). The technique was originally presented by Kajiya and Von Herzen (1984b). Then, Kruger and Westermann (2003b) improved the volume ray-casting performance by adapting the technique to be run in GPUs. Their algorithm uses the GPU programable graphics pipeline. Later, Congote et al. (2011) extended the algorithm to be used in WebGL. In this method, the ray-casting is performed in two rendering passes.

The graphics pipeline is intended to be used with surface data and not volumetric data: the first input of the pipeline is required to be composed triangle-based geometry. For ray-casting, in order to position the volume data in the scene a proxy geometry is defined: a cube. This cube is the Volume Bounding Box (VBB). Figure 5.2a shows the vertex positions of the VBB in local space coordinates.

To cast rays from the camera position into the VBB, the ray-direction on each pixel in the screen must be computed. The method proposed by Kruger and Westermann (2003b) uses the GPU rasterization step to compute the ray entry and exit points in the VBB. A colour is assigned to each vertex of the cube matching the bounds of the 3D texture UV coordinates. Figure 5.2b shows how these colours are assigned to each vertex. Both the position and colour of the vertices are uploaded to the *vertex shader* and then, the colour of each vertex are interpolated for each pixel in the faces of the cube in the GPU rasterization step (see Section 5.1). The purpose of this process is to compute the ray entry and exit points (spatial position) and pass them to the *fragment shader* in a valid format (encoded as *RGB* values in a texture). In this way, with the casted ray data (3D data), the volume rendering can be computed at the pixel level in the projected 2D image.

The VBB is projected into the 2D canvas. However, from the view position the back side of the cube will be hidden: the front side will overlap the back side. For this reason, they performed this method in two rendering passes:

First pass (Ray exit point determination): The three back-faces of the VBB (volume bounding box) are rendered into a Frame Buffer Object (FBO). The FBO output result is a coloured 2D projection as a 2D *RGBA* texture. As a result of this rendering pass, the 3D texture coordinates from the back side of the cube are projected into this 2D texture. The projected coordinates (FBO output) match the exit location of the rays in the volume data texture coordinate space (see Figure 5.3).

Second pass (Ray direction determination): The three front-faces of the volume bounding box are rendered in the screen buffer with the same *vertex shader* as in the first render pass. In the *fragment shader*, using the interpolated per-vertex colour of the front faces and the back-faces coordi-

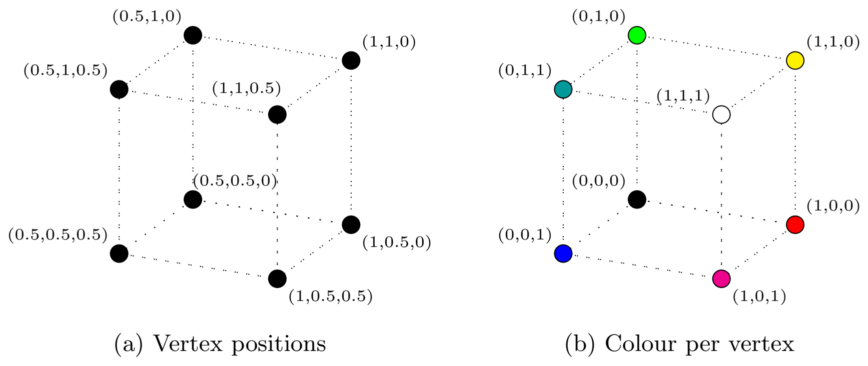


Figure 5.2: Volume bounding box. Proxy cube structure for the ray entry and exit point computation.

nates obtained in the previous pass, the ray direction and length is computed by substracting the back and front colour values at per pixel level.

The pipeline of the rendering passes is depicted in Figure 5.3.

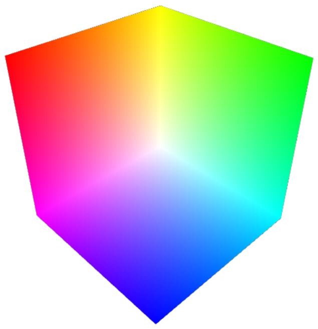
The second pass calculates the ray direction and actually performs the ray traversal. Each ray traverses the cube, sampling the volume data at equidistant intervals. At each sampling interval, a scalar value is obtained. This operation is usually done by re-sampling the volume data with trilinear interpolation. In WebGL, the data is sampled using the *ImageTextureAtlas* approach, as presented in Chapter 3. This sampled value is accumulated along the ray using *alpha blending* in front-to-back or back-to-front order. (see composite Equation 2.3 in Chapter 2).

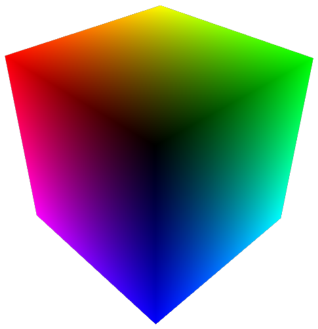
The obtained scalar value at each sample interval can be mapped to a given colour and opacity by providing a transfer function (TF) or alternative methods to alter the accumulation composition. In this way, colour is added or characteristics are enhanced in the volume data (see Chapter 4). When the ray finishes the bounding box traversal, the accumulated colour and opacity is set to the pixel from which the ray has been originally generated.

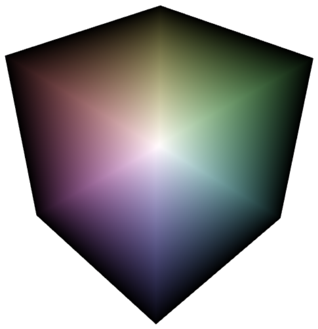
The pseudo-code shown in Listing 5.1 summarizes the ray-casting algorithm.

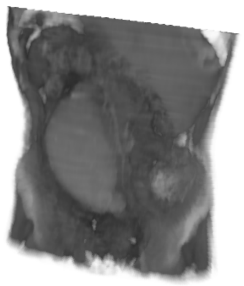
5.3 Hybrid multi-pass volume ray-casting

In some scenes, volumetric and surface data must be rendered together. For example, in medicine, flow streamlines are typically visualised combined with MRI scanned data (Stankovic et al., 2014). Also, medical surgical training simulations require the interaction between 3D modelled objects


(a) **First pass**: Ray exit 3D texture coordinates


(b) **Second pass**: Ray entry 3D texture coordinates


(c) **Second pass**: Ray direction (ray exit - ray entry) → (c)-(a)


(d) **Second pass**: Ray traversal

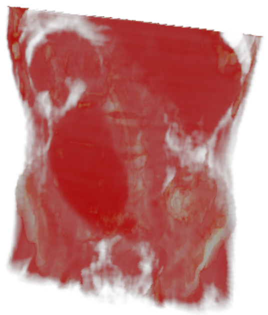

(e) **Second pass**: Ray traversal with TF

Figure 5.3: WebGL ray-casting multi-pass approach. First pass: a) screen space projected 3D texture coordinates of the ray exit position in the VBB encoded as *RGB*. Second pass: b) screen space projected 3D texture coordinates of the ray entry position in the VBB as *RGB*, c) ray direction (back−front) and d, e) ray traversal result.

Listing 5.1: Ray-casting pseudo-algorithm

```
**For** each pixel **in** the screen
  Initialize number **of** steps $S$
  Compute the ray position $r_p$, ray direction $r_d$
                 **and** maximun distance $D$
  Compute interval step $s$
  **For** $i = 0; i < S; i = i + 1$
    Interpolate sample at current $r_p$
    Compute colour $C$ **and** opacity $\alpha$ **with** $TF$
    Accumulate $C$ **and** $\alpha$
    **If** $\alpha >= 1.0$ **or** $r_p > VBB$
        break;
    **End If**
    Increment $r_p$ **with** $s$
  **End For**
**End For**
```

and real volumetric data in order to simulate surgery using haptic devices (Vlasov et al., 2012; Xu et al., 2016). This section describes additional procedures that must be performed in the two pass ray-casting algorithm to solve the integration of 3D geometry and 3D volumetric datasets.

Volumetric models can be considered as semi-transparent objects, and therefore, there are two cases to take into account when rendering 3D geometry and volume data together. In the first case, the rendering order and the blending process must deal with 3D geometries in front of the volume (occlusion of the volume) and 3D geometries behind the volume (occlusion of the 3D object). In the second case, changes in the ray-casting algorithm have to be added to support the rendering of 3D objects inside or intersecting with the volume data. The following subsections present solutions to solve these two cases. Section 5.3.1 presents the shared part of the algorithm. Section 5.3.2 completes the algorithm for the first case and Section 5.3.3 for the second one.

5.3.1 Multi-pass rendering

To support the rendering of 3D polygonal meshes and volumetric datasets, the ray-casting method presented at Section 5.2 must be modified with additional rendering passes. These new passes are required to gather additional information that will be used during the ray casting traversal (the second rendering pass).

For the complete rendering of the scene, when mixing opaque and transparent objects, the rendering order of the objects is important. In traditional computer graphics techniques, when rendering transparent and opaque polygonal meshes, they are sorted in z-depth order and rendered from back-to-front. The multi-pass algorithm follows a similar solution when rendering volumes and 3D opaque meshes.

These are five passes that are necessary to support the rendering of 3D meshes and volume data (see Figure 5.4):

First pass (Depth pass, Figure 5.4a): In the first pass, the depth of all the 3D meshes in the scene are rendered into a Frame Buffer Object (FBO). A simple shader is executed per 3D object in which the *fragment shader* outputs the current depth encoded in the colour channels as RGBA.

Second pass (Colour of surface objects, Figure 5.4b): In the second pass, all the 3D meshes are rendered into a separate FBO to store their colour information. In this case, the desired shader is used.

Third pass (Back cube depth, Figure 5.4c): With the same shader used in the first pass, the depth spatial data of the back side of the volume bounding box is rendered into a FBO encoded in the colour channels as RGBA.

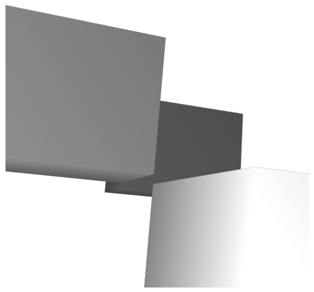
Fourth pass (Back coordinates of the cube, Figure 5.4d): This pass is the same as the first pass of the previous algorithm in Section 5.2.

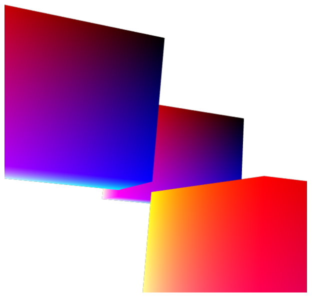
Fifth pass (Ray traversal and 3D surfaces, Figure 5.4e): In this pass, 3D objects are rendered in the screen along volume objects computing the ray-casting traversal. Next two sections describe the fifth pass for each of the two cases considered at the beginning of Section 5.3.

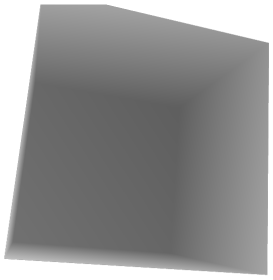
As an immediate consideration of the presented passes, the objects must be rendered in certain order to correctly obtain the final composition. A downside of this multi-pass approach is that it requires four FBOs to compose the final rendering in contrast to one FBO for the two-pass volume rendering method, without hybrid rendering. These extra rendering passes are necessary to render 3D objects and volume data together.

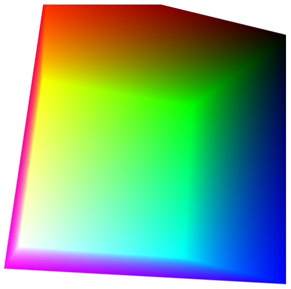
5.3.2 Blending

The rendering order of the objects in the scene must be specific. Volume objects are semi-transparent and thus, if an opaque 3D mesh is behind the volume object, it will be partially occluded. As a consequence, the 3D object


(a) First pass


(b) Second pass


(c) Third pass


(d) Fourth pass

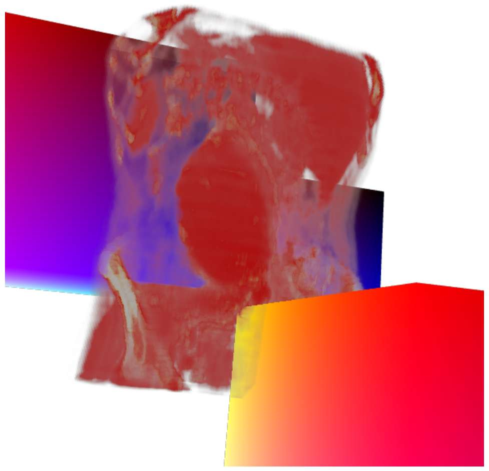
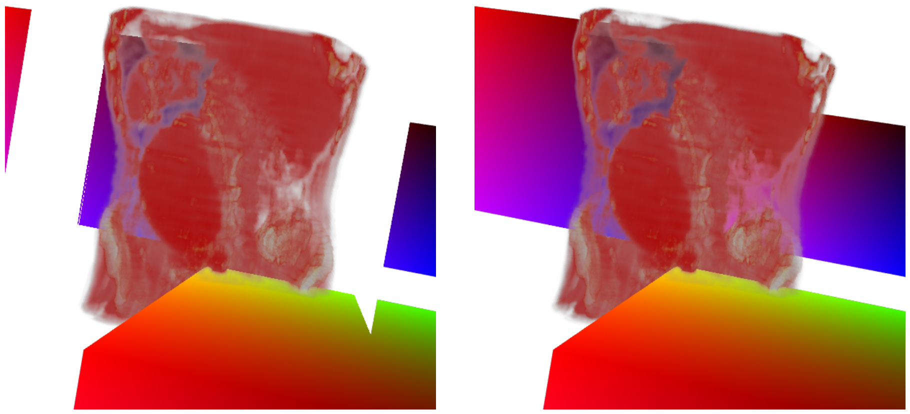

(e) Fifth pass

Figure 5.4: Multi-pass ray-casting pipeline to render volume and 3D surface meshes.

partially contributes to the final rendering output. This step is traditionally known as *alpha blending*.

When the ray-casting shaders are executing the final rendering pass (fifth pass), the colour and alpha output of the computed pixels are going to be rendered in the screen buffer. This algorithm uses a front-to-back blending composition. Figure 5.5 shows that the alpha blending is necessary to render correctly any object behind the volume object. Figure 5.5a shows that without the blending enabled, the proxy cube is visible as a white area around the volumetric dataset producing rendering artifacts in the polygonal meshes. In Figure 5.5b, with the blending enabled, the GPU can mix the different models correctly.



(a) Blend off (b) Blend on

Figure 5.5: Alpha blending of volumetric objects and 3D surface meshes. a) Rendering the volume with blending disabled (with rendering artifacts). b) Front-to-back alpha blending enabled (without rendering artifacts).

This algorithm assumes that there are only opaque 3D objects in the scene. That is, it assumes there are not transparent 3D polygonal objects. Minor adjustments of the presented algorithm could be implemented to support 3D transparent objects.

5.3.3 Intersection with 3D surface mesh data

The ray-casting *fragment shader* (see Listing 5.1) must be modified to support blending of the 3D meshes and the volume. In this section, algorithm changes are presented with GLSL code samples.

The first problem to solve is the use of the depth information coming from the 3D objects at the ray-casting final pass. During the ray-casting traversal, rays advance inside the volume bounding box fetching data from the volume texture. During this traversal, it is required to determine at each step if the ray collides with a 3D object or not.

To achieve this goal, the depth data produced by passes 1 and 3 (see Section 5.3.1) must be accessed at the fifth pass of the ray-casting. The Frame Buffer Objects (FBO) produced in those passes are created as $RGBA$ textures, since WebGL has not support for *Float* textures. A WebGL extension has been provided later on, in order to support depth textures. As a fallback solution to all WebGL compatible devices, packing and unpacking functions can be used. The depth information is packed into multiple 8-bit colour channels ($RGBA$) in the passes 1 and 3. This $RGBA$ information is then unpacked to a float 24-bit value in the fifth pass using the GLSL code samples at Listings 5.2 and 5.3.

Listing 5.2: Float packing function.

```
function packFloat(in float value){
  const vec4 bitSh = vec4(256.0*256.0*256.0,
                          256.0*256.0, 256.0, 1.0);
  const vec4 bitMsk = vec4(0.0, 1.0/256.0,
                           1.0/256.0, 1.0/256.0);
  vec4 res = fract(value * bitSh);
  res -= res.xxyz * bitMsk;
  return res
}
```

Listing 5.3: Float unpacking function.

```
function unpackFloat(in vec4 value){
  const vec4 bitSh = vec4(1.0/(256.0*256.0*256.0),
                          1.0/(256.0*256.0),
                          1.0/256.0, 1.0);
  return(dot(value, bitSh));
}
```

In the *fragment shader*, parameters are initialized per pixel to start the ray-casting process: *ray_origin, ray_direction, ray_steps...* In addition to these parameters, depth data from previous passes are also fetched to initialize the ray depth step in the same coordinate space as the scene objects. The GLSL code at Listing 5.4 stores the depth information in independent variables that will be used in the final comparison: `backDepth` contains the value of the distance from the camera position to the exit position of the ray in the VBB, the `frontDepth` is the distance value from the camera position to the ray entry point in the VBB and the `surfDepth` is the distance value from the camera position to the surface. The `depthStep` is the value to increment on each iteration of the ray traversal loop.

Listing 5.4: Fething depth data before ray-traversal.

```
float backDepth = 1.0 - unpackFloat(texture2D(uBackDepth, ↩
    texD));
float frontDepth = 1.0 - gl_FragCoord.z*gl_FragCoord.w;
float surfDepth = 1.0 - unpackFloat(texture2D(uDepthSurface, ↩
    texD));
float depthStep = (backDepth - frontDepth)/ray_steps;
```

During the ray casting traversal, the decoded depth value is compared with the depth of the ray, while the *ray step* and the *depth step* are iteratively incremented.

The depth test compares the ray distance to the 3D object, including its thickness, tracking the position of the ray-casting (*ray_depth*). If the 3D object is hit, the accumulated colour and opacity is updated with the colour of the object. In addition, it is possible to update the depth value of the pixel with the `surfDepth` (surface depth) value using the `EXT_frag_depth` WebGL extension. The GLSL code in Listing 5.5 shows that procedure and that the ray-casting loop finishes with the *break* instruction since an opaque object has been hit:

Listing 5.5: Ray intersection depth test.

```
if((ray_depth-surfDepth) > -(depthStep))
{
  accum.rgb = (accum.a * accum.rgb) + ((1.0 - accum.a) * ↩
      objectcolour.rgb);
  accum.a = accum.a * accum.a + (1.0-accum.a);
  final_depth = surfDepth;
  break;
}
```

The ray-traversal per pixel can finish in three different cases:

1) The ray traverses the volume completely.

2) The ray fills the alpha value during the ray traversal (early-ray termination)

3) The ray hits an opaque object.

The first case uses the traditional ray-casting algorithm, but the possibility of having 3D objects behind the volume has to be taken into account. The second case falls back to the traditional ray-casting algorithm and the accumulated colour and depth is returned as the colour of pixel in the final rendered image. The third case is when the ray hits a 3D object inside the volume. Each scenario is described in the following sections.

1) Complete ray traversal, accumulated alpha less than 1.0

When a ray traverses the volume dataset, a colour is returned with an alpha value less than 1.0, which means that it is semi-transparent. Therefore, the final colour in that pixel has to take into account the possibility of any 3D objects that might be behind the volume. This situation is solved by WebGL blending directives. As the 3D objects have been rendered before the volume, the colour buffer has already the 3D object colour information. Rendering the volume into the colour buffer activates directly the WebGL blending functions and the expected behaviour is achieved (see Figure 5.5).

2, 3) Early ray termination, accumulated alpha equal 1.0

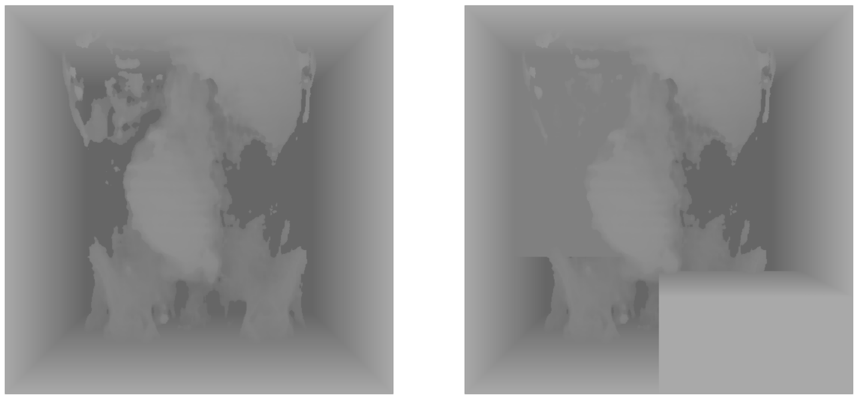
A common ray-casting acceleration technique is known as *early ray termination* (see Kruger and Westermann (2003b)). In this technique the ray traversal is interrupted when the accumulated opacity reaches an opaque value (accumulated alpha equal 1.0) before the ray gets out of the boundaries of the volume bounding box.

The *early ray termination* checks for the state of the accumulated opacity inside the ray traversal loop before the computing next ray step. The GLSL code at Listing 5.6 shows how to check if the current ray position is inside the volume bounding box or rather the accumulated alpha requires an *early ray termination*.

Listing 5.6: Early ray termination.

```
if(accum.a>=1.0 || any(greaterThan(rpos.xyz, vec3(1.0, 1.0, ↩
    1.0)))
  break;
```

The `break` statement will abruptly interrupt the loop of the ray traversal. As previously stated in this section, the essence of this technique is also being applied with 3D surface data inside the volume. When the ray intersects the mesh, the accumulated colour and opacity is blended with the 3D mesh surface colour properties. As a result, the accumulated opacity reaches a totally opaque value, and thus, the ray is early-terminated.



(a) Ray depth (b) Ray and surface depth

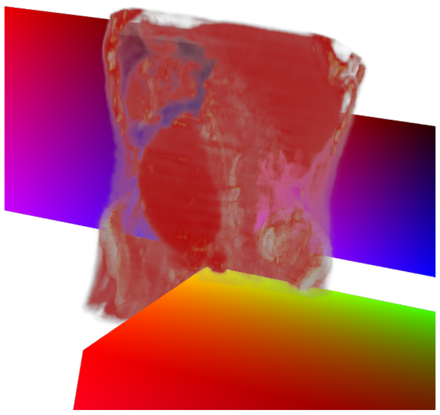
Figure 5.6: Early ray termination and 3D geometries surface intersection with the `aorta` dataset. a) The depth of the volume when the accumulated opacity reaches 1.0. b) The ray termination when a 3D surface is intersected.

Figure 5.6 shows a normalized rendering output of the ray depth using the *aorta* dataset. With the *early ray termination* the shape of the 3D object can be shown. In the last shader pass, the *early ray termination* saves unnecessary computations. This performance improvement is proportional to the area of the volume intersected by the 3D object and to the proximity of the 3D object to the camera.

The presented WebGL implementation of the algorithm has been tested in desktop and mobile devices. Figure 5.7 shows how 3D coloured and textured boxes can be integrated with volumetric datasets.

Listing 5.7 shows an updated ray-casting pseudo-algorithm accounting for the ray depth in the global scene and the depth test during the ray-traversal.

Figure 5.8 shows different situations regarding the relative positions of the volume, the boxes and the camera. There are pixels that corresponds to a box placed in front of, behind and in the middle of the volume. All these


(a) Per vertex colour

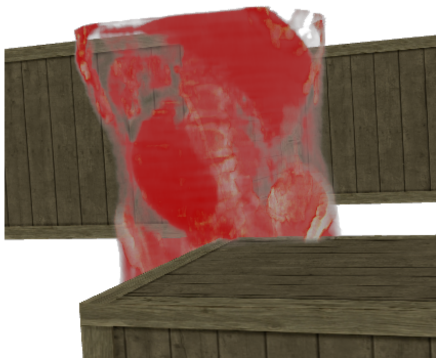
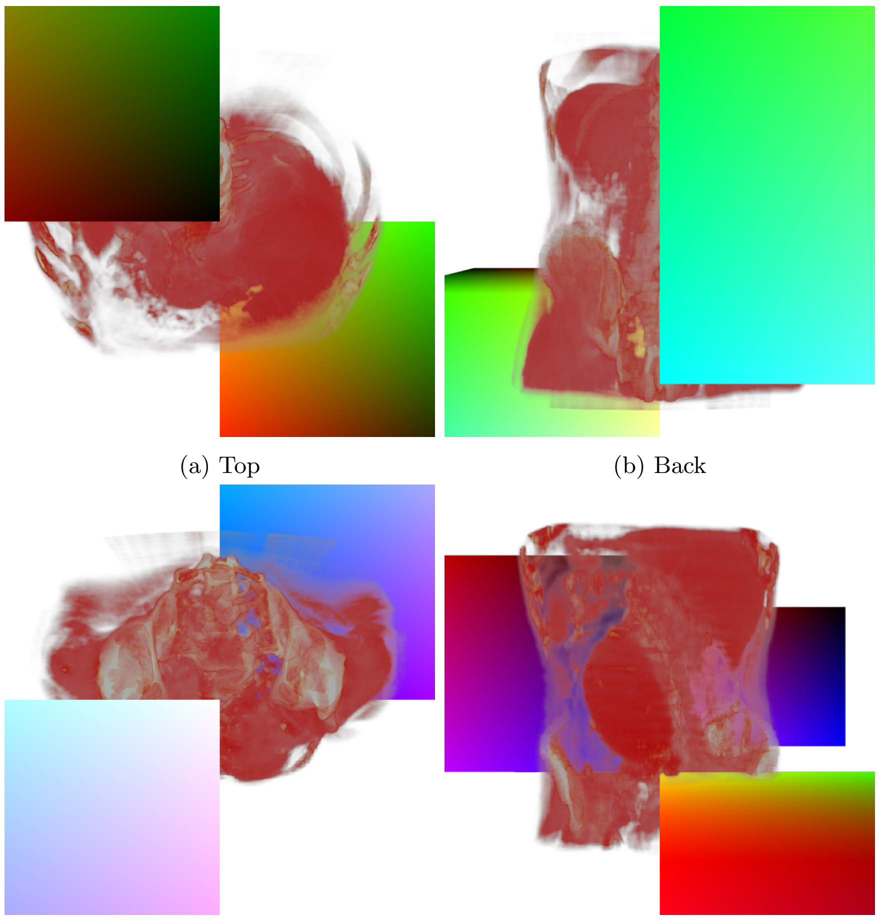

(b) Texture mapping

Figure 5.7: WebGL volume rendering of volumetric data and 3D geometry. a) 3D box geometries with per vertex colour. b) 3D box geometries with textures.

Listing 5.7: Hybrid ray-casting pseudo-algorithm

```
For each pixel in the screen
  Initialize number of steps $S$
  Compute the ray position $r_p$, ray direction $r_d$, ray_depth $r_z$ and
      maximun distance $D$
  Compute interval step $s$
  Compute depth interval step $s_z$
  For $i = 0; i < S; i = i + 1$
    Interpolate sample at current $r_p$
    Compute colour $C$ and opacity $\alpha$ with $TF$
    Compute depth test with $r_z$
    Accumulate $C$ and $\alpha$
    If $\alpha >= 1.0$ or $r_p > VBB$
        break;
    End If
    Increment $r_p$ with $s$
    Increment $r_z$ with $s_z$
  End For
End For
```



(a) Top (b) Back

(c) Bottom (d) Front

Figure 5.8: Volume rendering and 3D geometry intersection from different point of views using the aorta dataset.

situations produce visual renderings that are visually correct.

5.4 WebGL extensions

The implementation of the GLSL code is targeting WebGL 1.0. But WebGL 1.0 provides some official extensions that can be used to produce better results and to simplify the GLSL code.

The *WebGL_depth_texture* extension provides the possibility of declaring *Float* textures for the depth component. Using this extension, the pack and unpack functions used in the Section 5.3.3 can be replaced by a direct access to the depth texture.

The *WebGL EXT_frag_depth* extension provides the possibility of modifying the depth value in the *fragment shader*. Using this extension provides more flexibility in the depth buffer reading, testing and writing. Ultimately, it could lead to a reduction in the number of passes in the presented methodology.

The *WebGL_draw_buffers* extension provides multiple colour buffers and colour render targets that could be use from the fragmet shaders. This would allow to reduce the number of FBO passes used to increase the overall performance.

Chapter 6

Progressive volume rendering

Due to the evolution in CT and MRI image scanning technologies, the output resolution of the acquired datasets has increased over time. Additionally, new use cases which require a higher degree of resolution have emerged, i.e. quality inspection of 3D printed parts. The medical domain has also benefitted from the increase of resolution. Finer detail acquisition translates to easier diagnosis and better data interpretation. However, the increasing resolution and memory requirements trend implies that the computational power for the visualisation of such datasets also needs to be increased.

In direct volume rendering approaches for high-resolution volumetric datasets, a higher degree of sampling is required to accurately visualise the data. In these datasets, smaller details also contribute to the visualisation outcome. However, as more sampling is required, higher memory bandwidth is required as well, leaving out a great number of GPUs as suitable acceleration solutions for the real-time volumetric rendering. This situation is more pronounced in an ubiquitous scenario: the variation of computational power is as big as it can be: From low power devices such as mobile phones to high performance desktop PCs.

A different approach is required in order to allow the visualisation of these datasets. In this regard, a progressive rendering approach postulates as a viable solution to allow an interactive rendering of high resolution volume data that can be adjusted to the computational power of any device. This chapter presents a WebGL-based progressive ray-casting algorithm that enables the interactive visualisation of any dataset that fits on the targeted GPU memory. Other desktop-based alternatives will require per device installation and maintenance. Therefore, the software may not be accessible for everyone. In contrast, the proposed approach will be available for any device with access to a modern web browser.

This chapter is structured as follows. Section 6.1 presents the single-pass ray-casting algorithm and Section 6.2 describes the proposed progressive ray-casting algorithm based on the previous algorithm.

6.1 Single-pass ray-casting

The single-pass ray-casting method was presented by Stegmaier et al. (2005). Later on, Mobeen and Feng (2012b) adopted this method to WebGL using Congote et al. (2011) texture atlas approach. Both methods use a volume bounding box (VBB) as a proxy geometry (see Section 5.2 in Chapter 5). The two-pass approach requires the use of an additional offscreen buffer (FBO) to perform the ray entry and exit computation. However, the first pass can be avoided with the method presented by Stegmaier et al. (2005), and the volume rendering can be performed in a single-pass of the graphics pipeline.

In the single-pass ray-casting a unitary cube is used as a VBB. The vertices of the VBB are multiplied with the *ModelView* and *Projection* matrices into the clip space. In this process, the vertices are assembled into triangles and the vertex positions are interpolated by the rasterization process. Since the VBB is unitary, the interpolated object space positions represent the 3D texture coordinates of the ray entry points.

In the *fragment shader* the camera position can be obtained from the inverse of the *ModelView* matrix. The ray direction is calculated by subtracting the interpolated object space position of the VBB vertices with the camera position. In this manner, rays are casted from the camera position towards the VBB.

The ray origin is positioned at the front side of the cube (from the camera view perspective). Then, there are two possible methods to perform the ray-traversal:

1. Mobeen and Feng (2012b) proposed the use of a ray-box intersection test to compute if a given fragment should be discarded. In this thesis, the Axis Aligned Bounding Box (AABB) ray-box *slabs* based intersection algorithm by (Kay and Kajiya, 1986b,a) has been used. The intersection test provides the exit point of the ray in the VBB. With the exit and entry points in the VBB, the ray is discretized into a fixed number of steps. Then, a ray-VBB traversal is initiated with a loop.

2. Since the VBB is unitary, the maximum length of the ray is $\sqrt{2}$. The ray is discretized into equidistant size steps. Then, a loop is started to sample each step of the ray along the computed direction. On each iteration (step increment) an out-of-bounds test is performed terminating the ray loop if the step position in object space is outside the VBB.

At each step of the ray traversal the position of the ray in object space represents the 3D texture coordinate required to fetch the volume data. In WebGL 1.0 the volume data is sampled using the *ImageTextureAtlas* (see Chapter 3). During the ray traversal the opacity and colour is accumulated and blended in front-to-back order approximating the volume rendering integral presented in Chapter 2.

6.2 Progressive ray-casting

To visualise detailed sections of the volume obtained from high resolution scans, the volume data is required to be sampled at a high resolution, that should be high enough to clearly render isosurfaces without skipping voxels during the ray traversal. Díaz-García et al. (2018) proposed a solution for the visualisation of large datasets in mobile devices with the use of progressive GPU ray-casting. Their approach targeted the OpenGL ES 3.0 API (desktop and mobile).

The progressive approach prevents the application from stalling, that is, it preserves user interaction. This is achieved by distributing the rendering task over subsequent rendering frames after every user interaction. The control is returned to the application's main event loop frequently, drawing into the screen framebuffer in short periods of time. In this way, the user can interrupt the progressive rendering at any time. This approach does not provide real-time high-quality rendering, but it allows user to interact with the visualization at any time without stalling the browser.

Based on the *FSlab* approach by Díaz-García et al. (2018), the single-pass real-time ray-casting method presented in Section 6.1 has been modified in order to create a WebGL compliant progressive volume rendering algorithm.

The progressive rendering behaves like a refinement algorithm where the whole volume is already visualised from the start, and details are progressively rendered until completion. Two rendering cases are taken into account:

1. When the user interacts with the visualization (rotation, pan, zoom, etc.)

a real-time volume rendering algorithm is used to generate a low-quality, but fast rendering.

2. When the image remains still (no user interaction, static scene), a high quality image begins to appear in the screen by progressively rendering the volume. Every frame till the next user interaction, the rendering receives more and more details.

The following subsections describe the proposed WebGL progressive volume rendering algorithm.

6.2.1 User interaction

When the user is interacting with the visualization, the image needs to be reconstructed in real-time to allow the user to navigate through the volume data. Fine details are not visible during the interaction. However, the user can locate the volume data in the scene and interact with it accordingly. The number of steps (number of 3D samples on each casted ray) must be kept low in order to allow real-time performance. In this case, the single-pass ray-casting algorithm presented in Section 6.1 is used.

6.2.2 No user interaction

When there is no user interaction, a high-quality rendering process is started. For this purpose the volume data is rendered in a progressive manner, i.e, the details of the volume data are refined over time in the view direction in the following frames.

To achieve a high-quality render, the casted rays are discretized into a large number of points along the ray (steps) to be sampled. This quantity of steps can not be computed in one frame for large datasets, otherwise the visualization would be stalled and the interaction would not be possible. In the presented progressive approach, the casted rays are split into segments of fixed length in the view direction (slabs). Thus, each slab is composed of ray segments of same length that can be computed in a short period of time.

The slab is further discretized into sampling points (steps) along the ray segment. The sampling points will be used during the ray-casting loop to fetch the volume data and perform the rendering integral for the current slab. Figure 6.1 illustrates the slab and steps concepts in a single ray.

The high-quality rendering is completed when all the slabs are rendered. Both the number of slabs and the number of steps can be adjusted to ade-

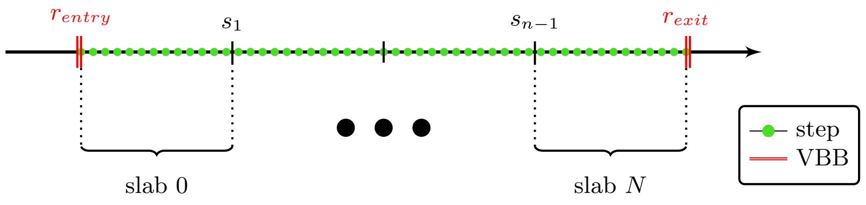


Figure 6.1: Illustration of the slabs and sample points in a single ray

quate the quality and rendering time to the computing power of the targeted device.

The progressive rendering is a cycle that renders one slab after another, serially in front-to-back order. Therefore, the number of slabs determines the number of iterations of the refinement loop.

The first step, after the user stops to interact with the visualization is to clear a high-resolution texture (T_{high}). This texture is used to accumulate the output result of each slab until completion. Then, the refinement loop is initiated in order to render each slab in front-to-back order. For each iteration, three steps are performed: two rendering passes and a copy operation:

i) **First pass (slab rendering)**: This render pass takes as input the accumulated result in T_{high} as the initial value of accumulated colour and opacity in every pixel. Then, the ray segment (slab) of the current iteration is rendered using the single-pass volume ray-casting algorithm with a fixed number of steps (i.e 40). Because the length of the slab is a fraction of the whole ray, that small number of steps samples the slab in fine detail (high resolution). The output of the projected slab rendering blended with all the previous slabs (T_{high}) is written to an offscreen buffer (FBO). Listing 6.1 shows the pseudo-code for the slab rendering.

ii) **Slabs buffering**: The output rendering of the previous pass is copied to T_{high} texture with the WebGL *copyTexSubImage2D* API function. The content of the texture is overwritten in this step.

iii) **Second pass (remaining slabs rendering)**: This render pass also takes as input the accumulated result in T_{high} (in *ii*) as the initial value of accumulated colour and opacity in every pixel. Then it renders the remaining ray segment in the view direction with a low count fixed number of steps (i.e. 100). The single-pass ray-casting algorithm is also used for this purpose. The low step count rendering computed in this pass is blended with the accumulated high-quality T_{high} rendering. Listing 6.2 shows the pseudo-code to render the rest of the ray, taking as input the current slab iteration. It shares the same structure as the high-quality render pass (first pass), the only difference is the ray segment computation (*ComputeRemainSlabSegment* function).

Listing 6.1: Slab rendering algorithm with a fixed number of steps

```
Function Initialize()
  number of steps → $N_{step}$
  number of slabs → $N_{slab}$
  colour value → $C$
  alpha value → $\alpha$
End Function

Function ComputeSlabSegment($slab_i$, $\vec{r}_{pos}$, $\vec{camera}_{pos}$)
  $\vec{r}_{dir} \leftarrow \|\vec{r}_{pos} - \vec{camera}_{pos}\|$
  $t_{entry}$, $t_{exit}$ with AABB Ray−Box intersection
  $\vec{r}_{entry} = \vec{r}_{pos} + (\vec{r}_{dir} * t_{entry})$
  $\vec{r}_{exit} = \vec{r}_{pos} + (\vec{r}_{dir} * t_{exit})$
  $slab_{len} = (|\vec{r}_{exit} - \vec{r}_{entry}|)/N_{slab}$
  $\vec{r}_{start} = \vec{r}_{pos} + (\vec{r}_{dir} * (slab_i * slab_{len}))$
  $\vec{r}_{end} = \vec{r}_{start} + (\vec{r}_{dir} * slab_{len})$
  $\vec{S}_{incr} = (\vec{r}_{end} - \vec{r}_{start})/N_{step}$
  $return \leftarrow \vec{r}_{start}, \vec{r}_{end}, \vec{S}_{incr}$
End Function

Function RenderSlabHigh($slab_i$)
  For each pixel in the screen
    Fetch $C, \alpha \leftarrow T_{high}$
    $\vec{r}_{pos} \leftarrow$ varying VBB position
    If $\alpha >= 1.0$
      break
    End If
    $\vec{r}_{start}, \vec{r}_{end}, \vec{S}_{incr} \leftarrow$ ComputeSlabSegment($slab_i$, $\vec{r}_{pos}$, $\vec{camera}_{pos}$)
    For $i = 0; i < N_{step}; i = i + 1$
      $d \leftarrow$ Sample volume at current $r_p$
      $C$, $\alpha \leftarrow TransferFunction(d)$
      Blend and accumulate $C$ and $\alpha$
      If $\alpha >= 1.0$ or $r_p > VBB$
         break
      End If
      Increment $r_p$ with $S_{incr}$
    End For
    $FBO \leftarrow (C, \alpha)$
  End For
End Function
```

Listing 6.2: Ray-casting pseudo-algorithm

```
Function ComputeRemainSlabSegment($slab_i, \vec{r}_{pos}, \vec{camera}_{pos}$)
  $\vec{r}_{dir} \leftarrow \|\vec{r}_{pos} - \vec{camera}_{pos}\|$
  $t_{entry}$, $t_{exit}$ with AABB Ray−Box intersection
  $\vec{r}_{entry} = \vec{r}_{pos} + (\vec{r}_{dir} * t_{entry})$
  $\vec{r}_{exit} = \vec{r}_{pos} + (\vec{r}_{dir} * t_{exit})$
  $\vec{S}_{incr} = (\vec{r}_{end} - \vec{r}_{start})/N_{step}$
  $slab_{len} = (|\vec{r}_{exit} - \vec{r}_{entry}|)/N_{slab}$
  $\vec{r}_{start} = \vec{r}_{pos} + (\vec{r}_{dir} * ((slab_i + 1.0) * slab_{len}))$
  $return \leftarrow \vec{r}_{start}, \vec{r}_{end}, \vec{S}_{incr}$
End Function

Function RenderSlabLow($slab_i$)
  For each pixel in the screen
    Fetch $C, \alpha \leftarrow T_{high}$
    $\vec{r}_{pos} \leftarrow$ varying VBB position
    If $\alpha >= 1.0$
      break
    End If
    $\vec{r}_{start}, \vec{r}_{end}, \vec{S}_{incr} \leftarrow$ ComputeRemainSlabSegment($slab_i$, $\vec{r}_{pos}$,$\vec{camera}_{pos}$
        )
    For $i = 0; i < N_{step}; i = i + 1$
      $d \leftarrow$ Sample volume at current $r_p$
      $C$, $\alpha \leftarrow TransferFunction(d)$
      Blend and accumulate $C$ and $\alpha$
      If $\alpha >= 1.0$ or $r_p > VBB$
         break
      End If
      Increment $r_p$ **with** $S_{incr}$
    **End For**
    $FBO \leftarrow (C, \alpha)$
  **End For**
**End Function**
```

Figure 6.2 illustrates the steps of the progressive slab rendering. In one hand, a segment of the ray traversal (slab) is rendered incrementally into a high-resolution texture that accumulates the result across frames. In the other hand, at each incremental iteration, the non-traversed ray segments are rendered with a low number of steps taking the high-resolution slab position at the current iteration as the starting point. Both results are combined with front-to-back alpha blending as the final outcome in every frame. Over time, as the high resolution render pass advances incrementally over the ray (accumulating the high resolution result), the low step count render pass will render an smaller length of the ray each time. When all iterations have been completed, the accumulated result in the high resolution texture (T_{high}) will converge to the final high-quality rendering.

The progressive approach implies that the visual outcome improves over time. In the presented algorithm both a high resolution and low resolution rendering passes are combined to show a rendering of the whole volume in every frame. The low resolution pass is necessary to create a more pleasant transition between the low-quality rendering (real-time rendering during interaction) and high-quality rendering (during lack of interaction).

Figure 6.3 shows how the accumulated rendering advances over subsequent frames until completion. It shows how the volume details appear progressively along the view direction.

6.2.3 WebGL constraints

WebGL and JavaScript have added some technical constraints to create a functional web compatible progressive volume rendering algorithm. The incremental slab rendering into subsequent frames was not possible to implement in WebGL using *for-while* loop statements. On one hand, the different draw calls inside the loop where being batched under the hood and with the lack of synchronization primitives the accumulation results were inconsistent. On the other hand, the loop context was stalled until draw completion, blocking the application and breaking any possible interaction with the visualization.

This problem is solved using an event driven approach (see Listing 6.3).

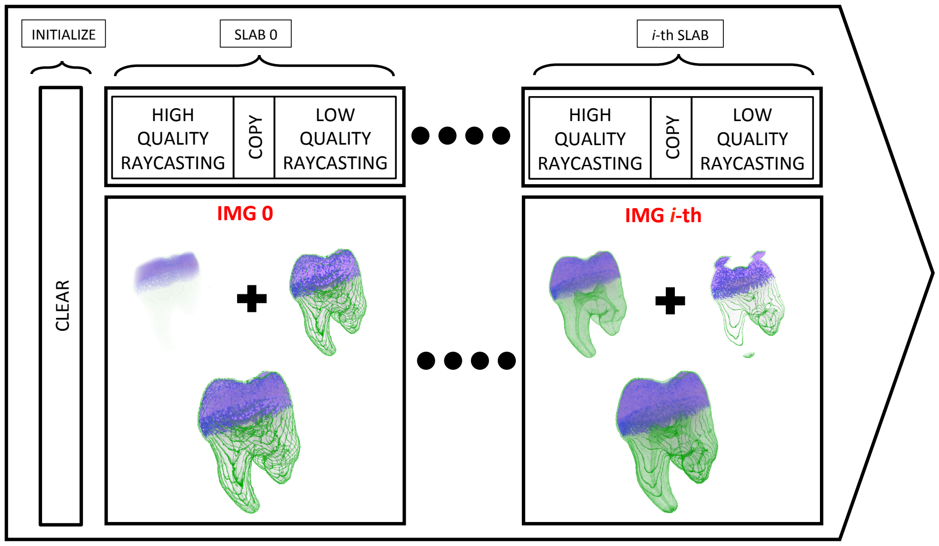
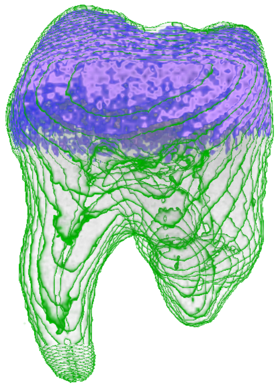


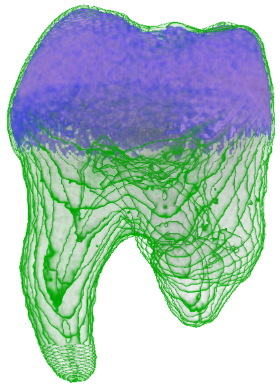
Figure 6.2: Rendering evolution after the user stops the interaction with the volume. The initialization phase clears the information from previous progressive renders. Then, the first slab is rendered (SLAB 0) as a composition of a set of high quality ray-casting steps and a low quality ray-casting from the last point to the end of the volume. As time progresses, the rest of slabs (i-th SLAB) are rendered using the already calculated high quality ray-casting information.

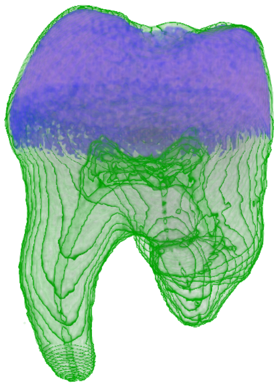
A recursive function is used that calls itself with a *setTimeout* method. Each recursive call creates a new event every time, simulating a loop. Using this recursive strategy, a conditional statement for the loop exit condition (user interaction) is evaluated after each slab rendering. This allows to interrupt the recursive progressive rendering whenever the user interacts with the 3D scene.

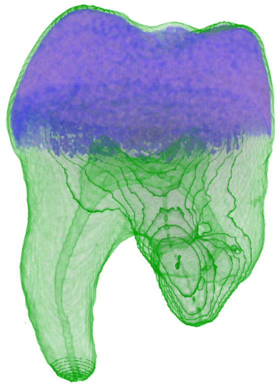
The recursive method has a direct consequence in the WebGL pipeline, as the concatenation of the output of a pass and the output of the previous pass has to be done via global textures. The use of an intermediate texture T_{high} and one additional copy operation (see Section 6.2.2) were required to correctly accumulate slab render outputs over subsequent frames.

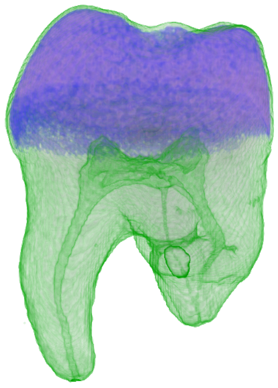
Since the number of incremental iterations can be easily changed through JavaScript, it is easy to allow the user to dynamically change the rendering quality. The number of steps can be fixed to a value in the range of [25-40] steps to target an ubiquitous deployment: a progressive high-quality rendering visualization that will work in any device. Since the low step rendering can be handled by any device, this approach makes suitable vol-


(a) Slab 25


(b) Slab 50


(c) Slab 75


(d) Slab 100


(e) Slab 125

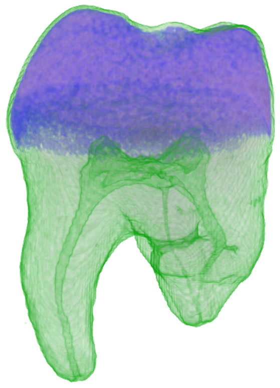

(f) Slab 150

Figure 6.3: Progressive rendering of the `tooth` dataset with 150 slabs of 40 steps. Different outputs are shown every 25 slabs until completion.

Listing 6.3: Progressive refinament loop

```
$slab_i = 0$
$N_{slab} = 150$
Clear texture $T_{high}$
**Function** Progressive()
  **If** $slab_i < N_{slab}$ **and** $onInteraction == False$
    $buffer_{tmp}$ = RenderSlabHigh($slab_i$, $T_{high}$)
    $copyTexSubImage2D$($buffer_{tmp}$, $T_{high}$)
    $buffer_{screen}$ = RenderSlabLow($slab_i$, $T_{high}$)
    $slab_i = slab_i + 1$
    $setTimeout$(Progressive, 1)
  **End if**
**End Function**
```

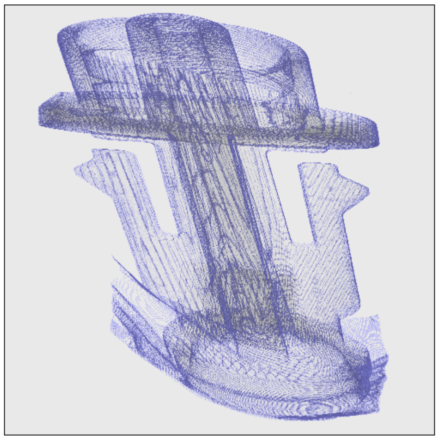
ume rendering to devices with low compute power. However, to obtain the best performance in GPUs of high computation power, the number of steps should be higher.

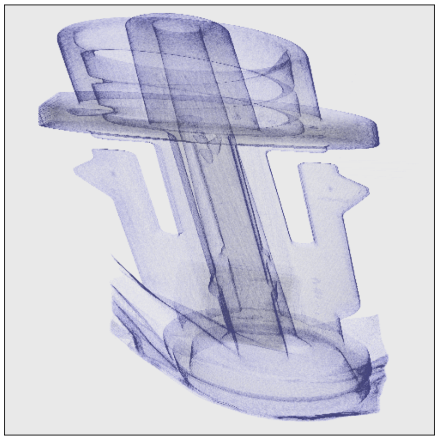
Figure 6.4 shows the resolution difference between a real-time ray-casting approach and the presented progressive approach using the `aorta` and the `plastic injected mould part` datasets. Rendering the volume with the progressive approach leads to a clearer and smoother reconstruction in Figures 6.4b and 6.4d. As it can be seen in Figures 6.4a and 6.4b, a higher rendering quality is obtained when isosurfaces are being displayed.

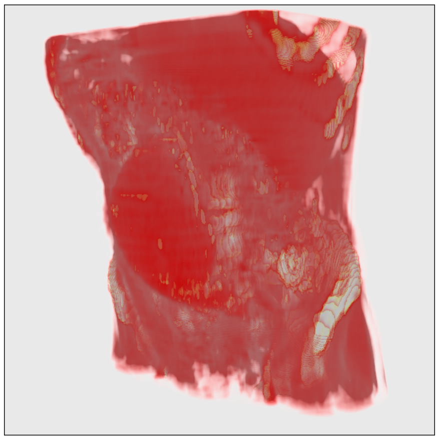
In contrast to the approach proposed by Díaz-García et al. (2018) based in the two-pass volume ray-casting approach, the single-pass ray-casting method (see Section 6.1) computes the ray entry-exit points with a ray-box intersection test in the *fragment shader*. Therefore, there is no need of a separate step for this computation.

Additionally, the new presented algorithm in this chapter simplifies the low resolution render pass by only using a low step count render pass for the non-traversed ray segments and directly rendering the combined result in the screen buffer. This step could also be computed into a low resolution off-screen framebuffer, as Díaz-García et al. (2018) presented, but then it would require an additional copy and blending step, due to WebGL limitations.

The presented progressive volume rendering algorithm is validated in Chapter 9 with a quality inspection use case of the `plastic injected mould part` dataset (see Figure 6.4a and 6.4b).


(a) 80 steps (real-time)


(b) 6000 steps (progressive)


(c) 80 steps (real-time)

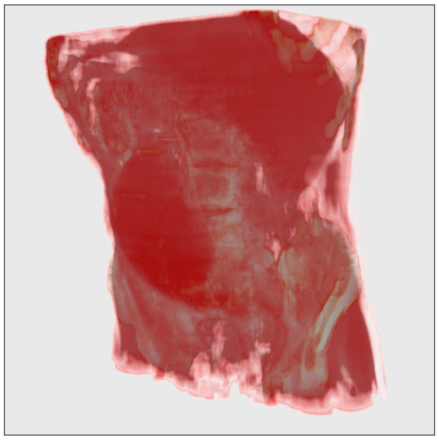

(d) 12000 (progressive)

Figure 6.4: Comparison between real-time ray-casting and progressive ray-casting of the `aorta` dataset $512 \times 512 \times 97$ and the plastic injected mould part $720 \times 720 \times 1770$ dataset.

Chapter 7

Medical volume data visualization

The medical field has undergone a great evolution thanks to the improvements in medical imaging acquisition technologies like Computer Tomography (CT) and Magnetic Resonance Imaging (MRI). Both techniques in combination with computer based image processing algorithms have greatly contributed to improve medical diagnosis. For this reason, the medical field makes direct use of volume rendering visualization.

Unlike other scientific fields, the medical field imposes more restrictions and has more regulations in regard to the use and privacy of patient data. Thus, this restrictions also apply to the use and storage of medical imaging. The medical healthcare system is always looking for new information systems and better forms of storing patient data. It can be difficult to distribute and share data between peers, hospitals or countries, due to current law regulations and incompatible proprietary data formats. This situation has reinforced the need to use an standardized format to store medical imaging data, which addresses the patient related medical annotations and privacy measures. As a possible solution, the DICOM data format has become the de-facto standard for the storage and exchange of medical image data.

Volumetric data is often used in a large variety of situations: from research and diagnosis to educational purposes. In terms of visualization, interactivity and usability, mobile platforms should provide the same tools as their desktop counterpart. This chapter explores the use of DICOM files for ubiquitous volume rendering, that is, for direct visualisation of volumetric data in browsers. *The implementation approaches that are presented in this chapter showcase the potential of the work of this thesis.* The contributions presented in previous chapters are taken as a base to construct viable solutions for ubiquitous medical volume visualization scenarios.

In pursuit of a ubiquitous medical volumetric visualization, this work considers four main features that volume visualisation must provide: *i)* the support of DICOM volume data, *ii)* the visualisation of segmented data, *ii)* camera navigation inside the 3D volume and *iv)* colourizing the output rendering with a transfer function. These four features are practical real-world requirements that the medical domain needs to visualise volumetric data.

This chapter demonstrates how these features can be solved providing the building blocks that developers can take as reference to provide ubiquitous web based applications. All computing steps: data processing, data visualization and user interaction run in the web browser (client device). Further validation for the approaches of this chapter is presented in Chapter 9, where an ubiquitous web based DICOM volume visualization application is presented: *Mirror4all*.

Section 7.1 presents the fetching and processing of volume data to be visualised in the Web. Section 7.2 shows how segmented volume data can be displayed. Section 7.3 explains extensions to the ray-casting algorithm to visualise the volume from the inside. Finally, Section 7.4 presents a transfer function editor for colourizing the output rendering by means of a dynamic look-up table.

7.1 DICOM dataset visualization

In pursuit of an ubiquitous medical volumetric visualization, the support of DICOM file format is necessary: DICOM is the *de-facto* standard that the software applications of the medical imaging devices use to store their scans. This section presents how volumetric DICOM datasets composed of 2D slices can be processed for interactive visualisation in modern desktop browsers using exclusively open web technologies.

Medical imaging devices do not only produce the actual set of 2D slices. They are linked with a large amount of metadata related to the patient health information and other medical procedures. DICOM is the medical image standard that stores and transfers all the information from and between imaging devices (CT, MRI) and medical image storage repositories (Fernandez-Bayó et al., 2000). The wide utilization of DICOM by all manufacturers had a major impact on usability of the file format. The standard gathers a large set of tags to be read, interpreted and combined in order to achieve coherent transmission of the images for the final viewer.

Section 7.1.1 introduces the technological approach to load DICOM datasets into data structures ready to be used with WebGL and X3DOM (see Chapter 4). Section 7.1.2 presents interactive controls that filter DICOM data for its visualization.

7.1.1 *ImageTextureAtlas* generation in the browser

The generation of the texture atlas is a technical constraint of the WebGL API (see Chapter 3). A solution to this problem is to create this data structure in the background in the client device (transparent to the user). For this purpose, an integration of the native Drag and Drop HTML5 API into any web volume visualisation application provides an easy and transparent interface for the composition process.

The whole volume dataset can be contained in a single DICOM file or split across multiple DICOM files into data slices. In the later case, all files must be loaded and merged in order (split direction) to load the whole volumetric data.

As stated before, the rendering 3D `canvas` can be equipped with drag and drop functionality. The user drags a DICOM dataset and drops them into the rendering context element. These DICOM file(s) must be stored in the file-system of the device where the browser is running.

Cornerstone (2016) JavaScript library provides a set of functions to read and interact with the imaging data stored in DICOM files and it relies on the software library to load DICOM tags, including pixel data. Using this library the DICOM data is loaded into a data buffer within the browser context. Then, the content of this buffer is drawn into a sub-area of a 2D canvas. The 2D canvas is the container for the *ImageTextureAtlas* structure (see Figure 7.1) and it is drawn hidden to the user.

There is a distinction between a volume dataset in a single file or in multiple DICOM files. In the first case, the whole data must be parsed first before rendering any result. On the second case, as each file is loaded separately, the browser displays the slices stored within that file. As more files (slices) are loaded, the image is populated till all the slices of the volume are processed and transferred to the atlas.

The generation of the *ImageTextureAtlas* in the browser is specially interesting because it does not enforce the users to adapt the volume data files to a certain file format. The processing is made automatically in the background. The same approach is used with other image file formats such as NRRD, JPG, PNG, etc.

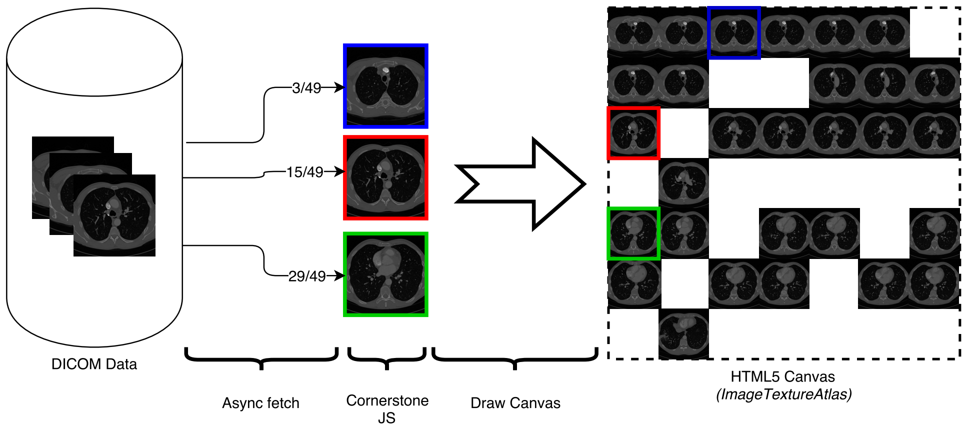


Figure 7.1: Loading process from DICOM images to *ImageTextureAtlas*. The diagram shows an intermediate state where the dataset is not fully loaded and hence, some subtiles are not filled into the final HTML5 *canvas*.

The combination of client side atlas generation with the volume rendering component presented in Chapter 4 provides a general and ubiquitous solution. The 2D `canvas` contains the volume data structured as a *ImageTextureAtlas*. Therefore, this `canvas` is a valid input for an X3D scene.

There are two possible ways to link the composed atlas with the X3DOM volume rendering component (see Chapter 4).

i) Defining the `<canvas>` element as a child node of the *ImageTextureAtlas* node in the X3D scene declaration. Then, the canvas element is filled with the actual DICOM data (see Listing 7.1).

ii) Using the `<canvas>` element as an auxiliary data structure defined outside the X3D scene declaration. The *VolumeData* node uses an empty *ImageTextureAtlas* (no real URL is given). The JavaScript loader draws the atlas texture in this canvas as before, and then, the correct URL is provided as a *DataURL*, which means that the whole volumetric information is encoded and passed in the URL (see Listing 7.2).

Listing 7.1: Canvas as child of the *ImageTextureAtlas* node.

```
<ImageTextureAtlas id="atlas" hiddenChildren="true">
  <canvas id="voxelCanvas"></canvas>
</ImageTextureAtlas>
```

Listing 7.2: Canvas outside the X3D scene definition.

```
<ImageTextureAtlas id="atlas" url="data:"></ImageTextureAtlas>
<script>
document.getElementById("atlas").setAttribute(
  "url",
  document.getElementById("voxelCanvas").toDataURL()
);
</script>
```

Both the visualisation and the data processing (*ImageTextureAtlas* generation) are done in the client side (the users device). In Chapter 3 the *ImageTextureAtlas* generation is considered to be done in a pre-processing step. Therefore it can be done manually or in a server process. This section demonstrates that this step can also transparently be done in the browser. Consequently, it showcases the potential of the proposed web compatible volume data structure.

Solutions from other authors perform the volume rendering in the server side. By contrast, the presented approach performs all the computation in the client side. This facilitates the deployment of volume visualisation applications, reduces communication complexity (securization) and allows to easily scale resources as the main computation (rendering) is distributed across client devices.

7.1.2 Dynamic window level selection

DICOM files contain rich information that can be used in the UI presentation and during the rendering stage. Some metadata attributes can be added to the UI (capture date, acquisition machine, software version, etc.) and give information about the imaging data itself, i.e. the number of slices and their resolution.

Usually, DICOM datasets have 16-bit depth and they contain high-dynamic-range data. Consumer displays can only display 8-bit colour depth pixel data. Therefore, a conversion is required to transform the stored data into data that can be rendered and visualised.

A solution is to define a sub-range within the stored data range. In this manner, the contrast in the defined range is improved, while the data outside the defined sub-range is lost. However, by allowing the dynamic configuration of this sub-range, the user can explore the data without losing acquisition detail.

The defined sub-range is called *window* in the DICOM vocabulary. The *window* specifies a linear conversion from stored pixel values in the DICOM file to values to be displayed on the screen. The *window* is adjustable by the user using two levels: *window width* and *window center*. The *window width* selects the length of the data range to be converted and the *window center* locates the selected range (window) within the original data range.

Listing 7.3 shows the default linear conversion applied in the DICOM specification. x represents the input value, y is the output value within the range $[y_{min}, y_{max}]$. w_c is the *window center* and w_w is the *window width*.

Listing 7.3: Default linear conversion from stored pixel data to display data in DICOM.

```
**if** $(x <= w_c - 0.5 - (w_w - 1)/2)$, **then** $y = y_{min}$
**else if** $(x > w_c - 0.5 + (w_w - 1)/2)$, **then** $y = y_{max}$,
**else** $y = ((x - (w_c - 0.5))/(w_w - 1) + 0.5) * (y_{max} - y_{min}) + y_{min}$
```

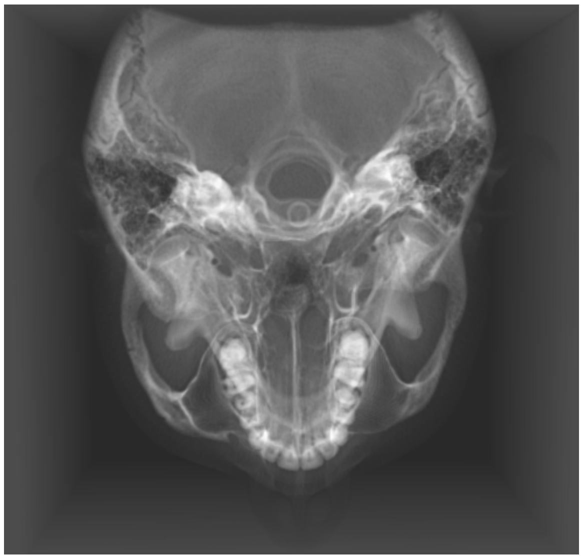
To change the *window*, dynamic sliders are provided as UI elements to the users. The user can manually discard or extend the desired *window* by modifying the *window width* and *window center*.

Figure 7.2 shows the *INCISIX* dataset from *OsiriX* library (OsiriX, 2018) rendered in the browser using the proposed approach with different window levels.

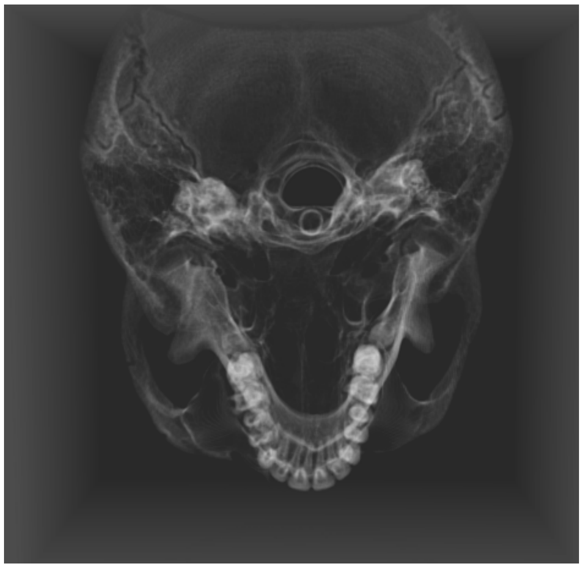
In Figure 7.2a the *window width* is larger than Figure 7.2b: more data values are being interpolated into the [0-255] (8-bit) range. The *window center* is positioned on the lower bound of the original data range. With that configuration, this *window* allows to visualise the skull and teeth data, while in Figure 7.2b the *window center* has been moved to the upper bound of the data range. As a result, only the teeth are visible because of their higher density value. The majority of the skull data is outside the *window* in Figure 7.2b.

7.2 Segmented medical data

Trained specialists in medical imaging need to visualise segmented volume datasets. The segmentation of the volume is important to focus the attention or to limit further work at the region of interest. When volume data is segmented, the voxels are labelled so that they can be processed independently.



(a) *window center* = 395 and *window width* = 2068.



(b) *window center* = 1056 and *window width* = 1489.

Figure 7.2: *INCISIX* dataset from the *OsiriX* repository OsiriX (2018) with different *window* levels.

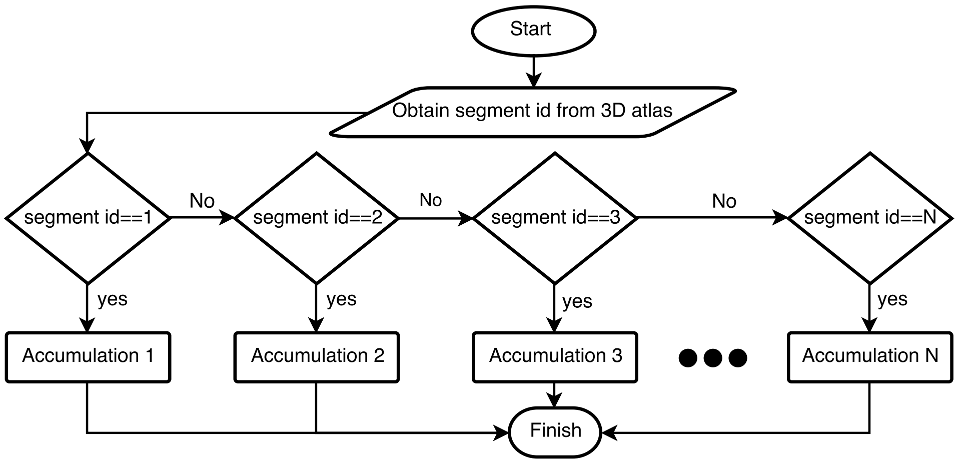


Figure 7.3: Flowchart describing the segmentation algorithm in the ray traversal

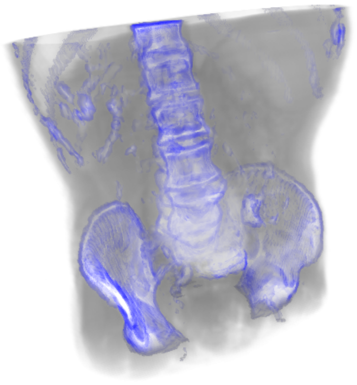
Any medical visualisation tool must be capable of helping the user to discern the region of interest from the whole volume. Volume visualisation must be extended to provide the tools that the medical use cases require.

The ray-casting algorithms presented in Chapter 5 and Chapter 6 can be extend to support the rendering of segmented data. In the ray-casting loop, the ray traverses the volume accumulating colour and opacity. The segmented data acts as an identifier on each voxel to allow the selection of an alternative accumulation algorithm (render style) at the region of interest. A distinctive colour and opacity in sections of the ray traversal can be applied selectively. This allows to stand out a region by visually enhancing the segmented area from the whole volume.

The segment data fetching is translated into a switch or selection statement that is performed in the ray-casting loop at per ray-step basis. Figure 7.3 illustrates this process. On each step of the ray, the voxel data is sampled. Then, the voxel identifier is sampled from the segmented data. A given rendering style will be applied in function of the obtained identifier.

From a performance point of view, this must be carefully considered, as branching is not well performed in GPU hardware. The type of variation in the accumulation process (render style) will change depending of the use case, e.g., mapping colours with a transfer function (TF), modifying the opacity given the camera view and discarding the data.

Figure 7.4 shows two renders of datasets with segmented regions. Each one has been generated with different accumulation processes to make them distinguishable from the volume.


(a) *EdgeEnhacementVolumeStyle*

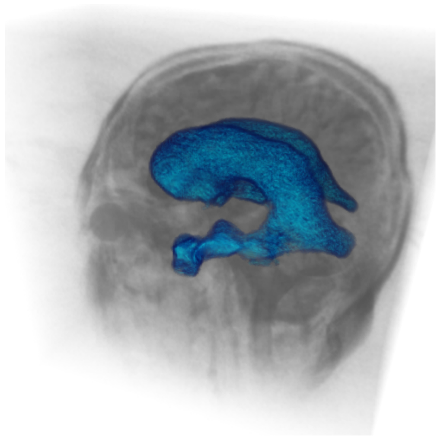

(b) *CartoonVolumeStyle*

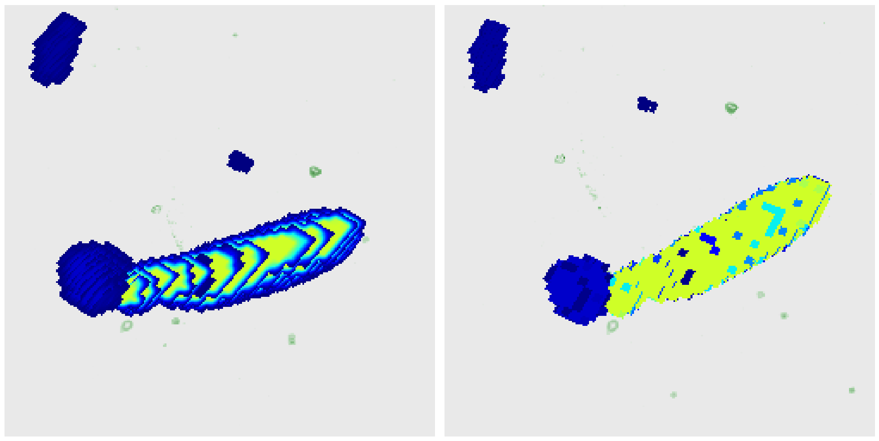
Figure 7.4: Segmented data renderings of the aorta ($512 \times 512 \times 97$) and the *Head MRI* ($256 \times 256 \times 124$) datasets. a) the edges of the segmented bones have been enhanced with colour. b) the segmented ventricles of the brain had been coloured with blue tones.

It is advisable to specify a parameter to explicitly know the maximum number of segments beforehand. The maximum number of segments limits the branching statements that will be added to the shader. These statements can be dynamically added at the creation time of the fragment shader. Otherwise, the number of branching statements will be unbounded and a big amount of unnecessary comparison operations will be performed with a potential impact on the performance.

The segmented data is tiled into a matrix configuration with the same procedure performed with the volume data (see Section 3.2.3 in Chapter 3). To improve performance the segmented data could be stored in the alpha channel of the original volume data texture atlas. However, it is better to create a separate texture, due to the following reasons:

i) The segmentation can be processed automatically or manually in an off-line tool and then be loaded on demand when required. Communication between server and client should be considered.

ii) Due to memory restrictions in mobile device GPUs compared to desktop GPUs, the segmented texture atlas can be stored with a smaller texture size than the original texture atlas. The same position coordinates would be valid to fetch the data from both textures. The texture atlas resolution should be considered. In some cases it is too big for real-



(a) Linear interpolation (b) Nearest interpolation

Figure 7.5: Difference between interpolated and nearest sampling in segmented texture data (zoomed area on the segmented blobs).

time rendering. The only downside of this method is a precision loss to discern the segmented regions due to downscaling.

An important consideration is the hardware base texture interpolation when sampling the texture containing the segmented *ImageTextureAtlas*. When the linear hardware interpolation is enabled and discrete values are used to label voxels, label identifiers are interpolated when sampling between voxels. This can lead to transition artifacts in between segmented regions. Figure 7.5 shows the difference between linear interpolation and nearest-interpolation when sampling for segmented identifiers. In this figure several segmented blobs with different colours are displayed. With nearest interpolation (see Figure 7.5b) voxels are clearly identified into a given segment, whereas with linear interpolation (see Figure 7.5a), voxels are not correctly identified leading to transition artefacts in the borders of the segmented blobs.

7.3 Inside exploration of volume data

Another visualisation requirement in the medical environment is related to the immersive experience that volumetric data visualisation can offer. This can be conceptualized as the ability to explore the data from within the volume. An inside exploration allows the user to easily discern the internal composition of the volume rather than looking at the cross-sectional 2D images. The exploration of the inside of the volume can be enabled by dynamically changing the initial position of the ray origin in the ray-casting algorithm presented in Chapter 6 at Section 6.1.

A unitary cube is used in the ray-casting as a proxy geometry. This cube is the volume data bounding box. The camera position is used as the ray origin when the camera is translated inside the cube, otherwise the front faces of the cube are used as the ray origin. Figure 7.6 describes the algorithm used to enable the inside exploration.

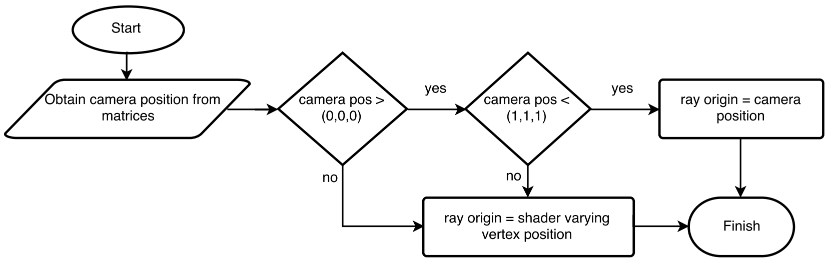

Start
Obtain camera position from matrices
camera pos > (0,0,0)
yes
no
camera pos < (1,1,1)
yes
no
ray origin = camera position
ray origin = shader varying vertex position
Finish

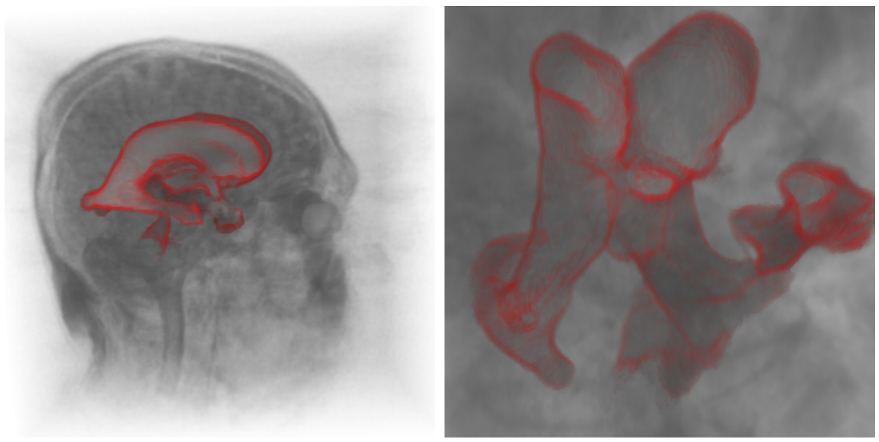
Figure 7.6: Flowchart describing the ray origin computation in the single-pass ray-casting algorithm.

In the rasterization process, the vertex coordinates are interpolated per fragment (`varying vertex` position). At each pixel in the front faces of the cube, the 3D texture coordinates of the ray-entry points are computed.

The camera position can be obtained in world space coordinates from the inverse `ModelView` matrix. The unitary cube is modelled so that it is located in the center of the scene in world coordinates.

Using the maximum and minimum boundaries of the cube, it can be determined whether or not the camera is inside the volume. If the camera is outside the cube, the ray origin is assigned to the interpolated vertex position at the front faces of the cube (computed output of the vertex shader, `varying vertex` position). If not, the ray origin is the camera position. Figure 7.7 shows an external and internal rendering of the *Head MRI* dataset (University of Tübingen WSI/GRIS, 2014).

Back face culling must be disabled when internal exploration is required. When the camera is moved inside the cube, the ray direction for each ray is obtained by subtracting the interpolated back face vertex position (ray-exit) from the camera position (ray-origin). With back-face culling enabled, the 3D texture coordinates in the back-face will not be rendered, since the cube will be clipped before rasterization.



(a) Camera outside the volume (b) Camera inside the volume

Figure 7.7: Rendering of the `Head MRI` dataset ($256 \times 256 \times 124$) enhancing a segmented area.

7.4 Interactive transfer function

In volume rendering, applying a transfer function (TF) is one of the most common techniques to illustrate a volumetric dataset. A TF is a lookup function that maps each scalar value of the dataset [0-255] with a given colour and opacity. Typically, a set of predefined and general TFs can be used (from red to green to blue, rainbow schema, etc.). It is very common to find domain-specific TF's (like in the weather radar information) that are widely accepted by the experts of that domain.

The web platform provides the tools to create interactive UI elements. Using the scalable vector graphics (SVG) technology, a transfer function editor can be built. With a TF editor, the user can interact with the volume rendering visualization, exploring and colourizing the desired data.

Figure 7.8 shows the first developed transfer function editor prototype under the 3D rendering context. The TF editor acts as an interactive chart. It has two axis: X represents the volume data values in the [0-255] range (8-bit), while Y axis represents the opacity from [0-1]. In this chart, an histogram of the volume data values is plotted in order to help the user infer in which value ranges the data is located. The user can add control points by clicking inside the plotting area. A colour is assigned to each control point. Defined colours will be interpolated between control points linearly (X axis). The opacity is also interpolated between control points in function of the point height (Y axis). As a result of the user interaction (modification of the control points) a Look-Up Table (LUT) is defined as a 255 pixel width 1D *RGBA* image. This image is directly passed as a texture input to the GPU and therefore, any changes in the TF editor have an immediate effect in the volume rendering visual output.

It is common to offer a predefined TF and give to the user the possibility to customize it interactively.

Web components are a set of web platform APIs that allow to create reusable encapsulated HTML tags to be used in web pages and applications. These web components use existing web standards. They are a good solution to integrate with the declarative approach of the X3DOM volume rendering component (see Chapter 4). Using the Polymer framework (Google, 2018) a reusable transfer editor web component has been developed and publicly shared (Arbelaiz, 2018).

The TF editor component can be inserted in any HTML5 web page using the `<tf-editor>` tag. This element can be configured defining attributes or properties on the DOM element. The number of bins in the histogram, the initial control points and editor size are customizable in a declarative manner. At any point the defined control points can be exported as JSON formatted string.

Listing 7.4 shows a X3DOM volume scene declaration of the `aorta` dataset along with the TF editor component declaration. It shows how to link the editor with the X3D scene using a CSS selector in the *x3domSelector* attribute of the `<tf-editor>` DOM element.

Figure 7.9 shows both the re-usable TF editor web component and the resultant 3D rendering of the `aorta` dataset. In this example, the user has set the a series of control points to illustrate the volume. The required code to built this visualisation is shown in Listing 7.4.

The work presented in this chapter builds a bridge between the real practical requirements of medical visualisation applications and the presented contributions on this thesis towards ubiquitous volume data visualization. The presented implementation approaches offer a solution that developers can use to build applications and web based visualizations in an easy way (see Listing 7.4).

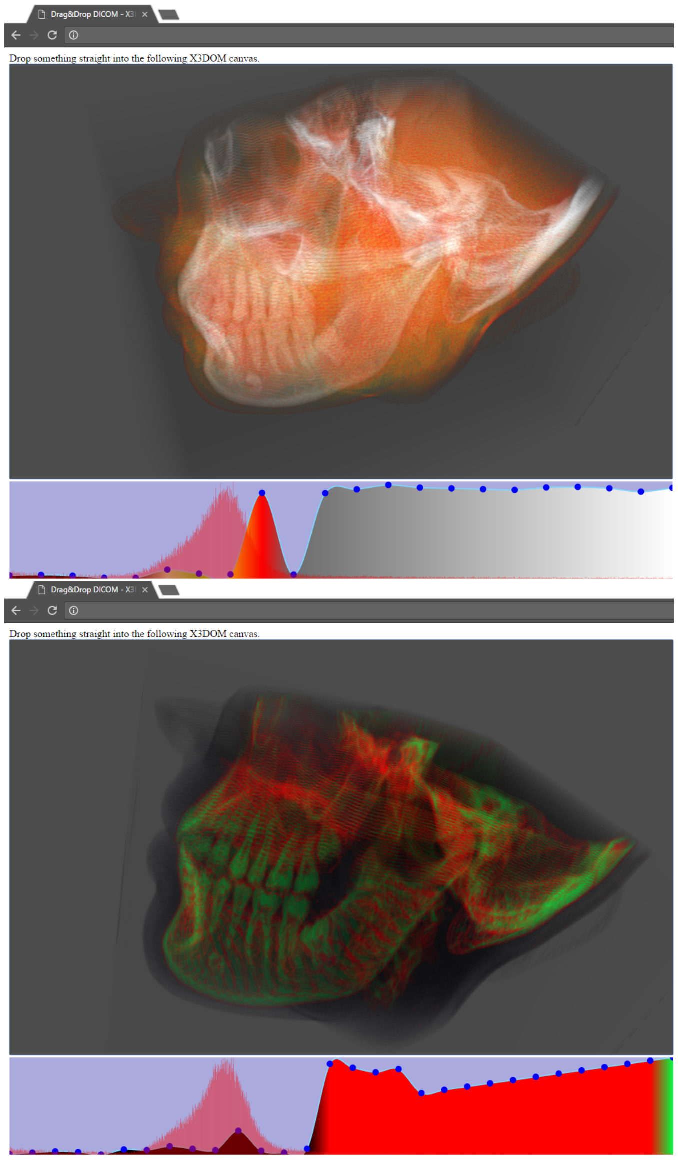

Drag&Drop DICOM - X3
Drop something straight into the following X3DOM canvas.
Drag&Drop DICOM - X3
Drop something straight into the following X3DOM canvas.

Figure 7.8: Volume rendering transfer function editor web application interface prototype.

Listing 7.4: X3DOM scene declaration with reusable TF editor linked by dynamic HTML5 canvas

```
<x3d>
  <scene>
    <background skycolor="1.0 1.0 1.0"></background>
    <viewpoint description="Default" znear="0.0001" zfar="100">
        </viewpoint>
    <transform>
      <volumedata id="volume2" dimensions="4.0 4.0 4.0">
        <imagetextureatlas containerfield="voxels" url="aorta.
            png" numberofslices="96" slicesoverx="10"
            slicesovery="10"></imagetextureatlas>
        <opacitymapvolumestyle lightfactor="1.2"
          opacityfactor="15.0">
          <imagetexture containerfield="transferFunction">
            <canvas width="255px" height="1px"></canvas>
          </imagetexture>
        </opacitymapvolumestyle>
      </volumedata>
    </transform>
  </scene>
</x3d>
<br>
<tf-editor name="TF" x3dom-selector="#volume2"></tf-editor>
```

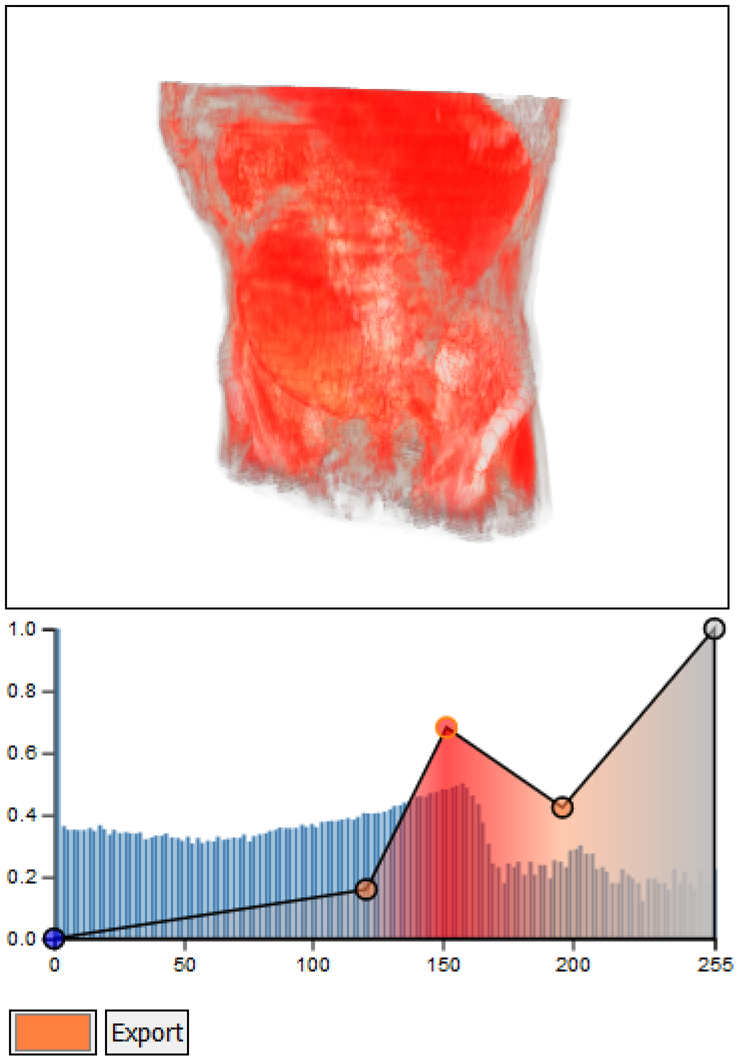


Figure 7.9: Web UI of the transfer function editor component. Control points have been positioned to colourize the volume data.

Chapter 8

Extensions for the ISO-IEC X3D standard

This chapter describes two new nodes and extensions that have been proposed to the X3D working group members and the Web3D community. These definitions are a consequence of the knowledge gathered in the research described in previous chapters. The proposal is leaded by the author of this thesis, but it required the cooperation of other researchers and feedback from the Web3D community. The chapter provides both the context and the proposal definition. It also shows the potential provided by the proposal.

With the adoption of WebGL in modern browsers, research and development in Web-based hardware-accelerated graphics have flourished. Agreements and conventions have been developed in order to sustain the exchange of 3D graphics. This includes volumetric content. However, new challenges create new needs that require new developments in existing standards or the creation of new ones. The interoperability between Web and non-Web applications, and to maintain a cross-device support are a must in the Web platform. Therefore this thesis has focused in the ISO-IEC Standard of X3D Web3DConsortium (2017b), which is a internationally ratified specification. This standard defines for web applications the interchange and delivery of declarative 3D graphics over the net.

The Web has become a medium to expose rich multimedia content to the general public. Using the Web as unified access point, volume rendering could become another tool for professionals and casual users alike. In order to reduce the gap between the expected capabilities of web-based tools and their desktop counterparts, requests and priorities have been analysed with the user community. The outcome of this research optimizes functionalities within the current limitations of the Web platform.

The evolution of 3D graphics in the Web is slow in comparison to traditional desktop solutions. This is mainly due to two factors: *i)* security: applications developed by third-party must be sand-boxed within the browser context and *ii)* wider cross-device support: they tend to reach to a wider range of devices and GPUs. Despite of this, the web ecosystem has benefitted from a great surge in 3D graphics content since the introduction of the WebGL API. This has provided an opportunity to create communities of both users and developers alike. They help with both the adoption and with the improvement of this technology into the future. One of the communities that has been supporting the exchange of 3D content over the net is the one behind the X3D (Extensible 3D) ISO (Web3DConsortium, 2017b).

This chapter is structured as follows: Section 8.1 introduces the X3D standard with some of its contributions in the scientific community along the X3D volume rendering component. Section 8.2 describes the proposal of new nodes not contemplated in the current volume rendering component. Section 8.3 proposes community-driven extensions to be included in the standard. Finally, Section 8.4 concludes with the state of the proposed changes towards the next iteration of the X3D volume rendering component.

8.1 X3D and X3D volume rendering component

The Extensible 3D (X3D) (Web3DConsortium, 2017b) is an ISO-IEC ratified standard to represent and communicate 3D computer graphics. It is maintained and developed by the Web3D consortium. X3D is composed by a rich set of extensible components targeting different computer graphics areas (e.g CAD, Geospatial, Humanoid-animation, NURBS, etc.).

X3D is actively being used in the scientific community. In order to extend its features, researchers publish modification proposals and enhancements to concrete components. For example, for the Geospatial Component (McCann et al., 2009) proposed enhancements to correct deficiencies with the visualization of data in a globally set context, to improve browser rendering for terrain data and to spread the adoption of the component.

X3D has been evolving with each iteration since its initial definition. Due to its component-based and profile-based architecture, new nodes can be added independently easily and in collaboration with its corresponding working group. The Web3D Consortium's Medical Working Group (MWG) (Web3DConsortium, 2017a) specifies and implements open standards to support cross-platform representation of medical visualisation from a wide variety of image modalities and medical data exchange capabilities.

Focused in the visualisation of volume data, the Medical Working Group (MWG) has created the X3D Volume Rendering Component (Web3DConsortium, 2017a) and the Medical X3D profile John et al. (2007). During the standardization process of the Volume Rendering Component, (Jung et al., 2008) presented a specialized endoscopic training simulation system based on an extended X3D. It showed one of the multiple use case cases of application for volume rendering. Polys et al. (2011b) evaluated the proposed X3D Volume Rendering Component for its suitability in the visualisation of several volume image data types from different scientific fields. Furthermore, Polys and Wood (2012) evaluated the volume component specification under several criteria: representation, implementation, interaction and interoperability/integration to validate X3D as a reproducible volume scene declaration interchangeable format.

Applications of X3D based volume rendering are widespread in the medicine field, from surgical planing to educational purposes. With the cost reduction and availability of modern stereo head mounted displays (HMD), a new frontier of application has been opened for X3D. Towards immersive VR environments, Behr et al. (2007) presented extensions to support different interactions and navigation tasks and Polys et al. (2013a) described the challenges and capabilities of X3D in an immersive volume rendering implementation.

Actually, X3D volume rendering component defines three abstract nodes, three data nodes and nine style nodes. Chapter 4 presents the web implementation of these definitions performed within this thesis. Currently, the X3DOM volume rendering component is the unique web-based existing implementation.

8.2 New node proposals

This section proposes new nodes that extend the functionality of the current medical profile of ISO/IEC X3D. First, a web-centered *ImageTextureAtlas* is proposed in Section 8.2.1. This node enables the interactive visualisation of volumetric data in Web based GPU accelerated volume rendering ray-casting algorithms. Finally, Section 8.2.2 presents the multi-planar reconstruction rendering style node.

8.2.1 ImageTextureAtlas — X3DTexture2DNode

In GPU-based volume rendering, 3D textures are used to store volume data. Thus, it is defined in the X3D ISO that volume data shall be declared using X3D's *Texturing3D* component. Unfortunately, the WebGL 1.0 API does not support this type of texture.

This limitation of WebGL 1.0 API has been overcomed with the method proposed by Congote et al. (2011). Using a 2D texture a 3D texture can be emulated by resampling the 2D texture and performing trilinear interpolation in the fragment shader at a pixel level.

Listing 8.1 shows the X3D definition of the proposed node: *ImageTextureAtlas*.

Listing 8.1: X3D definition for the *ImageTextureAtlas* node.

```
ImageTextureAtlas : X3DTexture2DNode
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFBool [in,out] repeatS TRUE
  SFBool [in,out] repeatT TRUE
  SFNode [in,out] textureProperties NULL [TextureProperties]
  MFString [in,out] url [] [URI]
  SFInt32 [in,out] numberOfSlices 0 [0,∞)
  SFInt32 [in,out] slicesOverX 0 [0,∞)
  SFInt32 [in,out] slicesOverY 0 [0,∞)
  SFBool [in,out] hideChildren TRUE
  SFString [in,out] channels "R"
  SFString [in,out] sortOrder "ROW" ["ROW", "COLUMN",
      "CHANNEL", "NONE"]
}
```

Volumetric data, specially those obtained from a MRI or a CT scan, can be seen as a set of 2D image slices in an array. The proposed *ImageTextureAtlas* node at Listing 8.1 allows the represent of the 3D volume data by composing all the 2D slices into a single 2D texture (Congote et al., 2011). Instead of adding a Z dimension to the texture, all 2D slices are arranged into one image with a matrix configuration. Figure 8.1 shows an atlas of slices for the proposed *ImageTextureAtlas* node.

The following attributes in Listing 8.1 are inherited by *X3DTexture2DNode*: *metadata*, *repeatS*, *repeatT*, *textureProperties* and *url*.

The *numberOfSlices* attribute indicates the number of slices or the dimension size in the Z axis direction. *SlicesOverX* attribute indicates the

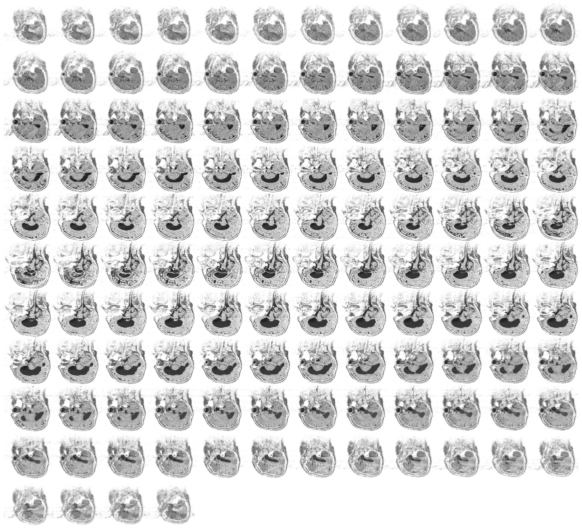


Figure 8.1: A 2D image representing an *ImageTextureAtlas* of the `Head MRI` dataset (colour inverted and contrast enhanced) as a set of 2D slices tiled into a matrix configuration.

number of slices along X axis or the number of columns in the matrix configuration, while the *SlicesOverY* indicates the number of slices along Y axis or the number of rows in the matrix configuration. These values must be provided by the user, since they cannot be deduced from the input image.

The amount of volume data that can be stored in a 2D *ImageTextureAtlas* is limited by the GPU's 2D texture size limit. However, some strategies can be followed to allow the visualisation of bigger datasets. Noguera and Jiménez (2012) used colour channels (Green, Blue and Alpha colour channels) of a volume data atlas to store larger datasets. For this purpose, Listing 8.1 shows the *channels* and *sortOrder* fields.

The *channels* attribute defines in which colour channel of the texture, volume data is being stored. The default behavior is to store the volume data in the Red colour channel by specifying the "R" value. When, a larger volume is required to be converted into an *ImageTextureAtlas* additional channels can be specified with "R", "G", "B", "A" characters. For instance, to store up to three times more data. A "RGBA" value in the *channel* attribute will indicate that all colour channels of the texture are being used to store the data. Once multiple colour channels are defined, the order in which 2D slices are tiled into the atlas must be set in the *sortOrder* attribute. The default behavior is to tile the 2D slices that represent the Z axis of the volume data in *"ROW"* order. For each slice of the *ImageTextureAtlas* the next slice in the Z axis is the contiguous slice in the row of the matrix of slices. When the *sortOrder* is set to *"CHANNEL"* the next slice of the atlas in the Z axis direction is stored in the contiguous colour channel.

Listing 8.2 definition represents how *X3DVolumeData* derived nodes, such as the *VolumeData*, should allow a 2D texture input to accept the new *ImageTextureAtlas* proposed node as a valid parameter.

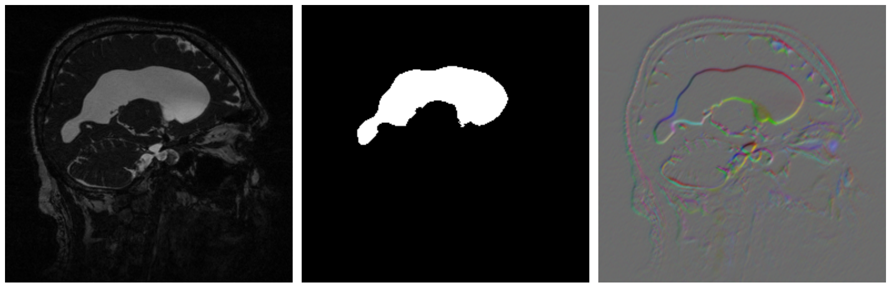
Listing 8.2: *ImageTextureAtlas* as an input to *X3DVolumeData* nodes. Example for the *VolumeData* node.

```
VolumeData : X3DVolumeDataNode {
 SFVec3f [in,out] dimensions 1 1 1 (0,∞)
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFNode [in,out] renderStyle NULL [X3DVolumeRenderStyleNode]
 SFNode [in,out] voxels NULL [X3DTexture2DNode,
    X3DTexture3DNode]
 SFVec3f [in,out] bboxCenter 0 0 0 (-∞,∞)
 SFVec3f [in,out] bboxSize -1 -1 -1 [0,∞) or -1 -1 -1
}
```

Arbelaiz et al. (2016b) have extended the *ImageTextureAtlas* approach to adapt its use to the other nodes described in the current X3D v3.3 ISO specification (Web3DConsortium, 2017a). The previous paragraphs explain how this node can store 3D texture data. The *ImageTextureAtlas* node can also be used to store gradient volume data, segmented volume data and to directly upload new data into the visualization.

The surface normals of the volume data are required to apply the volume rendering styles defined in the X3D ISO Volume Rendering Component (Web3DConsortium, 2017a). Voxel surface normals are stored as 3D texture. In this manner, illustrative and non-photorealistic styles can be applied to the volume visualization.

The surface normals are approximated with the gradient computation of the volume data. The gradient computation generates a three component vector for each voxel in the volume. Each component matches with the derivative of the volume data on each axis direction. Using the same approach as before, an *ImageTextureAtlas* is created using the colour channels of the 2D texture to store the vector information. The gradient vector is encoded for each pixel in the atlas in the RGB colour channels R: X, G: Y, B: Z.



(a) Volume data (b) Segmented data (c) Gradient data

Figure 8.2: Slice types of the `Head MRI` dataset (Volvis, 2017) to be composed into an *ImageTextureAtlas*: a) voxel data slice, b) segmented data slice and c) gradient data slice.

The proposed node in Listing 8.1 is valid for the surface normals input. For this use case, the *sortOrder* attribute should be "NONE" and the *channels* attribute "RGB". Listing 8.3 shows an example declaration of the gradient data.

Listing 8.3: An *ImageTextureAtlas* declaration for the surface normals data

```
<ImageTextureAtlas containerField="surfaceNormals" url="
   gradient.png" slicesOverX="8" slicesOverY="8">
</ImageTextureAtlas>
```

The *containerField* and *URL* attributes at Listing 8.3 are the modifications required for the *ImageTextureAtlas* declaration. The *containerField* attribute is used to target the volume data *voxels* field in a *X3DVolumeDataNode* or gradient data *surfaceNormals* attribute in a *X3DVolumeStyleNode*. This also requires, for the *X3DVolumeStyleNode* type nodes with a *surfaceNormals* attribute, the acceptance of a *X3DTexture2DNode* as an input argument like it is shown in Listing 8.2 for the *VolumeData* node.

The *SegmentedVolumeData* node allows the user to discern regions of the volume and apply different rendering styles to each one. The segmented regions must be labelled per voxel. The use of an *ImageTextureAtlas* is mandatory in order to make the *SegmentedVolumeData* compatible with WebGL 1.0.

Figure 8.2 shows the difference between a volume data slice (R colour channel), a segmented data slice (R colour channel) and the surface normals slice (RGB colour channels).

Textures provide a mechanism to upload data to the GPU and consequently, send directly to the screen. Listing 8.4 shows an *ImageTextureAtlas* node declaration with a canvas. Please note that the *hideChildren* attribute hides the atlas from the user, but makes it available to be modified with JavaScript.

Listing 8.4: *ImageTextureAtlas* declaration linked with the 2D HTML5 canvas API

```
<ImageTextureAtlas containerField="voxels" url="" slicesOverX=
   "8" slicesOverY="8" hideChildren="true">
  <canvas id="v" style="width:2048px; height:2048px;">
  </canvas>
</ImageTextureAtlas>
```

The 2D canvas API can be used in the web platform to create or modify images that can be copied to the GPU. This web feature provides a mechanism to render dynamic changes and, for instance, to perform the construction of the atlas in the browser. This approach has been used in Chapter 3 for 4D volume data and in Chapter 7 for DICOM volume data visualization.

With the WebGL 2.0 API, 3D textures are supported in modern browsers, allowing to make better use of current GPU memory capabilities. Nevertheless, this method is valid for both WebGL APIs (1.0 and 2.0) and it can also be combined with 3D textures to make use of less memory space. Still a lot of devices only support the 1.0 API and the new API adoption is slow (Bösch, 2019). The proposed *ImageTextureAltas* will still be necessary for an ubiquitous volume rendering deployment.

8.2.2 Multi-planar reconstruction (MPR)

The Multi-Planar Reconstruction (MPR) is a wide spread rendering technique for real-time slicing of the volume data. Essentially, it enables the user to define arbitrary planes through the data. The rendering algorithm resamples the volume data to reconstruct the volume into the desired plane. Usually, the following planes will be defined: Axial, Sagital, Coronal and Oblique.

Listing 8.5 shows the proposed X3D MPR rendering style node definition.

The *MPRVolumeStyle* node defines a *transferFunction* attribute with the same functionality as the one already defined in the *OpacityMapVolumeStyle*. It can be used to illustrate or filter regions in the reconstructed planes. In fact, this is necessary in many domains, for example, in physics simulation

Listing 8.5: X3D definition proposal for the MPRVolumeStyle node.

```
MPRVolumeStyle : X3DVolumeRenderStyleNode {
 SFBool [in,out] enabled TRUE
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFNode [in,out] transferFunction NULL [X3DTexture2DNode,
    X3DTexture3DNode]
 MFNode [in,out] planes NULL [MPRPlane]
}
```

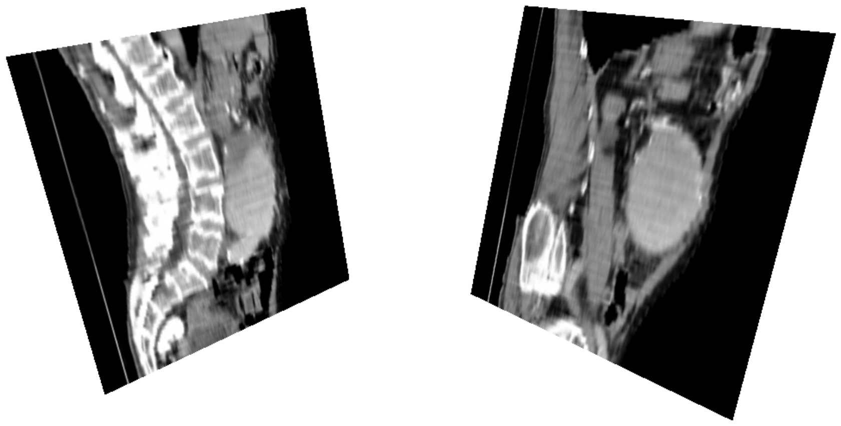


Figure 8.3: Plane reconstruction with *MPRVolumeStyle* on the `aorta` dataset.

it is used to correlate the visual output with the obtained results. Figure 8.3 shows an example prototype of the MPR style using the X3DOM framework.

This node also defines a *planes* attribute to allow the user to declare not only one, but multiple arbitrary planes. Listing 8.6 presents an arbitrary plane definition for the *MPRVolumeStyle* node.

Listing 8.6: X3D definition proposal for an arbitrary MPR volume plane

```
MPRPlane : X3DNode {
 SFBool [in,out] enabled TRUE
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFVec3 [in,out] normal 0 0 1
 SFFloat [in,out] pos 0.0 [0,1]
}
```

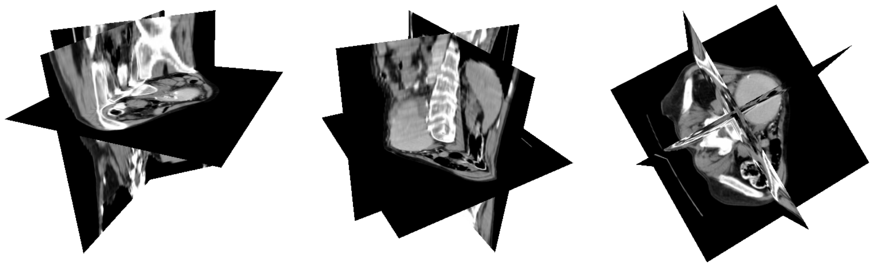


Figure 8.4: Multi-plane reconstruction with *MPRVolumeStyle* on the `aorta` dataset.

The *normal* attribute defines the normal vector of the plane, while the *pos* attribute establishes the position of the plane from the origin of the volume in the *normal* direction. Figure 8.4 shows a MPR prototype of the proposed node in X3DOM with multiple planes defined.

This technique is less memory expensive than a full volume rendering visualization: only the volume data in the neighbourhood of the defined plane is used in the computation. Less powerful GPU devices, like mobile devices, will handle easier this type of rendering.

8.3 Extension proposal to existing nodes

The X3D volume rendering component has been unchanged since the release of version v3.3 in 2013 (Web3DConsortium, 2017a). This section presents additions and enhancements which are focused in issues and problems received from web users in the X3D and X3DOM communities (X3DOM Community, 2017, 2015a, 2016b,a, 2015b). Proposals of how these enhancements could be added to the current X3D volume rendering component specification are presented in this section.

8.3.1 Transfer function edition

A transfer function (TF) is the most used method to add colour to the volume visualization. It allows to filter and enhance intensity ranges in the volume data by mapping each volume scalar value to a given colour and opacity. This enables the visualisation through some layers of specific densities and the increment of the opacity in other layers. For this purpose, usually a texture is used as a look-up table. However, a connection to the 2D texture that directly influences the TF is required in order to allow the creation of tools such as a native HTML5 transfer function editor presented in Chapter 7.

As stated in previous Section 8.2.1, Listing 8.4 shows the use of *hideChildren* attribute to attach a 2D canvas in order to update the volume data in the *ImageTextureAltas* node. The same approach can be used with the transfer function attribute of the *OpacityMapVolumeStyle* node. But, this time, to modify dynamically the TF within a 2D canvas (see Listing 8.7).

Listing 8.7: *ImageTexture* declaration linked with the 2D HTML5 canvas API

```
...
<OpacityMapVolumeStyle lightFactor="1.2" opacityFactor="15.0">
  <ImageTexture containerField="transferFunction" hideChildren
      ="true">
  <canvas id="tf" style="width:255px; height:1px;">
  </canvas>
  </ImageTexture>
</OpacityMapVolumeStyle>
...
```

Current X3D specification contemplates both 3D and 2D textures as valid input fields for the *OpacityMapVolumeStyle* node. The default behaviour is the use of 2D TFs (linear look-up tables). With the proposed addition the Web platform can support natively 2D textures that can be changed dynamically.

8.3.2 *quality* attribute

The X3D ISO specification is an abstract declaration unaware of the underlying implementations. A Web-based ubiquitous volume rendering implementation allows a volumetric scene to be deployed in a wide range of devices. A scene with multiple rendering styles may be plausible for a desktop PC with a dedicated GPU, but it could be too computationally expensive for a mobile device; not all devices have the same GPU features and computational power. Implementation-aware requisites should be considered in order to deploy X3D scenes in as many devices as possible. Otherwise, this situation can make volume rendering unavailable to some devices.

In order to allow one X3D volume rendering scene to be deployed into multiple devices, a quality control mechanism is proposed. This mechanism should focus on the target devices and it should control the amount of computation to be performed by the device via the underlaying implementation.

A *quality* attribute could be defined with a qualitative value (provided as a profile) or with a quantitative value (provided as ranged numerical scalar value). Thus, two proposals are presented.

Listing 8.8 shows an additional qualitative field for the *X3DVolumeDataNode*, where the *quality* attribute accepts three levels of quality *"LOW"*, *"MEDIUM"*, *"HIGH"*.

Listing 8.8: X3DVolumeDataNode definition with output quality control

```
X3DVolumeDataNode : X3DChildNode, X3DBoundedObject
 SFVec3f [in,out] dimensions 1 1 1 (0,∞)
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFVec3f [in,out] bboxCenter 0 0 0 (-∞,∞)
 SFVec3f [in,out] bboxSize -1 -1 -1 [0,∞) or -1 -1 -1
 SFString [in,out] quality "HIGH" ["LOW", "MEDIUM", "HIGH"]
}
```

The alternative is shown at Listing 8.9, where the *quality* field shows a quantitative value. For implementations where the quality of the output cannot be controlled, always a *"HIGH"* value or a *"1.0"* factor shall be expected.

Listing 8.9: X3DVolumeDataNode definition with output quality control

```
X3DVolumeDataNode : X3DChildNode, X3DBoundedObject
 SFVec3f [in,out] dimensions 1 1 1 (0,∞)
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFVec3f [in,out] bboxCenter 0 0 0 (-∞,∞)
 SFVec3f [in,out] bboxSize -1 -1 -1 [0,∞) or -1 -1 -1
 SFFloat [in,out] quality 1.0 [0,1]
}
```

In this manner, each possible implementation of the X3D standard could adequate the amount of computation to be performed at different levels. As an example, Figure 8.5 shows the same scene from the same perspective, under different rendering quality values. For real-time rendering the quality should be able to be changed to accommodate to the targeted GPU capabilities.

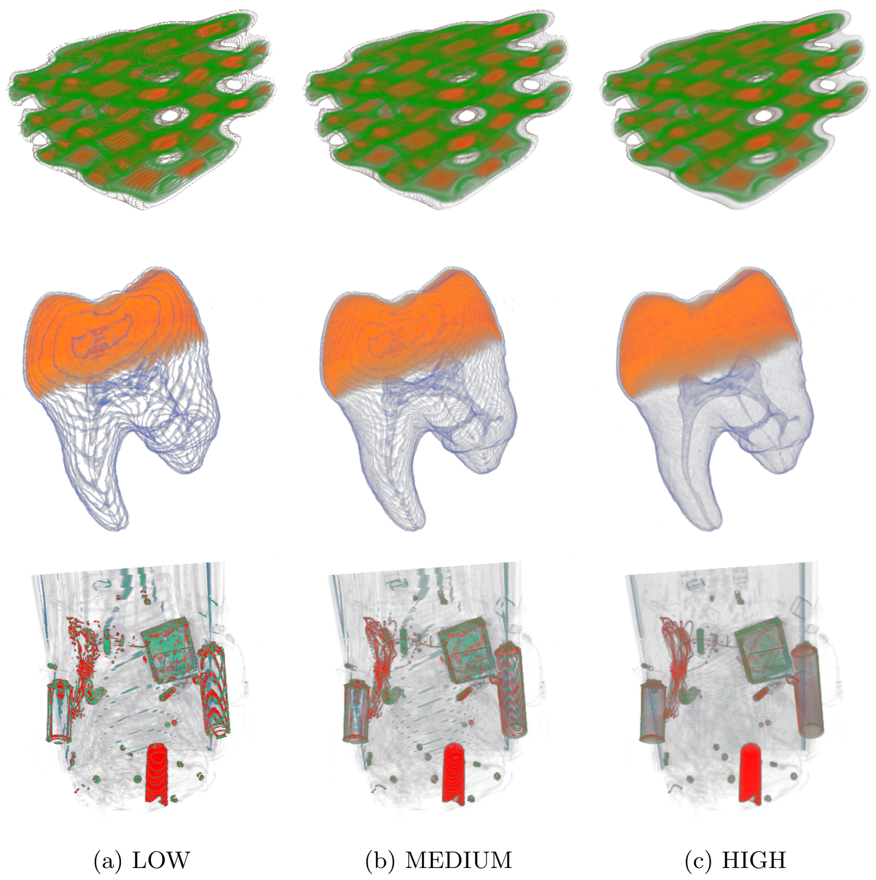

(a) LOW (b) MEDIUM (c) HIGH

Figure 8.5: Rendering of the `silicium`, `tooth` and `backpack` datasets at different rendering qualities.

8.3.3 *allowViewPointInside* attribute

Some regions of the volume data can be occluded even after filtering the data with a transfer function (TF). In those cases, the ability to explore the inside of the volume becomes necessary (X3DOM Community, 2015a). This feature has been already integrated in X3DOM (Arbelaiz et al., 2017a), but not in the standard itself. The current X3D ISO does not define the behaviour of the volume rendering algorithms in relation to the location of the camera inside the volume.

Allowing to place the viewer's virtual camera inside the volume provides a new perspective to analyse the data (see Figure 8.6).

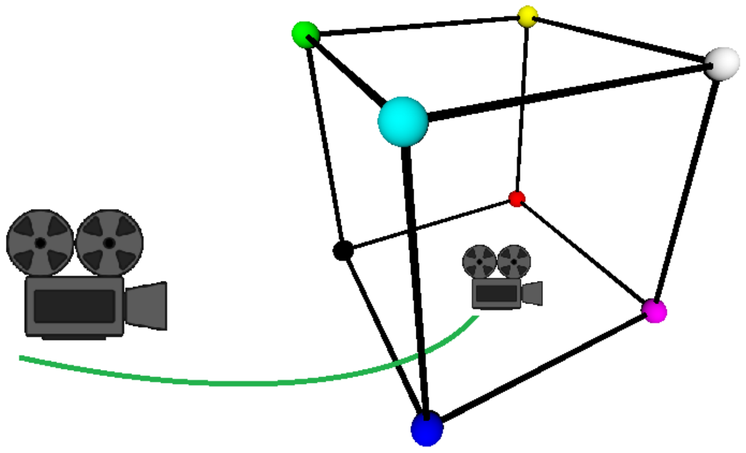


Figure 8.6: The visualisation from the inside of the dataset requires that the user moves the virtual camera location into the cube that holds the volumetric dataset. *Zoom* functionality can be triggered with the wheel mouse or equivalent mechanism in devices with touch screen.

Listing 8.10 shows the definition of the *X3DVolumeDataNode* with the proposed *allowViewPointInside* attribute.

Listing 8.10: *X3DVolumeDataNode* definition with the proposed *allowViewPointInside* attribute.

```
X3DVolumeDataNode : X3DChildNode, X3DBoundedObject
 SFVec3f [in,out] dimensions 1 1 1 (0,∞)
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFVec3f [in,out] bboxCenter 0 0 0 (-∞,∞)
 SFVec3f [in,out] bboxSize -1 -1 -1 [0,∞) or -1 -1 -1
 SFBool [in,out] allowViewPointInside TRUE
}
```

In this proposal a new boolean attribute *allowViewPointInside* is added to all nodes inherited by the *X3DVolumeDataNode*. In one hand, all X3D conformance implementations will follow the same behaviour. In the other hand, to explicitly enable or disable this functionality will allow to avoid the extra computation required.

Figure 8.7 shows an external and internal rendering of the *aorta* dataset.

For the *VolumeData* node the default definition will be the following:

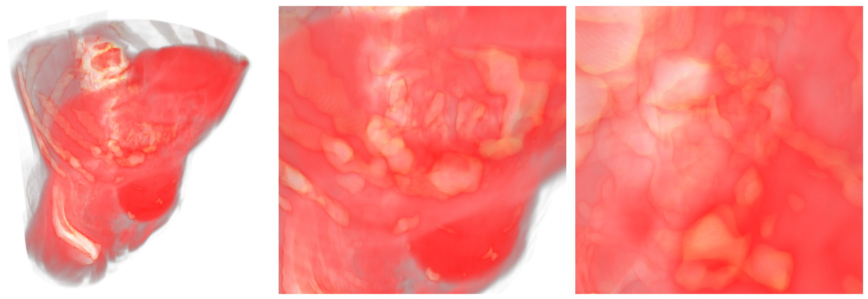

(a) Camera outside (b) Camera in BB (c) Camera inside

Figure 8.7: Moving the camera forward (in the view direction) inside the *aorta* dataset (512 × 512 × 97).

```
<VolumeData allowViewpointInside="true"></VolumeData>
```

By default it should be enabled the inspection of the volume data and if the user does not require to do so, it can be explicitly disabled.

8.3.4 ***sceneDepth*** **attribute**

A hybrid rendering of 3D polygonal meshes in conjunction with a volume object can be of great interest. It opens new use cases for several scientific fields. As a reference, Yang et al. (2015) has already presented a GIS use case for the visualisation of volumetric weather radar data and a polygonal terrain with X3DOM. The current X3D volume rendering component does not specify nor describe any polygonal and volume data intersection behavior. From a technical point of view, as presented in Chapter 5, it is already feasible to perform such hybrid rendering in a Web context (Arbelaiz et al., 2016a).

To add the scene depth information, nodes which inherit from *X3DVolumeDataNode* (*VolumeData*, *SegmentedVolumeData*, *IsoSurfaceVolumeData*) should allow to access the depth information of the rendered scene. Listing 8.11 shows the proposed *sceneDepth* attribute for the basic *VolumeData* node.

With the depth information of the previously rendered meshes in the scene. The coexistence of surface data and volume data together is possible. In this way, before rendering the volumetric data, the volume rendering algorithms can compute any intersection with polygonal surfaces and avoid any occluded computation while also performing the colour blending with the polygonal surface (see Section 5.3.3 in Chapter 5).

Listing 8.11: *VolumeData* definition with access to the scene depth information.

```
VolumeData : X3DVolumeDataNode {
 SFVec3f [in,out] dimensions 1 1 1 (0,∞)
 SFNode [in,out] metadata NULL [X3DMetadataObject]
 SFNode [in,out] renderStyle NULL [X3DVolumeRenderStyleNode]
 SFNode [in,out] voxels NULL [X3DTexture3DNode]
 SFNode [in,out] sceneDepth NULL [X3DTexture2DNode]
 SFVec3f [in,out] bboxCenter 0 0 0 (-∞,∞)
 SFVec3f [in,out] bboxSize -1 -1 -1 [0,∞) or -1 -1 -1
}
```

8.3.5 *cap* attribute

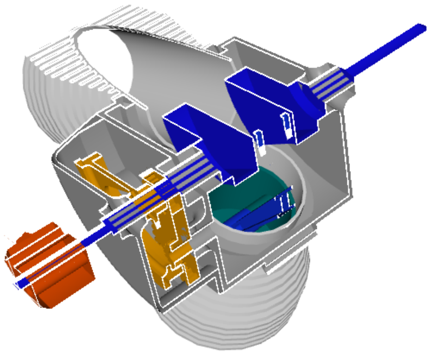
In the X3D specification the behaviour of clipping is directed towards surface data. This concept was introduced in X3D since the release of v3.2 (Web-3DConsortium, 2017b). It is defined as a plane that divides an space in two sub-spaces. The affected geometry in the outer-space, defined as being outside the plane, is removed from the rendered image as a result of applying the operation. Listing 8.12 shows the current node definition for the clipping plane.

Listing 8.12: Actual *ClipPlane* definition in X3D v3.3

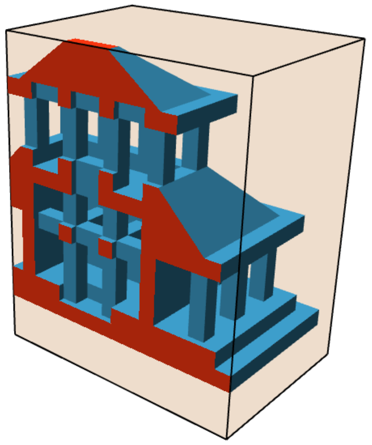
```
ClipPlane : X3DChildNode {
  SFBool [in,out] enabled TRUE
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFVec4f [in,out] plane 0 1 0 0 [0,1]
}
```

The *X3DVolumeData* derived nodes emulate a *Shape* node that handle volumetric data instead of geometry data. The clipping planes can be defined at any level of the transform hierarchy. This implies that the current specification of clipping planes should also be applicable to volume nodes.

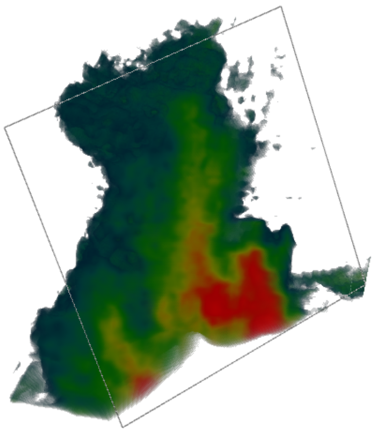
In a complex scenario, where both geometry and volume data are clipped a different behaviour is expected. Clipped volume data by definition will always contain data inside the clipped region (see Figure 8.8c). However, a clipped polygonal object could be empty. Figure 8.8a shows the experimental support of clipping planes in X3DOM, while Figure 8.8b show a capped



(a) Non-capped (surface)



(b) Capped (surface)



(c) Capped (volume)

Figure 8.8: Examples of the clipping behaviour in surface and volume data: a) clipping of a surface mesh, b) capped clipping of a surface mesh and c) clipping in volume data.

example from Mischek (2018).

Listing 8.13 proposes a *cap* attribute to mix volume clipped data and surface data together in the same scene.

A medical application using clipping planes will probably expect intersected 3D geometry to be capped (X3DOM Community, 2017). The specification should consider the possibility to cap clipped polygonal surfaces. Otherwise, this behaviour would remain undefined and therefore, any X3D implementation would provide a custom and non-standard implementation.

Listing 8.13: *ClipPlane* definition with *cap* attribute

```
ClipPlane : X3DChildNode {
  SFBool [in,out] enabled TRUE
  SFNode [in,out] metadata NULL [X3DMetadataObject]
  SFVec4f [in,out] plane 0 1 0 0 [0,1]
  SFBool [in,out] cap FALSE
}
```

For instance, the experimental clipping planes implementation in X3DOM has already defined additional attributes for the capping colour and strength. These are not included in the standard neither. An addition of a *cap* attribute to the *ClipPlane* should be enough to solve the hybrid rendering scenario.

8.4 Standardization

The ISO-IEC Standard Extensible 3D (X3D) version v3.3 specifies the integration and visual styling of volumetric data for real-time interaction. The specification is an important milestone describing a framework for expressive volume rendering scene presentation. However, it was written before the emergence of WebGL and the HTML5 platform. This chapter has described several extensions to adapt the X3D Volume rendering nodes to the Web platform and to enhance its functionality based on feedback provided by the X3D and X3DOM open source communities.

Two new nodes have been presented: *ImageTextureAtlas* and *MPRVolumeStyle*. The *ImageTextureAtlas* node can be used in the Web platform for real-time volume visualization. The Multi Planar Reconstruction (MPR) style rendering offers a lacking functionality in the current X3D medical profile. New extensions have been proposed: *i*) the edition of transfer functions, *ii*) intersection of the volume with 3D objects, *iii*) clipping planes with volume data and *iv*) control in the quality of the generated volume visualization. These proposals benefit X3D users by providing more advanced use cases. These extensions have been presented in the X3D Medical Working Group and they are being considered for future revisions of the ISO/IEC X3D volume rendering component.

Chapter 9

Results and discussion

This chapter presents the validation of the contributions of this thesis by showing applications and results in different domains under a variety of use cases. Developed applications and prototypes demonstrate how web based ubiquitous volume visualisation applications can be built upon the contributions presented in previous chapters.

This chapter shows how the nodes proposed in Chapters 4 and 8, and the algorithms implemented within them, provide a powerful tool for content developers. The visualizations described in this chapter are examples built with low coding effort and the performance of a selection of them are presented.

Furthermore, the potential of the contributions of this thesis are not limited to these examples. Based in the work developed on this thesis, other authors have further validated the presented contributions. Any content developer or researcher may tailor the components to solve other interactive volume visualisation problems. Examples by other authors of web applications and research work built with the volume render component presented in this thesis are also presented in this chapter.

The scene tree declaration to compose some of the generated renderings are shown in Section 9.1. Section 4.1.1 in Chapter 4 already presents a scene tree and its corresponding X3D declaration. Since in that chapter it is shown how little work is required by content developers to create a volume rendering scene, only a few lines of X3D/HTML of code are required. In Section 9.1 only the scene trees will be presented.

The chapter is structured as follows: Section 9.1 demonstrates the flexibility of the volume rendering component to create visualizations that can be applied in multiple domains. Section 9.2 shows an application for the quality inspection of plastic mould injected parts. Section 9.3 presents a 4D volumetric air-flow simulation as an engineering use case. Section 9.4 shows an application for the volumetric visualisation of DICOM data. Section 9.5 describes the prototyping of a web based virtual reality application. Finally, Section 9.6 presents applications and research work from other authors in the Web3d community, supported by these contributions.

9.1 Declarative multi-domain use cases

This section shows the advantages, flexibility and utility of the declarative approach for volume rendering. It demonstrates the powerful and easy-to-use methodology that offers the X3DOM volume rendering component for web content developers (see Chapter 4).

Four volumetric datasets from different thematic areas are presented: medical, engineering, physics and life sciences. For each use-case, some interaction examples are introduced and some possible solutions are devised by providing some X3D scenes that experienced users of the domain might use to explore the volumetric datasets.

Each use case is structured as follows: first, the objective and motivation for the visualisation is introduced, then a basic render of the volume is shown. Afterwards, a partial X3D scene for each use case is presented. Subsection 9.1.5 shows a table that resumes the performance achieved on each of the presented use cases.

9.1.1 Medical use case

Undoubtedly, a useful tool in the medical field is the segmentation, i.e. the partitioning of the volume data into different segments. The requirements and solution to this use case have been described in Chapter 7.

Sometimes, for a variety of reasons, a region of interest inside the volume needs to be enhanced or highlighted. The goal of this use case is to visualise segments which correspond to different organs, pathologies, tissue types and other biological structures. The user will differentiate better the segments from the rest of the data. This section uses the `Head MRI` (University of Tübingen WSI/GRIS, 2014) dataset. It consists of a Magnetic Resonance Imaging (MRI) scan of the head and a segmentation of the ventricles of the brain. With the use of the *SegmentedVolumeData* node, the ventricles shape are enhanced from the rest of the volume data. Figure 9.1 shows a basic visualisation of the *Head MRI* dataset, without the use of the segmentation data.

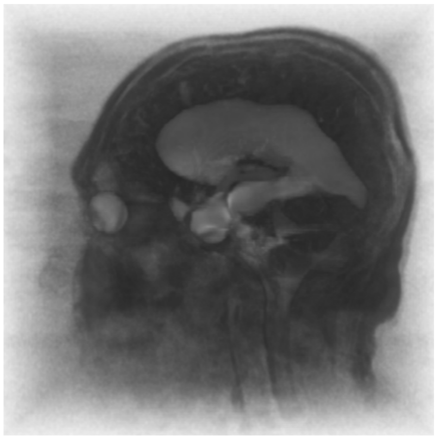


Figure 9.1: Direct volume rendering visualisation of the `Head MRI` dataset (2048×2048 atlas), using the *OpacityMapVolumeStyle* without a transfer function.

Using the segments information, a different rendering style can be applied to each segment. Figure 9.3 shows two rendering outputs that highlight the ventricles of the brain. Both use the *SegmentedVolumeData* node with two different rendering styles. Figure 9.2 is a partial X3D scene tree of Figure 9.3a. First, the volume is declared as a *SegmentedVolumeData*. In this case, two atlases must be provided to the component: the volume data atlas (*ImageTextureAtlas*) and the atlas containing the segmented information. Then, the rendering styles are declared. The first rendering style is applied to the first segment and the second rendering style on the second segment.

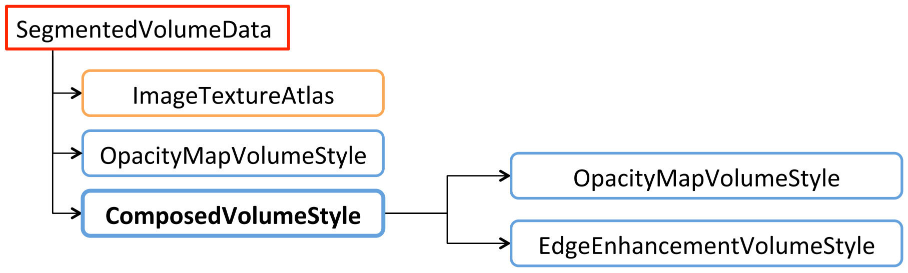
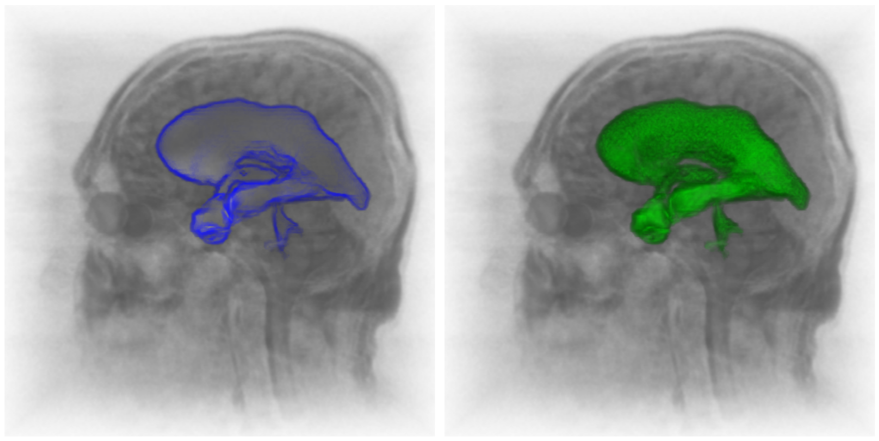


Figure 9.2: Partial X3D scene tree of the `Head MRI` dataset, using the *SegmentedVolumeData* node.

In the scene tree shown in Figure 9.2, the first segment has been rendered using the *OpacityMapVolumeStyle* with a low *opacityFactor* resulting more visible the insides of the head. The second segment, the ventricles of the



(a) *EdgeEnhancementVolumeStyle* (b) *CartoonVolumeStyle*

Figure 9.3: Direct volume rendering visualisation of the `Head MRI` dataset (2048× 2048 atlas), using the *SegmentedVolumeData* to enhance the ventricles of the brain

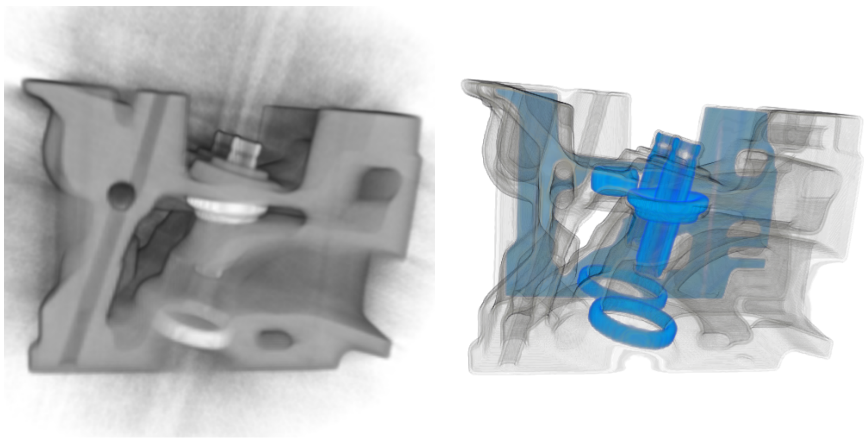
brain, has been rendered with a composition of two rendering styles: the *OpacityMapVolumeStyle* and the *EdgeEnhancementVolumeStyle*. Making the segmented region more noticeable and, at the same time, highlighting the shape of the ventricles. The difference between Figure 9.3a and Figure 9.3b is the render style used on the second segment. In Figure 9.3b the *CartoonVolumeStyle* is used instead of the *EdgeEnhacementVolumeStyle*.

9.1.2 Engineering use case

For the engineering field the `engine` (University of Tübingen WSI/GRIS, 2014) dataset has been selected. This dataset consists of a Computed Tomography (CT) scan of an engine block. The objective of this use case is to visually enhance a region of interest: the two cylinders inside the engine. This objective is achieved using two approaches with different rendering styles: firstly, with the aid of a transfer function, and secondly, with the visualisation of an isosurface. Figure 9.4a shows the `engine` dataset with the default rendering style: the *OpacityMapVolumeStyle*.

Using a 1D *transfer function* on the *OpacityMapVolumeStyle*, each value from the volume data can be mapped to a colour and opacity, enhancing and illustrating the volume (see Figure 9.4b and Figure 9.5).

The creation of the transfer function is out of the scope of the volume rendering component at the X3D level. However, a possible web based trans-



(a) Engine dataset (b) Engine with TF

Figure 9.4: Direct volume rendering visualisation of the `engine` dataset with a 2048×2048 atlas. a) Basic visualization (by default *OpacityMapVolumeStyle*). b) Engine dataset with a 1D transfer function

fer function editor has been presented in Chapter 7. The use of a transfer function offers a lot of control on how the volume is illustrated, allowing to enhance the desired information. Usually, it is a manual and time-consuming process. In this example, an alternative is to use the *IsoSurfaceVolumeData* and automatically extract the region of interest by selecting a correct set of *surfaceValues* (see Figure 9.6 and Figure 9.7).

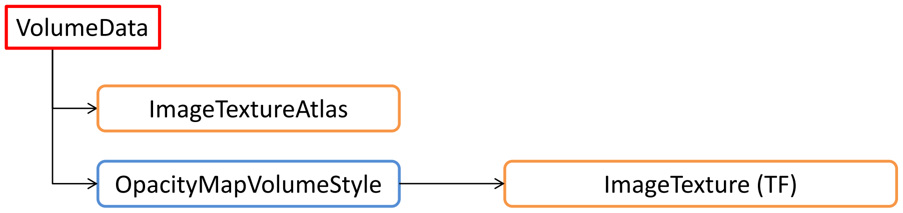
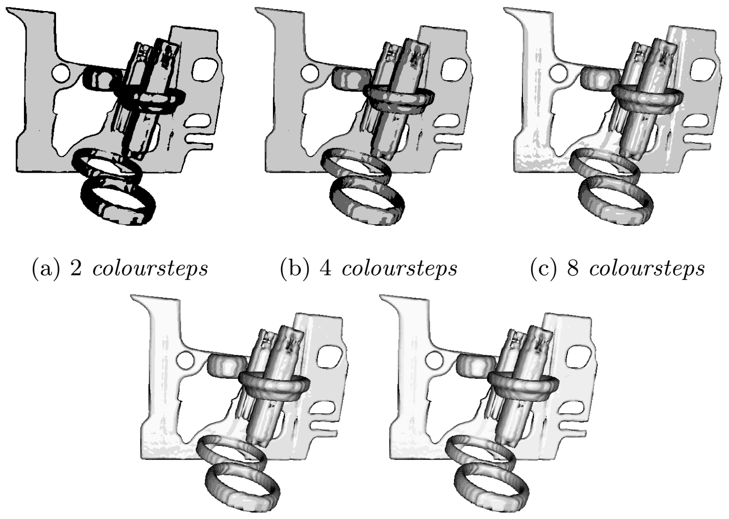


Figure 9.5: Partial X3D scene tree declaration of the `engine` dataset, using the *OpacityMapVolumeStyle* with a *transfer function*.

In Figure 9.6 the *CartoonVolumeStyle* has been used to illustrate the extracted isosurfaces, showing a comparison between different *coloursteps*. A more *cartoonish* effect can be achieved when the number of *coloursteps* is low (Figure 9.6a and 9.6b), whereas a higher value of *coloursteps* can be used to get a more solid appearance (see Figure 9.6e). Please note that in this case, the whole volume can not be seen as before (Figure 9.4b). Both



(a) 2 *coloursteps* (b) 4 *coloursteps* (c) 8 *coloursteps*

(d) 16 *coloursteps* (e) 32 *coloursteps*

Figure 9.6: Rendering of the `engine` dataset with a 2048 × 2048 atlas. Dataset declared as an *IsoSurfaceVolumeData* with a set of *surfaceValues* of [0.7, 0.75, 0.8] and illustrated with the *CartoonVolumeStyle* at different *coloursteps*.

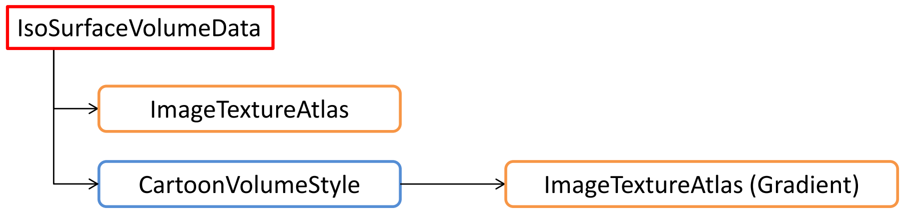


Figure 9.7: Partial X3D scene tree of the `engine` dataset, declaring the volume as a *IsoSurfaceVolumeData* and using a *CartoonVolumeStyle*.

presented solutions are able to visualise the cylinders, which denotes the flexibility of the proposed component and the X3D specification.

9.1.3 Physics use case

Volume rendering is useful for scientific volume data visualization. The `neghip` and `hydrogen-atom` (University of Tübingen WSI/GRIS, 2014) datasets have been selected to showcase the utility of the volume rendering component in the nuclear physics field. The `neghip` dataset (64 × 64 × 64) is a simulation of the spatial probability distribution of electrons in a high potential protein molecule. Knowing the distribution of the electron in such molecules has important benefits for chemistry-based areas, such as pharmacology, to better understand the relation between molecules and the organism (Linsen et al., 2012).

The `hydrogen atom` dataset ($128 \times 128 \times 128$) is a simulation of the spatial probability distribution of the electron in a hydrogen atom residing in a strong magnetic field. Both cases show the shape of the density distributions with the visualization of a selection of iso-values from the datasets. A basic visualisation of both datasets (Figure 9.8) gives a general idea of the shape, but does not help to accurately pinpoint the location of the electrons, nor the evolution of the distribution fields.

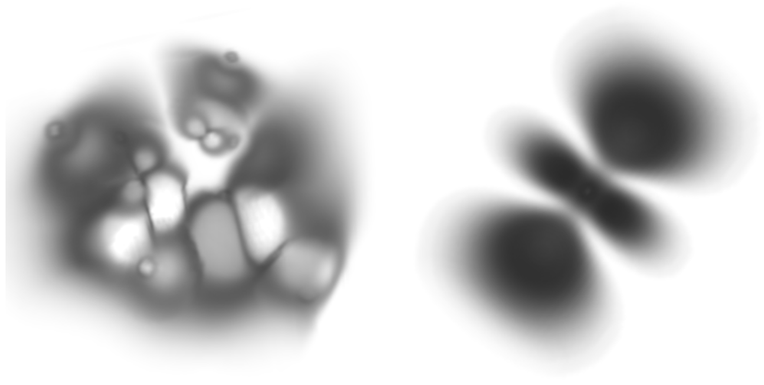

(a) Neghip dataset (b) Hydrogen atom dataset

Figure 9.8: Direct volume rendering visualisation of the `neghip` (512×512 atlas) and `hydrogen atom` (1024×1024 atlas) datasets with no styles applied.

To explore the distribution, an example of the visualisation of a set of isovalues and its illustration is presented (see Figure 9.10). The use of the *IsoSurfaceVolumeData* allows to apply a rendering style on each specified *surfaceValue*. The illustration of the volume is necessary to enhance the perception of depth in the generated output.

Figure 9.9 and Figure 9.12 are defined with the same tree. In both cases, an isosurface of the volume is being visualised with the *IsoSurfaceVolumeData* node and then, illustrated with the *ToneMappedVolumeStyle*. The desired isosurface value can be selected by specifying a *surfaceValue*, and it limits the surface detection with the *surfaceTolerance* parameter (Figure 9.11). The *PointLight* node is declared in both scenes, because it is necessary for the *ToneMappedVolumeStyle* to know the light location or direction.

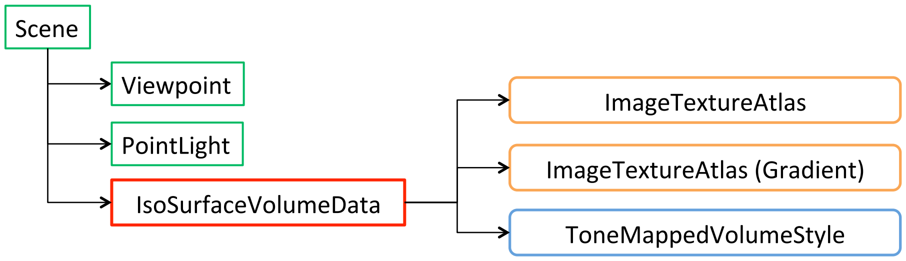
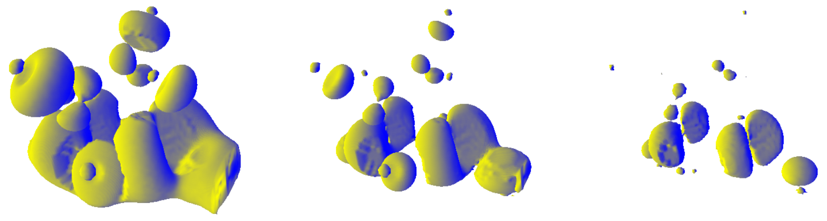


Figure 9.9: Partial X3D scene tree for the `neghip` dataset illustrated with the *ToneMappedVolumeStyle*.



(a) *surfaceValue* 0.2 (b) *surfaceValue* 0.4 (c) *surfaceValue* 0.9

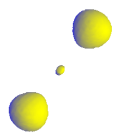
Figure 9.10: Volume rendering visualisation of the `neghip` dataset (512 × 512 atlas). Using the *IsoSurfaceVolumeData* node with a single isovalue on the *surfaceValues* parameter and illustrated with the *ToneMappedVolumeStyle*.

9.1.4 Life science use case

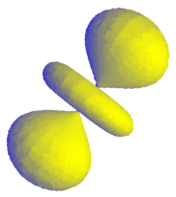
Educational articles are usually illustrated with hand-made figures or illustrative images making easier to understand their material. Volume rendering is adequate for the exploration of real data. By allowing to explore the data, a better understanding of its composition and morphology is obtained. Thus, the presented volume rendering component can be used to complement web articles and teaching material. As an example, to visualise its inner structure, the `orange` [1] dataset (256 × 256 × 64) has been selected. Figure 9.13a shows a basic rendering of the `orange` dataset. This can be easily declared in a few lines of HTML and X3D.

Figure 9.13b shows a cartoon rendering of the `orange` illustrated with orange and yellow colours to make it closer in appearance to the real fruit. As an alternative, a transfer function could be used to achieve a similar

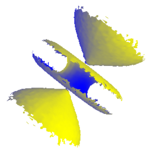
[1] Available at *http://www9.informatik.uni-erlangen.de/External/vollib/*



(a) *surfaceValue* 0.20



(b) *surfaceValue* 0.05



(c) *surfaceValue* 0.05 and *surfaceTolerance* 0.035

Figure 9.11: Volume rendering visualisation of the `hydrogen atom` dataset (1024 × 1024 atlas) illustrated with the *ToneMappedVolumeStyle*. a,b) Using the *IsoSurfaceVolumeData* node with a single isovalue on the *surfaceValues* parameter and a *surfaceTolerance* value of 0. c) Using the *IsoSurfaceVolumeData* node with a single isovalue on the *surfaceValues* parameter and a *surfaceTolerance* value of 0.035

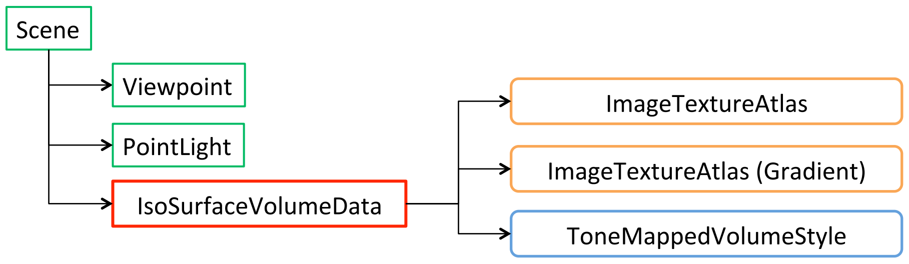
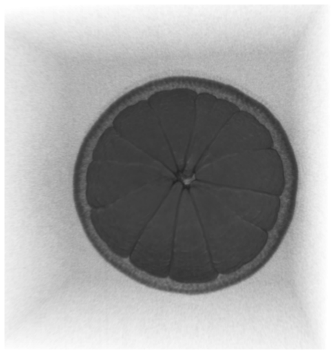


Figure 9.12: Partial X3D scene tree of the `hydrogen-atom` dataset illustrated with the *ToneMappedVolumeStyle*.


(a) Orange dataset

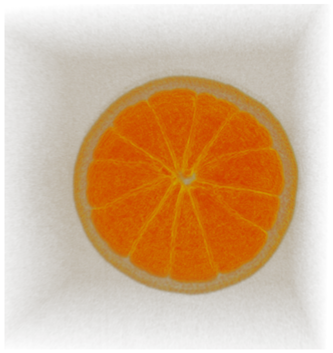

(b) Composed with OpacityMap and Cartoon styles

Figure 9.13: Volume rendering of the `orange` dataset with a 1024 × 1024 atlas. a) Basic volume visualisation (by default *OpacityMap VolumeStyle*). b) `Orange` dataset rendered with the *OpacityMap VolumeStyle* and *Cartoon VolumeStyle*

result, but it would be a more time consuming approach if the transfer function has to be edited manually. Figure 9.14 shows the scene tree used to illustrate the volume with the *Cartoon VolumeStyle*.

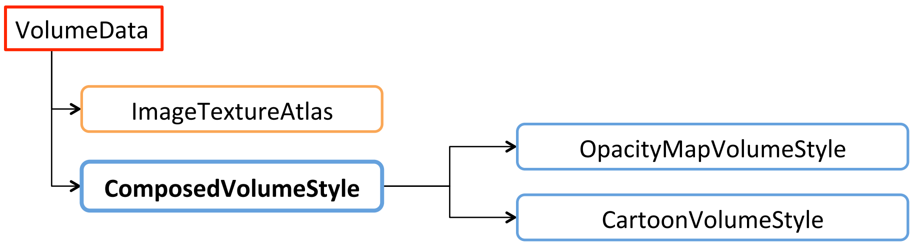


Figure 9.14: Partial X3D scene tree of the `orange` dataset to illustrate the volume.

In this use case, the objective is to highlight the composition of the orange. A quick and effective way to achieve this objective is to enhance the boundaries and silhouette considerably. Figure 9.15 shows the composition of several rendering style nodes, which allows the visualization of the orange sections and seeds.

The final rendering (see Figure 9.15) is produced using the scene described at Figure 9.16. The *ComposedVolumeStyle* provides a way to combine different style nodes. The first stage of this case is to use an *OpacityMap VolumeStyle* to adjust the amount of opacity accumulated on each

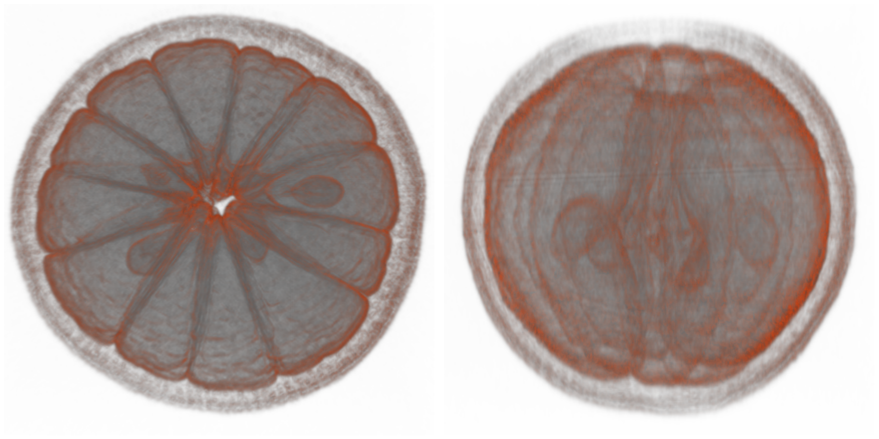


Figure 9.15: Volume rendering of the `orange` dataset with a 1024 × 1024 atlas from two point of views. Applying several composable styles to highlight the sections and seeds of the orange.

sample. Then, applying the *BoundaryEnhancementVolumeStyle* to make it more noticeable to changes between regions inside the orange. Afterwards, the *SilhouetteEnhancementVolumeStyle* has been applied to make slightly more visible the contours of the volume and retains less opacity in uniform areas of the volume. Finally, the *EdgeEnhancementStyle* has been used to fill with colour the previous filtered contour.

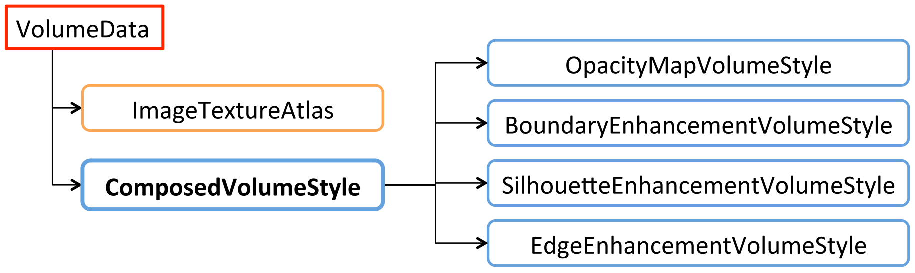

VolumeData
ImageTextureAtlas
ComposedVolumeStyle
OpacityMapVolumeStyle
BoundaryEnhancementVolumeStyle
SilhouetteEnhancementVolumeStyle
EdgeEnhancementVolumeStyle

Figure 9.16: Partial X3D scene tree, enhancing and highlighting the insides of the `orange` dataset.

9.1.5 Performance

The previous examples were carried out on a PC with an Intel Quad Core Q8200 processor, 4GB RAM and a NVIDIA GeForce GTX 295 GPU under Windows 7. Tests were performed with Chrome and Firefox. Both browsers use Google's Angle Library to gain major hardware compatibility by translating OpenGL ES 2.0 API calls to DirectX 9 or DirectX 11 API calls. All the datasets were transferred from an Internet server. Table 9.1 summarizes the performance obtained on each of the cases described before. For the

Use case	Figure	Dataset	Resolution	Gradient	FPS
9.1.1	9.1	`Head MRI`	2048×2048	no	50-55
9.1.1	9.3a	`Head MRI`	2048×2048	no	25-35
9.1.1	9.3b	`Head MRI`	2048×2048	no	40-50
9.1.2	9.4a	`engine`	2048×2048	no	50-60
9.1.2	9.4b	`engine`	2048×2048	no	50-55
9.1.2	9.6	`engine`	2048×2048	no	30-39
9.1.3	9.10a	`neghip`	512×512	yes	40-45
9.1.3	9.10b	`neghip`	512×512	yes	40-45
9.1.3	9.10c	`neghip`	512×512	yes	38-45
9.1.3	9.11a	`hydrogen-atom`	1024×1024	yes	38-45
9.1.3	9.11b	`hydrogen-atom`	1024×1024	yes	40-45
9.1.3	9.11c	`hydrogen-atom`	1024×1024	yes	38-45
9.1.4	9.13a	`orange`	2048×2048	no	50-60
9.1.4	9.13b	`orange`	2048×2048	no	40-45
9.1.4	9.15	`orange`	2048×2048	no	25-35

Table 9.1: Performance, frames per second (FPS) on each use case example at different resolutions. As a help to the reader, the section and figures are referenced. Resolution indicates the *ImageTextureAtlas* size and *Gradient* if a pre-computed surface normals *ImageTextureAtlas* has been used.

creation of the figures and performance tests, 120 steps have been set: each ray is sampled 120 times at maximum in the ray-casting method. If the ray comes out of the volume, or the ray sampling is early-terminated. By default, the publicly released version of the volume rendering component currently shipped in X3DOM is configured with 60 steps. The download time of the datasets does not impact in the performance tests and they are not included in the table.

9.2 Industrial use case: quality control inspection

This section showcases several rendering outcomes in an industrial use case: the quality control of a plastic component acquired with an industrial X-ray microCT scanner. Results validate the benefits provided by the progressive ray-casting approach described in Chapter 6. Performance measures with the *Firefox* browser are also presented.

This section is structured as follows. Firstly, Section 9.2.1 presents the component to be inspected and the initial exploration of the dataset. Section 9.2.3 shows a segmented visualisation for the rendering of the air void analysis. Finally, performance measures are detailed in Section 9.2.4.

9.2.1 Injection moulded plastic part dataset

The selected plastic part is a component used in the medical sector. In particular, the scanned part was produced in a pre-production stage at the initial quality control process for the calibration of the injection mould machine.

To assess the internal state of the manufactured component an industrial Nikon X-ray microCT was used for the volume dataset generation. The original dataset is composed of 1875 slices of 836×939 in 16-bit, with a total size of 2.9 GB. To make this dataset suitable for visualization, firstly, it was downscaled to 8bit (unsigned int) reducing its size to 1,6 GB. Then a region of interest (ROI) was selected (no resolution reduction) obtaining a $720 \times 720 \times 1770$ dataset with a total size of 918 MB. Finally, to make it compatible for the web, it was processed to compose both multi-channel $RGBA$ and $RGB + A$ *ImageTextureAtlases* (see chapter 3). The generated *ImageTextureAtlases* have a resolution size of 15840×15840 and downscaled versions were also created for performance testing.

Figure 9.17 shows the initial rendering of the plastic component from various camera positions.

Progressive rendering allows to effectively explore an navigate through the data with sufficient detail within the web browser. The interaction through the rendering process is never blocked or stalled, so the user can interrupt the current render at any time to change the camera to the desired direction or position.

Using the transfer function editor (see Section 7.4 in Chapter 7) the user can modify the accumulation process of the ray-casting allowing the exploration of the volumetric data. For instance, by changing the control points to select a narrow range of values, the iso-values matching the surface of the plastic component can be identified. This behaviour allows a semi-transparent view of the surfaces. In this way, air void bubbles inside the component can be visualised (see Figure 9.18).

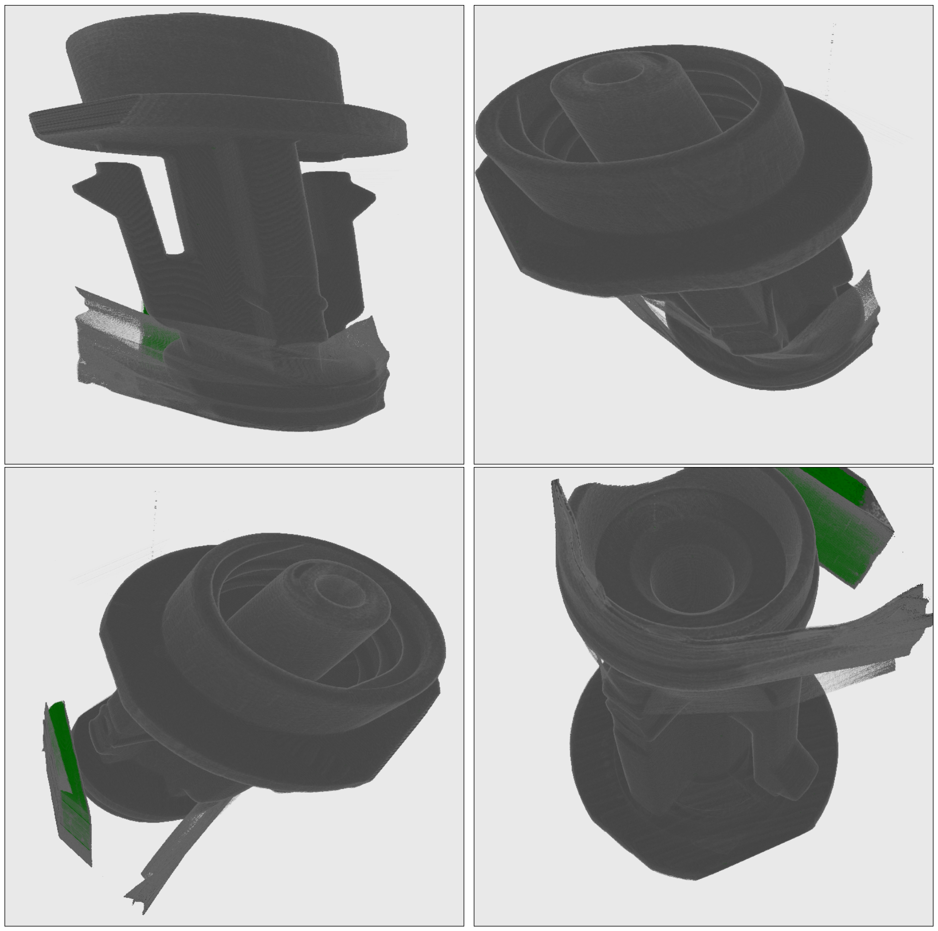


Figure 9.17: Progressive ray-casting of the selected plastic part $720 \times 720 \times 1770$ using 6000 steps.

9.2.2 Air void segmentation

In some industry quality control systems, air voids detection is a critical procedure to determine the quality of the inspected part since they are considered defects. Air void segmentation is the only procedure that is performed at the server side, because it requires a large amount of memory capacity and compute time to process the high-resolution CT scans.

Air void defects are differentiated from the rest of the data by a segmentation. Afterwards, the volume area of each labelled void is quantified. Finally, a multi-channel *ImageTextureAtlas* is composed (see Chapter 3) from the computed results.

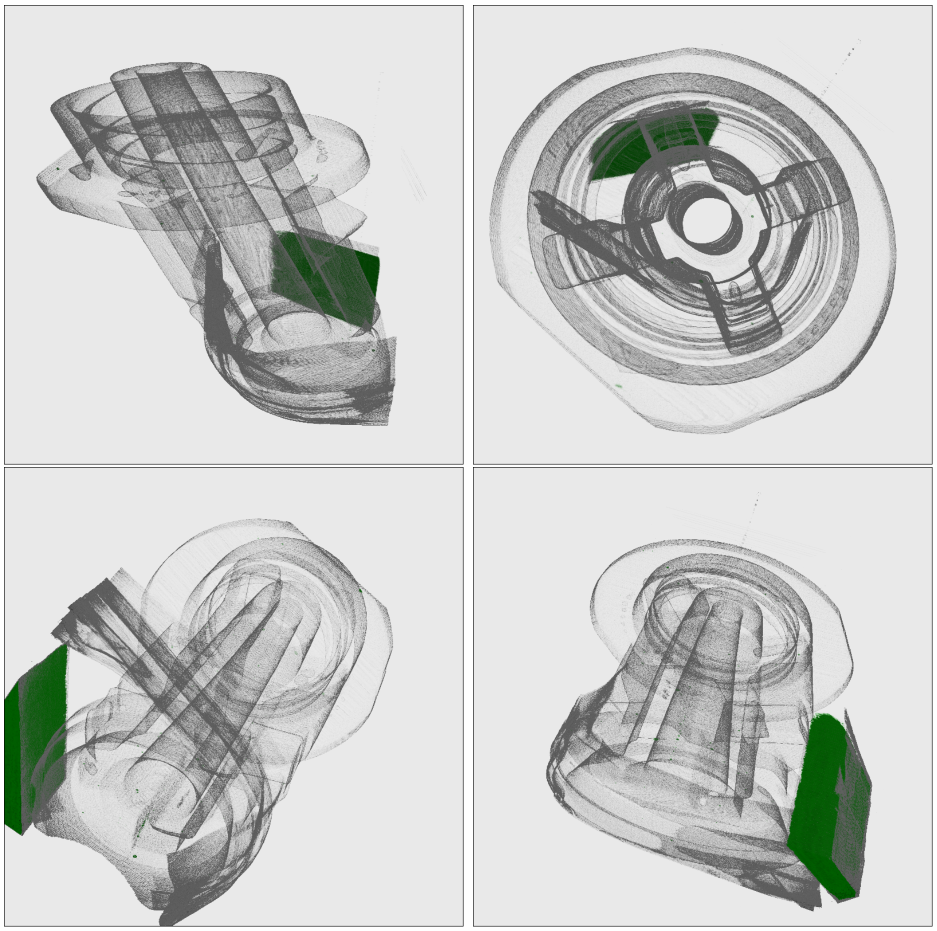


Figure 9.18: Progressive ray-casting of the selected plastic part $720 \times 720 \times 1770$ using 6000 steps. In this case, the transfer function enhances the surface data.

In order to perform the segmentation for the plastic injection moulded part, the Insight Segmentation and Registration Toolkit (ITK) (Johnson et al., 2013) library has been used. First, for each volume slice of the dataset the Otsu's binary thresholding *(itkOtsuThresholdingImageFilter)* is applied to segment the background from the plastic material. To achieve a better background extraction, this thresholding operator is applied locally on each slice, instead of globally to the whole volume. Then, all the binary segmented slices are stacked to create a binarized 3D volume. To the final volume, a binary fill holes filter (*itkBinaryFillHolesImageFilter*) is applied in order to create a 3D mask of the plastic part.

In order to label the multiple air voids in the dataset, the connected components filter *(itkConnectedComponentsImageFilter)* is applied to the segmented binary volume. From this last labelled volume, the background region is discarded by applying the previously computed 3D mask of the plastic part. Later, the spatial volume in voxels is computed for each air void bubble (label) to measure the scale range and a value is assigned to each label in function of its estimated volume size. Finally, the generated 3D labelled volume is sliced in Z axis direction and transformed into a multi-channel *ImageTextureAtlas* (see Chapter 3) that will be transferred back to the client device.

9.2.3 Air void analysis

The plastic component dataset has been processed through a segmentation algorithm (see Section 9.2.2) to identify air voids inside the component. The volume size of each air void is estimated counting the number of voxels and in function of its size, a fixed value in the [0-255] range is assigned to each of them. At the rendering stage, using another TF, this value is used to map each air void to a colour value.

Figure 9.19 shows both volume and segmented data rendered together in one visualisation from different camera view positions.

The segmented volume data also must be composed into a multi-channel *ImageTextureAtlas*, similarly to the volume data. To assist in the assessment of the air void distribution, in the ray-casting algorithm both the volume data and the segmented data are sampled and mixed together. Figure 9.19 shows some air voids, located in the top side of the component. Consequently, it demonstrates that the mould injection machine requires adjustments.

9.2.4 Performance

The performance results for the proposed progressive ray-casting volume rendering algorithm have been measured using a desktop PC with the following specifications: Intel i5-6500 CPU, 8GB RAM and a NVIDIA GTX 960 graphics card. This graphics card supports up to 16384 $\times$ 16384 2D texture size. The selected web browser: *Mozilla Firefox 61* was used under *Windows 10* OS. Table 9.2 summarizes the performed measurements, with the total time (T. time) for the completion of the progressive rendering after each user interaction under different *ImageTextureAtlas* sizes. These times are the average of 5 random camera movements and match to the

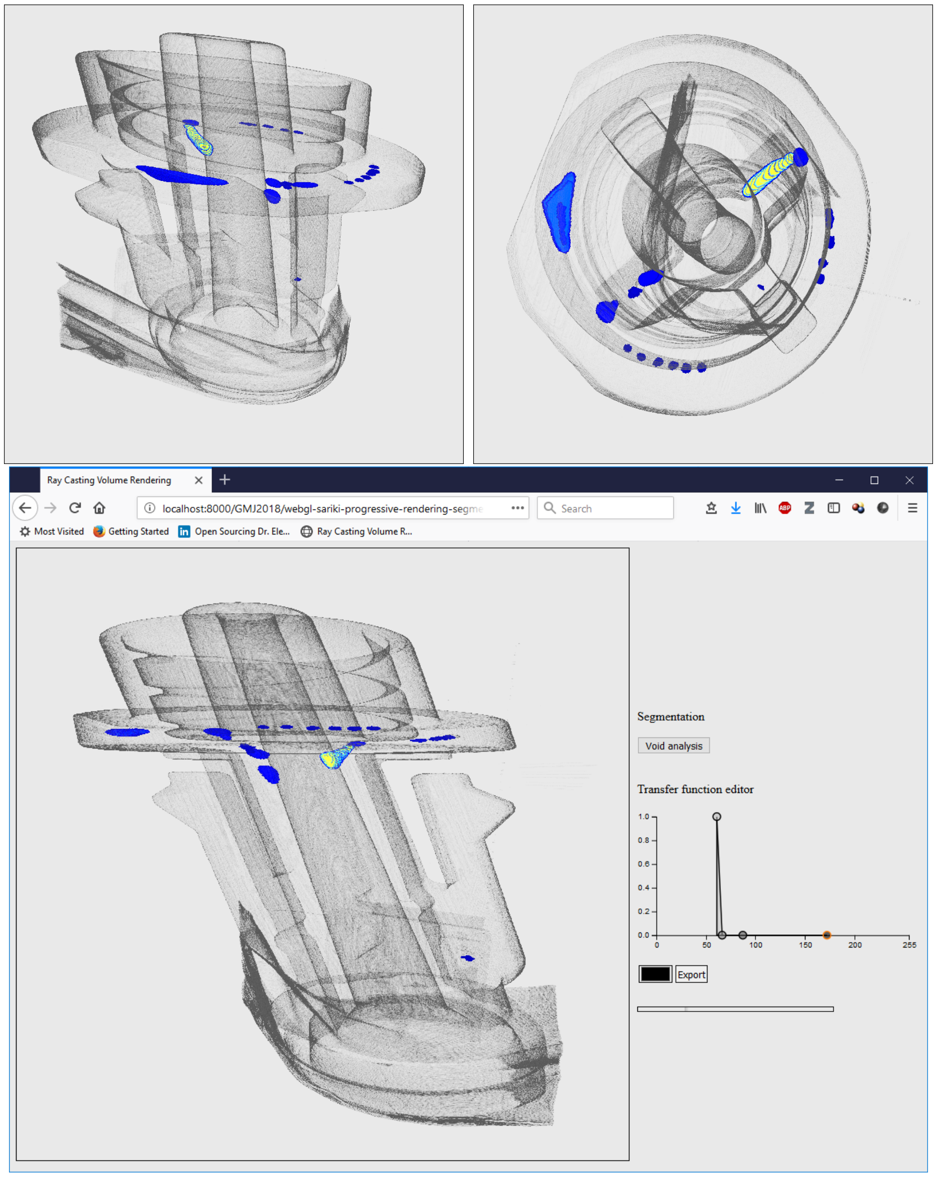


Figure 9.19: Progressive segmented ray-casting visualisation of the selected plastic part 720 × 720 × 1770 using 6000 steps.

visualizations of the presented figures in this section.

Figure	Atlas size	Seg. atlas	Atlas type	T. steps	T. time (s)
9.17	4096 × 4096	no	RGBA	6000	1,14
9.17	4096 × 4096	no	RGB+A	6000	1,05
9.17	8192 × 8192	no	RGBA	6000	1,28
9.17	8192 × 8192	no	RGB+A	6000	1,20
9.17	15840 × 15840	no	RGBA	6000	7,51
9.17	15840 × 15840	no	RGB+A	6000	8,21
9.18	4096 × 4096	no	RGBA	6000	1,25
9.18	4096 × 4096	no	RGB+A	6000	1,16
9.18	8192 × 8192	no	RGBA	6000	1,47
9.18	8192 × 8192	no	RGB+A	6000	1,39
9.18	15840 × 15840	no	RGBA	6000	12,76
9.18	15840 × 15840	no	RGB+A	6000	12,25
9.19	4096 × 4096	yes	RGBA	6000	3,27
9.19	8192 × 8192	yes	RGBA	6000	8,87
9.19	15840 × 15840	yes	RGBA	6000	9,9

Table 9.2: Total rendering completion times for the progressive approach under the presented figure cases and different atlas sizes.

*ImageTextureAtlas*es of 8192 × 8192 resolution fit in dedicated memory, so the rendered is completed in less than 2 seconds. For *ImageTextureAtlas*es of 15840 × 15840 the *NVIDIA GTX 960* graphics card uses a combination of shared and dedicated memory, so the rendering times are considerably larger. In the desktop computer described in the performance tests, results indicate that total completion time for the progressive rendering with $RGBA$ or $RGB + A$ *ImageTextureAtlases* are roughly the same, with non-clear distinction from one over the other.

Additional tests where performed on two mobile devices: *Xiaomi Mi5s* (smartphone) and a *Xiaomi MiPad* (tablet). On both devices the total completion times where between 3-5 seconds for a 4096 × 4096 texture size and 6000 steps. However, for these devices, the $RGB + A$ texture type is always slightly faster than the $RGBA$ atlas. On these devices the memory access is more critical and the additional texture sampling required in the last colour channel (alpha) penalizes the performance to some degree.

9.3 Time-varying data use case: 4D volume visualization

This section showcases the potential of the web compatible *ImageTextureAtlas* data structure presented in Chapter 3. In combination with the X3DOM volume rendering component in Chapter 4, it can be used to create a 4D volume rendering visualization.

There are a number of use cases where the volume data evolves over time. For example, in the medical field: 4D CT scanning (Pan et al., 2004) and 4D MRI capture are already possible (Stankovic et al., 2014). In the geosciences field, data which evolves over a large period of time is also of interest of study (Ho and Jern, 2008). With 4D volume rendering (Dani Tost, 2006). is possible to recreate the 3D data movement (variation) over time and visualise it in real-time.

Using the proposed *ImageTextureAtlas* and the HTML5 capabilities of current modern browsers, 4D visualizations can be rendered in the web browser (see Section 3.2.4 in Chapter 3). To validate this approach, two time-varying use cases have been selected. Section 9.3.1 presents the volume rendering of a heart in motion.

9.3.1 AGECANONIX dataset: heart beat visualization

OsiriX is a DICOM viewer software with volume rendering support for the MacOS platform and it has a public repository with datasets for research purposes. The `AGECANONIX` dataset (OsiriX, 2018) is available from this repository. This dataset is originated from a cardiac and coronary study and it includes a 4D CT acquisition from a series of 3D dynamic scans in 10 phases. Figure 9.20 shows data slices of the same spatial location evolving over time.

This approach exploits the native video reproduction and 2D canvas API features of HTML5 modern browsers. Figure 9.21 summarizes the architecture. To create a web compatible 4D volumetric data structure, the `AGECANONIX` dataset is converted into an *ImageTextureAtlas* video. First, a *window level* has been selected for the DICOM data and converted into a series of 8-bit images. For each time frame, the set of slices of a 3D scan are converted into an *ImageTextureAtlas*. As a result, a series of *ImageTextureAtlas*es are generated, one for each acquired time step. All the *ImageTextureAtlas*es are added sequentially as frames of a video. This process has been performed offline with open source tools and it can be automatized in the server side.

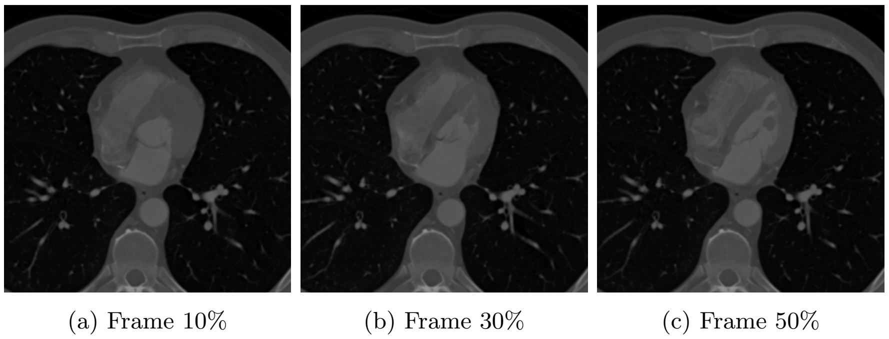


Figure 9.20: `AGECANONIX` dataset ($512 \times 512 \times 35$). Different frames of the same position (slice) in Z axis: 17.

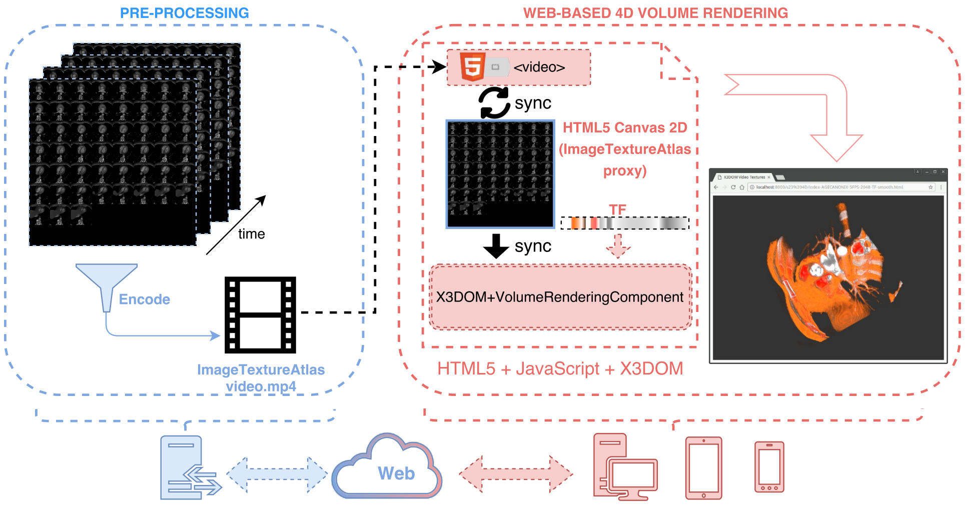
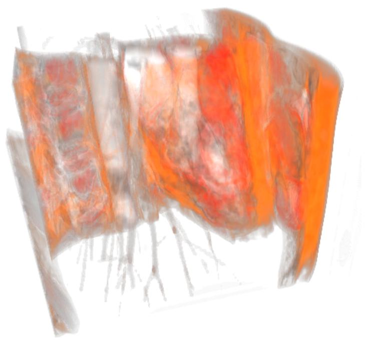
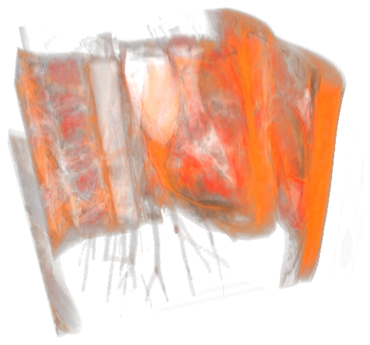

PRE-PROCESSING
time
Encode
ImageTextureAtlas video.mp4
WEB-BASED 4D VOLUME RENDERING
<video>
sync
HTML5 Canvas 2D (ImageTextureAtlas proxy)
TF
sync
X3DOM+VolumeRenderingComponent
HTML5 + JavaScript + X3DOM
Web

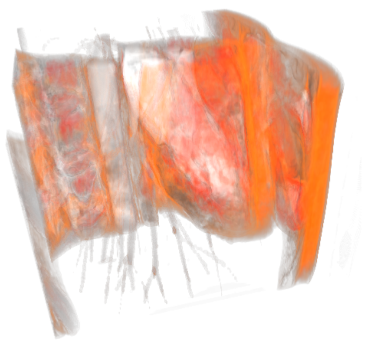
Figure 9.21: Scheme of the proposed architecture for a 4D volumetric visualization with the `AGECANONIX` dataset.

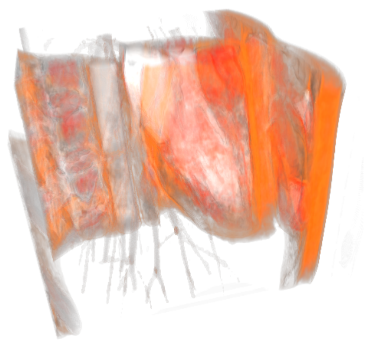
When all the atlases are converted and encoded into a video, this resource is sent to the client device and played in a loop within the browser context. However, it is important to remark that the video will be hidden to the user and playing in the background.

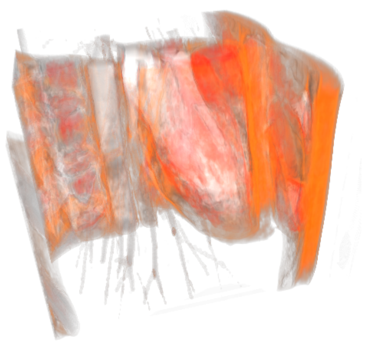
While the *ImageTextureAtlas* video is being reproduced in the background, the content of each frame is uploaded to the GPU periodically. Figure 9.22 shows a series of sequential renders at different time steps with the same camera position to show the evolution of the visualization over time. This approach allows the web based real-time 4D volume rendering visualisation of the heart beat motion (*AGECANONIX*).


(a) Time 0.0s


(b) Time 0.2s


(c) Time 0.4s


(d) Time 0.6s


(e) Time 0.8s

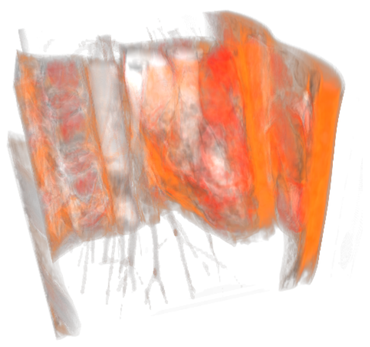

(f) Time 1.0s

Figure 9.22: Evolution of the `AGECANONIX` dataset (heart motion) at different frames during a period of one second.

9.4 Medical use case: Mirror4all

This section presents a web-based application designed for the medical domain. The *Mirror4all* (Kabongo and Arbelaiz, 2018a) web application is intended to be used by the general public (non-experts on the field). This web application is an ubiquitous web-based DICOM volumetric data visualisation viewer.

The target user of this application already has in his possession (on his device) a series of DICOM files to be visualised. With *Mirror4all* the user only requires to navigate to the application URL using a browser and drag and drop the DICOM files into the application. All the components of the application run on the client side (the user device). Therefore, no patient data is transferred to any web server. This approach allows to ensure data privacy and reduce the effort required to create secure communication channels between a client device and the application server.

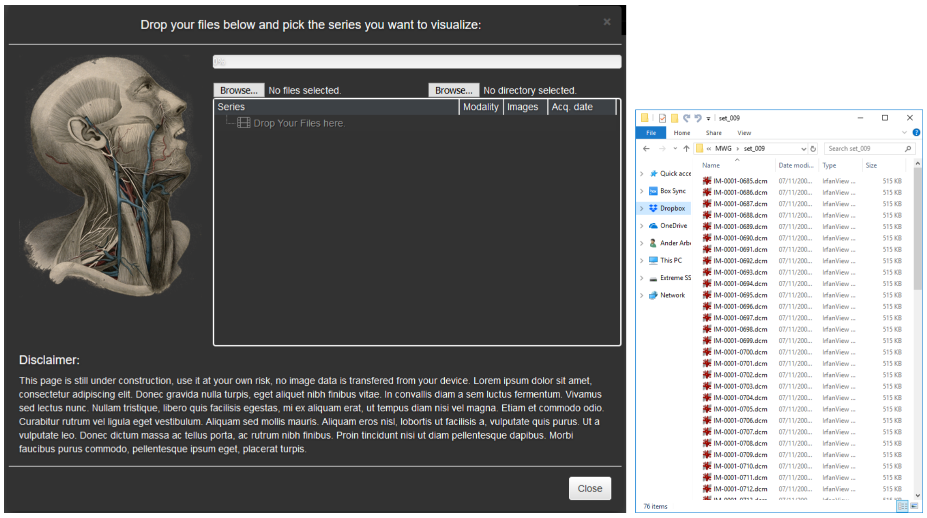
Mirror4all showcases the combination of previously presented components into a single application. It is built based on contributions presented in Chapter 4 and Chapter 7. This application is publicly available in GitHub, as an open source project (Kabongo and Arbelaiz, 2018b). Content developers can take this application as an example to build their own custom solution.

Once the user enters in the application landing page, the first step is to drag the DICOM files and drop them into the designated area. The application also offers an alternative UI element: a button that opens a file explorer window to select the DICOM files. Figure 9.23 shows the landing page for the drag and drop step.

Once the DICOM files have been loaded the dataset information is displayed. The Cornerstone (Cornerstone, 2016) library is used to parse DICOM data to generate the *ImageTextureAtlas* required by the volume rendering component (see Section 7.1.1 at Chapter 7). Figure 9.24 shows an example of parsed DICOM metadata from the `INCISIX` dataset (OsiriX, 2018). If it has been parsed correctly the user can proceed to the volume visualization.

The rendering of the volumetric data is performed using the X3DOM volume rendering component (see Chapter 4) where the *OpacityMapVolumeStyle* is used to illustrate the volume with a transfer function. Some UI elements and visual clues have been added to help the users to interact with the volumetric rendering: *Window Level* modification and an interactive transfer function editor.

Several DICOM files have been tested successfully. They were acquired

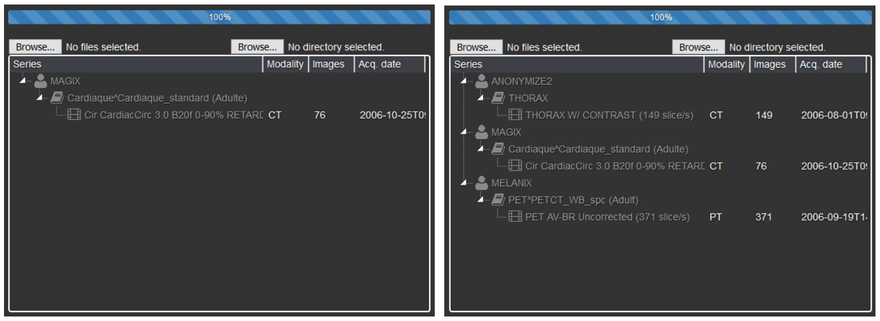


Disclaimer:

This page is still under construction, use it at your own risk, no image data is transfered from your device. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec gravida nulla turpis, eget aliquet nibh finibus vitae. In convallis diam a sem luctus fermentum. Vivamus sed lectus nunc. Nullam tristique, libero quis facilisis egestas, mi ex aliquam erat, ut tempus diam nisi vel magna. Etiam et commodo odio. Curabitur rutrum vel ligula eget vestibulum. Aliquam sed mollis mauris. Aliquam eros nisl, lobortis ut facilisis a, vulputate quis purus. Ut a vulputate leo. Donec dictum massa ac tellus porta, ac rutrum nibh finibus. Proin tincidunt nisi ut diam pellentesque dapibus. Morbi faucibus purus commodo, pellentesque ipsum eget, placerat turpis.

Name	Date modi...	Type	Size
IM-0001-0685.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0686.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0687.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0688.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0689.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0690.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0691.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0692.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0693.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0694.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0695.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0696.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0697.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0698.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0699.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0700.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0701.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0702.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0703.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0704.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0705.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0706.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0707.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0708.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0709.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0710.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0711.dcm	07/11/200...	IrfanView ...	515 KB
IM-0001-0712.dcm	07/11/200...	IrfanView ...	515 KB

Figure 9.23: Drag and Drop user interface to upload the DICOM files in the *Mirror4all* application.


100%
Browse... No files selected.
Browse... No directory selected.

Series	Modality	Images	Acq. date
MAGIX			
Cardiaque^Cardiaque_standard (Adulte)			
Cir CardiacCirc 3.0 B20f 0-90% RETARE	CT	76	2006-10-25T0!

Series	Modality	Images	Acq. date
ANONYMIZE2			
THORAX			
THORAX W/ CONTRAST (149 slice/s)	CT	149	2006-08-01T0!
MAGIX			
Cardiaque^Cardiaque_standard (Adulte)			
Cir CardiacCirc 3.0 B20f 0-90% RETARE	CT	76	2006-10-25T0!
MELANIX			
PET^PETCT_WB_spc (Adult)			
PET AV-BR Uncorrected (371 slice/s)	PT	371	2006-09-19T1

Figure 9.24: User interface of the parsed DICOM datasets in *Mirror4all*.

with different technologies (CT, MRI) of different manufacturers (vendors). DICOM parsing is implemented with the Cornerstone library (Cornerstone, 2016) and it and it should be considered a work in progress as it does not support all the features. However, DICOM parsing is out of the scope of this thesis.

This application has been used as an example of X3D capabilities promoted by the X3D medical working group and it has been showcased in the Web3D forums by members of the Web3D Consortium.

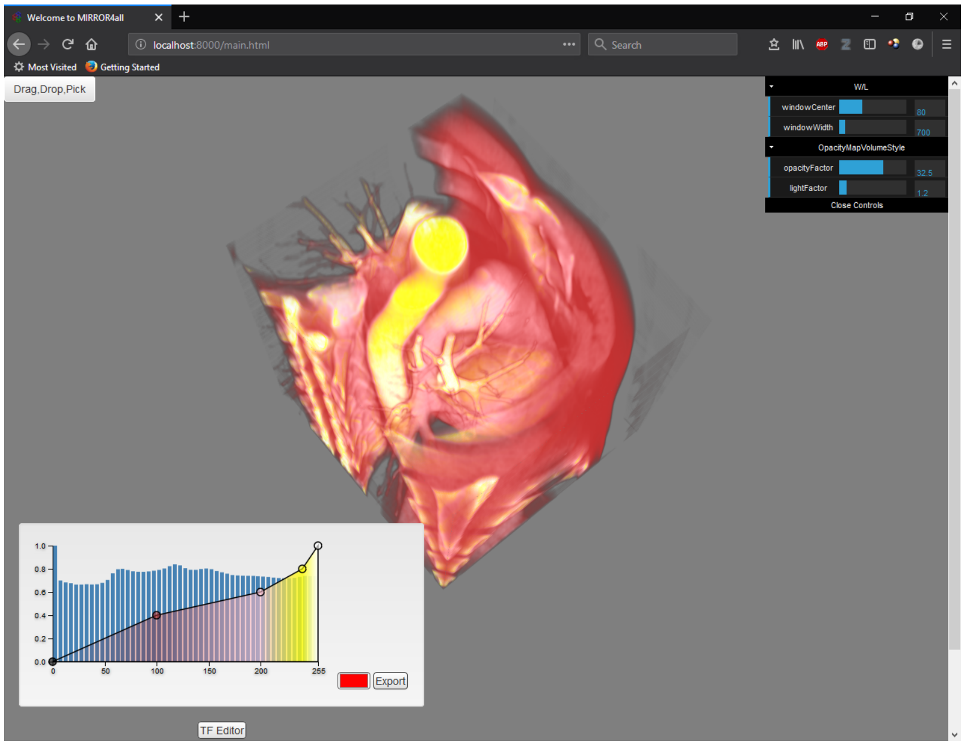


Figure 9.25: Volumetric visualisation of the uploaded DICOM data with the `MAGIX` dataset in the *Mirror4all* application.

9.5 Virtual Reality use case: WebVR application

The creation of new VR oriented content with the WebVR (WebVR, 2016) framework is simplified to the extent that only the knowledge of HTML, JavaScript, CSS and related web technologies are required to develop VR experiences. But even with this simplicity, teachers, researchers and other non-technological-aware target groups are not prepared for such a task.

To address this issue, the volume rendering component presented in Chapter 4 can be used. The declarative nature of X3D scene suits better for most of the people who want to deploy VR experiences but lack the technological capabilities of doing everything by themselves. This section showcase a WebVR application prototype that combines the medical oriented features implemented in Chapter 7 and the volume rendering component released under the X3DOM framework in Chapter 4.

The preliminary prototypes have been tested with the Oculus Rift DK2, under Windows 10, with the 1.5 and 1.6 runtimes. A developer build of the Chrome browser with WebVR support (Chrome, 2016) has been used to test the VR developments.

Once the setup is running and a X3D volume rendering scene is declared, loading a VR experience is as easy as loading a URL. Then, the VR experience is started by clicking in a *Enter VR* button or the *white goggles* icon, (see Figure 9.26).

One of the key elements in the success of the VR is to control how the VR experiences are perceived by the users. In this regard, the transitions from non-VR to VR and backwards should be controlled. In addition to this, any sudden change in the scene should be carefully treated to avoid intrusive pop-up effect in the user field of view.

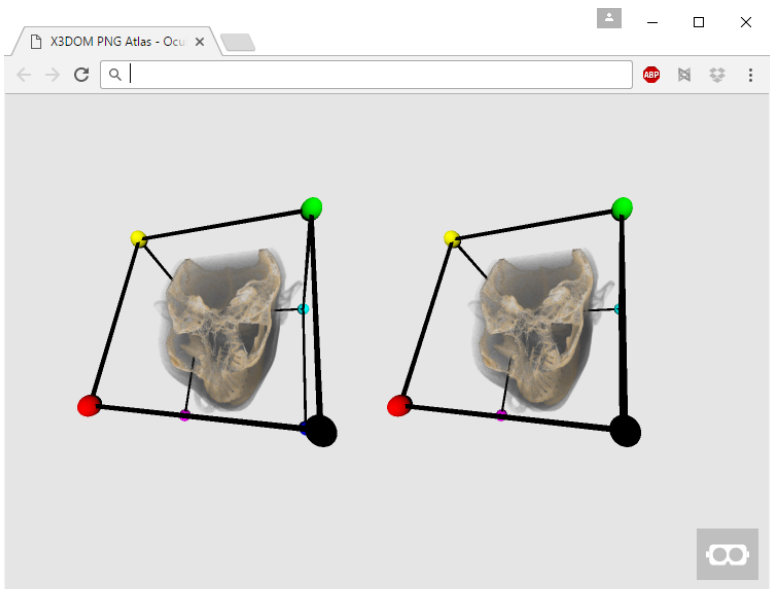
In the developed prototypes, DICOM files (through Cornerstone JavaScript library) are loaded asynchronously, i.e, each slice of the volumetric dataset is retrieved as soon as it is loaded by the library. Therefore, initially the 3D scene could be empty, which is something to avoid in order to reduce stress to the users: being *floating* in empty space is against the most basic ergonomic rules regarding VR (Rebenitsch, 2015).

To solve this potential situation, the empty 3D cube where the volume will be loaded is surrounded with a 3D wireframe with coloured solid spheres in the corners of the cube (see Figure 9.26). This solution offers two benefits: *i)* the scene is not empty while the dataset is being loaded and *ii)* the coloured spheres give visual clues of the orientation of the scene that can be used in case the users got lost. This 3D visual clue has been proved very helpful in VR environment, but also in non-VR setups.

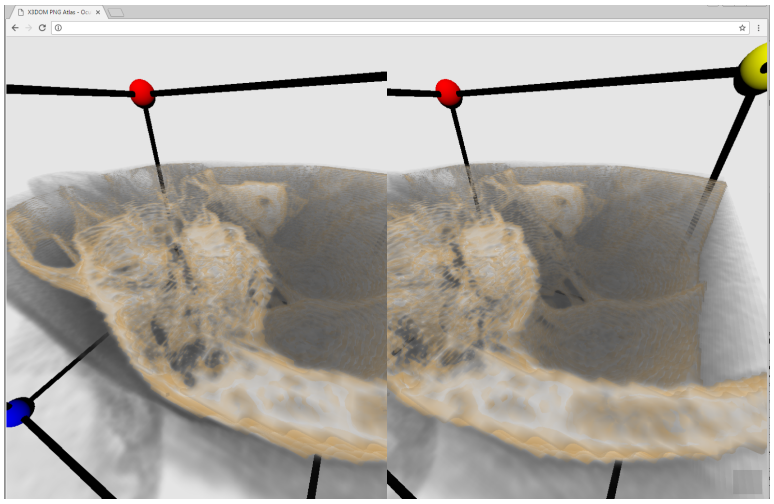
The HMD's like Oculus Rift are more focused in immersive environments where the user is transported to a virtual world that can be navigated and explored in *first person*. In this volume visualisation use case, the typical navigation and exploration of the volumetric dataset is more compatible with the *orbit* navigation style, which conflicts somehow with the nature of the VR. The reader is referred to this collection of publications (Christie and Olivier, 2009) to get an insight of the different camera styles in virtual environments.

To cope with this situation, the VR interaction capabilities were mapped to the actions that should be carried out when the volumetric datasets are being displayed:

- Mouse: Typically in VR, the mouse rotates the user in the world. In this case, the mouse rotates the 3D volumetric dataset around its center, like in the *orbit* navigation style.



(a) Firefox browser



(b) Chrome browser

Figure 9.26: Web based volume rendering in a VR scene with WebVR. Using the `INCISIX` dataset from the *OsiriX* repository (OsiriX, 2018), left and right eye images. a) inspection from a far point of view. b) visualisation from a closer point of view. The wireframe cube and the solid coloured spheres help the users to know their location at any time.

- Arrow Keys: Typically in VR, the keys allow to move freely in the virtual scene (in walking or flying mode). In this case, the keys have been disabled. In order to get closer to the volume or to inspect it from the inside, the wheel mouse is used to *zoom* into the scene.

- Head tracking: The modern HMD's provide information about where the user is looking at. This functionality was kept as it would be very intrusive to remove it.

- Information display: In VR mode all UI elements were removed (TF editor, window level management...) in order to provide a clean VR experience. Only the *Exit VR* button is present. In preliminary tests another button was added to reset the viewpoint in case the user got lost, but ultimately, it was discarded. It was easier for the user to get out VR mode in that extreme case.

The resulting VR experience allows viewers to inspect the volumetric dataset from any point in an easy and straightforward manner. The developed prototypes has introduced preliminary efforts to bring VR visualisation on the Web browser by using a combination of open web technologies (WebGL and WebVR) and the contributions presented in previous chapters. DICOM datasets are supported through the Cornerstone JavaScript library (see Chapter 7) in combination with the X3DOM volume rendering nodes presented in Chapter 4 to render the volumetric data.

Preliminary research activities have dealt with situations regarding the VR environments in the web platform and the transitions among them. The carried out tests show that there is room to improve the Human-Computer Interaction within the web environment and the current and recently models added to the collection of HMD devices: Oculus Rift, HTC Vive, Microsoft Hololens, Samsung GearVR, PlayStation VR, Google Cardboard and Daydream, etc.

The presented mixture of web technologies provide a real ecosystem that can facilitate the deployment of VR experiences of volumetric datasets for experts in their corresponding fields. They do not need to know about the visualization techniques that create the virtual experience.

9.6 Web3D community applications

This sections presents works from other authors that have taken the publicly available X3DOM volume rendering component (see Chapter 4) as a base

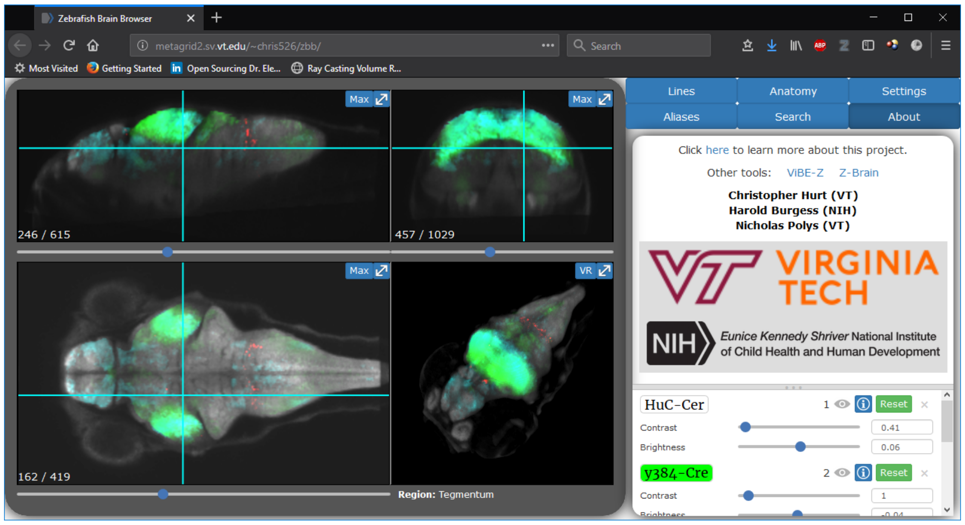


Figure 9.27: Zebrafish Brain Browser (Hurt et al., 2018) after selecting some Cre lines.

to create new web based application that require volumetric visualization. The following research works and applications demonstrate the utility and impact of the contribution of this thesis towards an ubiquitous approach to volume rendering. Some of the following research projects have extended the volume rendering component to cope with visualisation and interaction requirements on their targeted field.

9.6.1 Zebrafish brain browser

To explain and deduce the functional circuitry of the brain, visualisation of neuronal cells and record activity from identified neurons is a key process. Tabor et al. (2018) have created a zebra fish brain browser visualization interface. The online brain browser includes features for 3D spatial search, prediction of the area of intersectional expression between selected lines, MIP projection and information about the neuroanatomical identity to any selected voxel. It facilitates the reproducible targeting of neuronal subsets for circuit-mapping studies. Figure 9.27 shows their web browser application (Hurt et al., 2018) publicly available for general use.

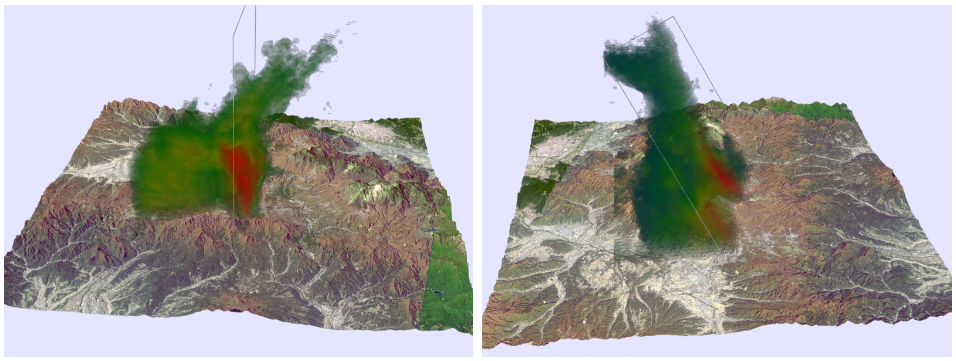


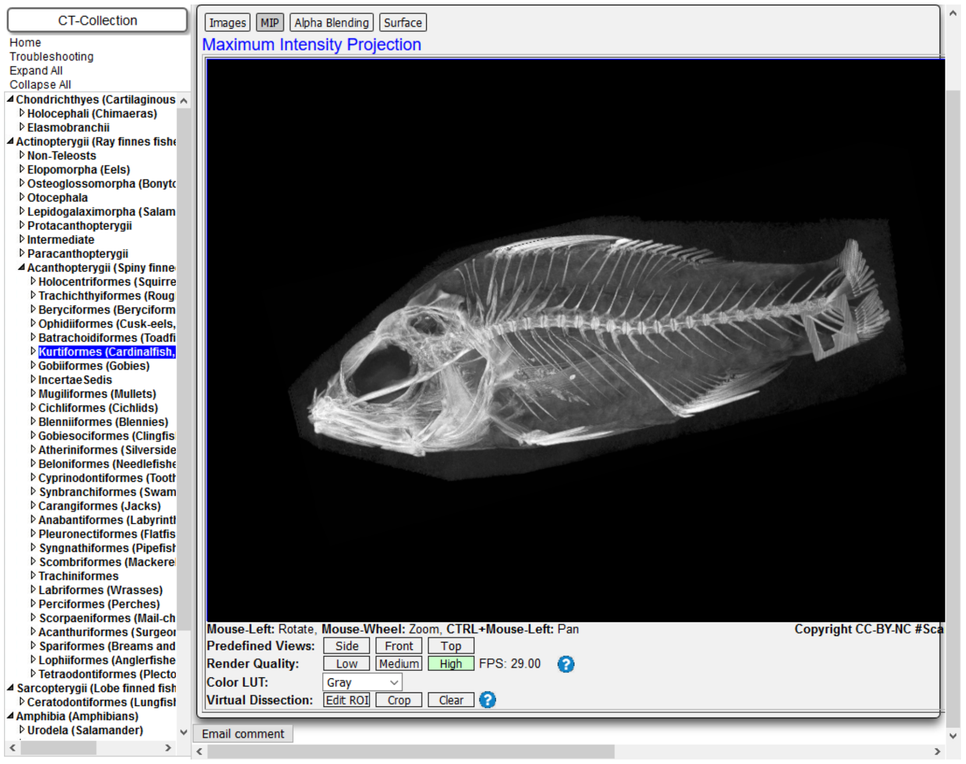
Figure 9.28: X3DOM example of weather radar volume visualisation with the *RadarVolumeStyle* (Yang et al., 2018).

9.6.2 Weather radar data visualization

Yang et al. (2015) presented a texture data compression technique for efficient transmission of volumetric weather radar data. They combine the S3TC compression method encoding the volume data in the RGBA channels of an image followed by DEFLATE compression. They also extended the X3DOM volume rendering component with a new rendering style to specifically visualise the volumetric weather data. Figure 9.28 shows their public example where a volumetric weather radar dataset can be inspected.

9.6.3 Virtual Natural History Museum

The Virtual Natural History Museum (VNHM) (Hofmann, 2018) is an online virtual museum created by Michael Hofmann. It is a work in progress that contains a large catalogue of CT scans of vertebrates, primarily composed of fishes. These are freely available for the scientific community. Currently the catalogue is composed of 5902 CT-scans. The website allows the volumetric visualization, maximum intensity projection and iso-surface visualisation of the CT scans. Figure 9.29 shows an example of maximum intensity projection rendering of the `Pristiapogon fraenatus`.



Mouse-Left: Rotate, **Mouse-Wheel:** Zoom, **CTRL+Mouse-Left:** Pan **Copyright CC-BY-NC #Sca**
Predefined Views:
Render Quality: High FPS: 29.00
Color LUT: Gray
Virtual Dissection:

Figure 9.29: MIP volume visualisation of the *Pristiapogon fraenatus* in the *VNHM* website (Hofmann, 2018).

Chapter 10

Conclusions

This thesis concludes summarizing the contributions presented in the previous chapters. Section 10.1 enumerates the contributions and Section 10.2 describes them in more detail. Finally, the future work is discussed in Section 10.3.

10.1 Summary of contributions

Previous chapters have contributed to the state of art in WebGL-based volume rendering. This results make possible an ubiquitous volume data rendering. The following items outline these contributions:

i) The extensions to the *ImageTextureAtlas* web compatible structure (see Chapter 3) have improved previous approaches in various aspects.

ii) The X3D volume rendering component (see Chapter 4) allows content developers to easily declare and generate volume visualizations within web pages.

iii) The hybrid volume rendering algorithm (see Chapter 5) solves the problems raised by the simultaneous visualization of volumetric and surface data.

iv) The progressive volume rendering algorithm (see Chapter 6) enables the interactive high-quality rendering of large volume datasets.

v) Reusable components have been designed and variations of the rendering algorithms have been proposed to solve some problems in the visualization of medical datasets (see Chapter 7).

vi) New extensions have been drafted to improve the current features of X3D standard (see Chapter 8).

These contributions have been validated in Chapter 9 under different domains. Showcased applications and use cases demonstrate the potential of these contributions to create new web based applications that can make use of volume visualization. Furthermore, the work of this thesis has already been used by other authors to contribute to the state of the art (Yang et al., 2015; Tabor et al., 2018).

10.2 Description of contributions

This section describes in more detail the scope of the contributions enumerated in Section 10.1.

10.2.1 ImageTextureAtlas – Web compatible volumetric texture

As stated in Chapter 1, WebGL is the graphics API to target for ubiquitous volume rendering. Unfortunately, the WebGL 1.0 API does not support the basic input data structure required for volume rendering: 3D textures. The *ImageTextureAtlas* enables to store the volume data to be used with volume ray-casting in all compatible WebGL devices (see Chapter 3).

The contributions to the *ImageTextureAtlas* are the following ones:

i) This work validates the *ImageTextureAtlas* as a mean to provide pre-computed gradient data to compute non-photorealistic renderings and local-illumination (see Chapter 3).

ii) It has been proved the compatibility of the *ImageTextureAtlas* structure with web standards that allow the dynamic composition of the structure based in DICOM data (see Chapter 7). This process is performed in the client device (web browser). Thus, the privacy of the patients data is maintained.

iii) It has been proven the use of this data structure to create 4D interactive volume rendering visualizations in the Web browser (see Section 9.3.1 and Chapter 3).

iv) A novel multi-channel order composition has been proposed to accommodate larger datasets and to improve the sampling performance of the *ImageTextureAtlas* (see Chapter 3).

10.2.2 Automatic shader composition and generation

Each field requires modifications in the algorithms to enhance the characteristics of the volume data in the visualisation step in a different manner. Hardware acceleration (GPU) is required to achieve an interactive experience. Consequently, not only substantial expertise in volume rendering algorithms is required, but also skills in GPU programming are required.

This obstacle should be removed: *content developers* work must be freed from these difficult challenges. They do not need to have advanced skills or expertise in the computer graphics domain. The declarative solution solves this problem by providing a set of composable X3D nodes that can be applied in multiple fields (see Chapter 4).

The nodes in the *volume rendering component* automatically compose the required GPU shader programs required by the compound of rendering styles requested by the content developer, transparently to the user. The declarative approach makes easier to learn how to define a volumetric scene and the provided framework solves the technical constraints needed to achieve a real-time rendering.

The approach of the presented component is well suited for the Web in terms of scalability, because the rendering computation is made on the client device. Other solutions that use servers for the rendering computation could have a potential scalability problem when the number of simultaneous users increases.

From the point of view of web content developers, this contribution empowers them providing the following simple tasks to make use of the wide range of solutions that Chapter 9 shows. Firstly, web content developers must store the volume data on a web repository. Afterwards, a web document is created which references to the X3DOM framework and the presented volume rendering component in Chapter 4. The document only includes the declared X3D scene using HTML markup language. Web content developers that are familiar with the X3D standard specification can integrate a volume rendering canvas within a web page.

In a nutshell, the component developed in this thesis makes it easier to create volumetric content for developers without specific knowledge on computer graphics rendering. The required GPU shaders are generated under the hood.

Overall, it can be asserted that the results achieved in this research work are valid to be used in consumer oriented desktop computers with domestic PC graphics cards. Chapter 9 validates the use of the presented component under several domains and Table 10.1 summarizes the advantages and disadvantages of this web based methodology.

X3D scenes can be easily exchanged between applications and users. Additionally, the X3DOM framework and the volume rendering component are publicly available for all compatible WebGL devices, providing an ubiquitous solution to share volume rendering content. The interactivity rate of the visualisation is only limited by the GPU computational power of the targeted device.

Web based approach	**Desktop based approaches**
Datasets up to $512 \times 512 \times 512$ at interactive rates	Larger volume datasets
GPU restrictions through WebGL	No GPU API restrictions
Seamless integration with the Web	Desktop oriented applications
No need for software installation	Software installation required
One deployment for multiple platforms	Applications targeted only to the desktop platform
Declarative scene and style composition	Programming skills required to deploy an application
Easy sharing of scenes and volume data across devices (URL's)	Sharing is complex: requires transferring local volume data and the installation of an application

Table 10.1: Summary of the key advantages and disadvantages of the proposed web based approach against desktop based approaches.

10.2.3 Hybrid volume rendering

There are cases in which polygonal meshes are needed to be rendered with volumetric data at the same time and in the same virtual scene, for example in a surgical simulation or cases like in engineering simulation where the addition of polygonal meshes provide semantic information to the scene. Chapter 5 presents and validates how the proposed algorithm solves the problem of hybrid volume rendering: scenes that mix volume data and polygonal models.

10.2.4 Progressive volume rendering

The quality of the rendered image is dependant on the discretization level of the rendering integral (see Equation 2.1 in Chapter 2) and the properties of the volume dataset. The volume rendering integral assumes that the volume data is a continuous signal. Therefore the resolution accuracy when sampling values in the volume data determines the output resolution. Additionally, the step size used for the discretization of the integral (numerical approximation) will determine the correctness of the computed radiance solution: larger datasets imply that the distance traversed by the ray is larger and consequently, the number of steps (number of samples along the ray) must be increased to correctly approximate the radiance value. The presented progressive volume rendering algorithm solves the challenge of computing such large number of steps, while maintaining interaction with the visualization. It allows to compute a very high number of steps by dividing the computation load into subsequent frames.

10.2.5 Medical volume data visualization

The medical domain requires more advanced functionalities that complement volume rendering to create useful visualization tools for practitioners. Among them, this thesis has contributed in the following (see Chapters 7 and 9).

1. The visualization of DICOM volume data (medical de-facto file format) within the web browser context has been demonstrated using the proposed volume rendering component.

2. An approach for the volume rendering of segmented data has been presented and validated.

3. The single-pass ray-casting algorithm has been extended to support the inside exploration of the volume data.

4. A reusable component has been created to facilitate the transfer function creation and edition in combination with the presented volume rendering component. The component allows the user to illustrate and explore the volume data in real-time within the web page.

5. The experience to create an immersive virtual reality volume visualization with WebVR has been presented.

The components and approaches presented in Chapter 7 show a possible solution that developers can use to build applications and web based visualizations in an easy way (declarative HTML5) taking as base the other contributions presented in this thesis, such as the X3DOM volume rendering component and the *ImageTextureAtlas*.

10.2.6 Extension proposals for the X3D standard

In the first phase of this thesis work, the X3D standard was taken as a foundation stone. The thesis contributes with a web based volume rendering component in the X3DOM framework. This component has been publicly available for the Web3D community. Being X3DOM an open source project, the volume rendering component has received feedback from content developers.

Taking into account the community feedback and the research work performed during this thesis, an X3D extension proposal has been applied. This proposal contributes to the X3D specification with new features to better cope with the Web platform ecosystem and to enable ubiquitous access for volume rendering.

This thesis wants to contribute to the X3D standard. In order to achieve this milestone several steps were identified.

1. Define the extensions specification as required by X3D submission guidelines.

2. Demonstrate that the proposed approach can be used in several platforms using the presented extensions.

3. Demonstrate that any 3D volume rendering application can be easily constructed using the proposed solutions.

4. Obtain approval from the X3D and scientific community and the Web3D Consortium.

5. Perform an official final proposal for the inclusion of the extensions to the X3D standard.

In Chapters 3, 4, 8 and 9, the steps 1 to 3 have been completed and validated. Step 4 has been already started through a research publication (Arbelaiz et al., 2017b), as well as with the presentation of the proposed extensions (see Chapter 8) in the *X3D Medical Working Group* of the *Web3D Consortium*. Future work will be focused on completing step 5.

10.3 Future work

The presented contributions have demonstrated the viability of the web platform for the visualization of volumetric data. However, there is still, a gap to reach the photorealistic renderings that are achievable in other platforms.

There are still improvements that could be applied to the proposed algorithms with techniques such a *pre-integration* that have not been applied in WebGL yet and could potentially have a positive impact in performance and in the rendering quality.

The web platform is evolving in a fast pace and new technologies will make possible a better use of the underlying hardware. The WebGL 2.0 API, although is not ready as an ubiquitous enabler technology, is a good candidate to study the out-of-core rendering of volumetric content. Further research could be dedicated to improve the streaming of volume datasets and rendering of volume data over the Web.

In the long run, the WebGPU API postulates as a disruptive standard that could be the key enabler technology which allows to make use of the same hardware capabilities as desktop APIs, but in the browser. This could lead to the research of new rendering techniques that are not possible now with the current technologies.

In the short run, future work will be directed towards pushing the standardization of X3D extension proposals presented in this thesis.

Bibliography

A. Arbelaiz. A web based transfer function editor component, 2018. URL `https://github.com/VolumeRC/tf-editor`.

A. Arbelaiz, A. Moreno, and L. Kabongo. Deployment of Volume Rendering Interactive Visualizations in Web Platforms with Intersected 3D Geometry. In *Proceedings of the XXVI Spanish Computer Graphics Conference*, CEIG '16, pages 37–42, Goslar Germany, Germany, 2016a. Eurographics Association. ISBN 978-3-03868-023-9. doi: 10.2312/ceig.20161312. URL `https://doi.org/10.2312/ceig.20161312`.

A. Arbelaiz, A. Moreno, L. Kabongo, and A. García-Alonso. X3DOM volume rendering component for web content developers. *Multimedia Tools and Applications*, pages 1–30, 2016b. ISSN 1573-7721. doi: 10.1007/s11042-016-3743-1. URL `http://dx.doi.org/10.1007/s11042-016-3743-1`.

A. Arbelaiz, A. Moreno, L. Kabongo, and A. García-Alonso. *Volume Visualization Tools for Medical Applications in Ubiquitous Platforms*, pages 443–450. Springer International Publishing, Cham, 2017a. ISBN 978-3-319-49655-9. doi: 10.1007/978-3-319-49655-9_54. URL `http://dx.doi.org/10.1007/978-3-319-49655-9_54`.

A. Arbelaiz, A. Moreno, L. Kabongo, N. Polys, and A. García Alonso. Community-driven Extensions to the X3D Volume Rendering Component. In *Proceedings of the 22Nd International Conference on 3D Web Technology*, Web3D '17, pages 1:1–1:9, New York, NY, USA, 2017b. ACM. ISBN 978-1-4503-4955-0. doi: 10.1145/3055624.3075945. URL `http://doi.acm.org/10.1145/3055624.3075945`.

A. Arbelaiz, A. Moreno, L. Kabongo, H. V. Diez, and A. García Alonso. Interactive Visualization of DICOM Volumetric Datasets in the Web - Providing VR Experiences within the Web Browser. In *Proceedings of* *the 12th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 3: IVAPP, (VISIGRAPP 2017)*, pages 108–115. INSTICC, SciTePress, 2017c. ISBN 978-989-758-228-8. doi: 10.5220/0006154801080115.

J. Behr, P. Dähne, Y. Jung, and S. Webel. Beyond the web browser-x3d and immersive vr. In *IEEE Virtual Reality 2007: Symposium on 3D User Interfaces (3DUI)*, volume 2007. Fraunhofer IGD, 2007.

J. Behr, P. Eschler, Y. Jung, and M. Zöllner. X3DOM: A DOM-based HTML5/X3D integration model. In *Proceedings of the 14th International Conference on 3D Web Technology*, Web3D '09, pages 127–135, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-432-4. doi: 10.1145/1559764.1559784. URL `http://doi.acm.org/10.1145/1559764.1559784`.

J. Beyer, M. Hadwiger, and H. Pfister. A survey of GPU-based large-scale volume visualization. 2014.

J. F. Blinn. Models of Light Reflection for Computer Synthesized Pictures. *SIGGRAPH Comput. Graph.*, 11(2):192–198, July 1977. ISSN 0097-8930. doi: 10.1145/965141.563893. URL `http://doi.acm.org/10.1145/965141.563893`.

F. Bösch. WebGL Stats. Statistics of supported WebGL hardware capabilities, 2019. URL `https://webglstats.com`.

M. Botsch, M. Spernat, and L. Kobbelt. Phong Splatting. In *Proceedings of the First Eurographics Conference on Point-Based Graphics*, SPBG'04, pages 25–32, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN 3-905673-09-6. doi: 10.2312/SPBG/SPBG04/025-032. URL `http://dx.doi.org/10.2312/SPBG/SPBG04/025-032`.

M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's GPUs. In *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pages 17–141, June 2005. doi: 10.1109/PBG.2005.194059.

S. Bruckner and M. E. Gröller. VolumeShop: An Interactive System for Direct Volume Illustration. In H. R. C. T. Silva, E. Gröller, editor, *Proceedings of IEEE Visualization 2005*, pages 671–678, oct 2005. ISBN 0780394623. URL `https://www.cg.tuwien.ac.at/research/publications/2005/bruckner-2005-VIS/`.

S. Bruckner and M. E. Gröller. Style Transfer Functions for Illustrative Volume Rendering. *Computer Graphics Forum*, 26(3):715–724, Sept. 2007. doi: 10.1111/j.1467-8659.2007.01095.x. URL `http://dx.doi.org/10.1111/j.1467-8659.2007.01095.x`.

S. Bruckner, S. Grimm, A. Kanitsar, and M. E. Gröller. Illustrative context-preserving volume rendering. In *EuroVis*, pages 69–76, 2005.

R. Cabello. Three.js a JavaScript 3D library, 2018. URL `http://www.threejs.org`.

B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Proceedings of the 1994 Symposium on Volume Visualization*, VVS '94, pages 91–98, New York, NY, USA, 1994. ACM. ISBN 0-89791-741-3. doi: 10.1145/197938.197972. URL `http://doi.acm.org/10.1145/197938.197972`.

S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, May 2005. ISSN 1077-2626. doi: 10.1109/TVCG.2005.46.

D. Catuhe, M. Rousseau, P. Lagarde, and D. Rousset. Babylon.js a 3D engine based on webgl and javascript, 2014. URL `http://www.babylonjs.com`.

J. Chandler, H. Obermaier, and K. I. Joy. WebGL-Enabled Remote Visualization of Smoothed Particle Hydrodynamics Simulations. In E. Bertini, J. Kennedy, and E. Puppo, editors, *Eurographics Conference on Visualization (EuroVis) - Short Papers*. The Eurographics Association, 2015. doi: 10.2312/eurovisshort.20151116.

M. Christie and P. Olivier. Camera Control in Computer Graphics: Models, Techniques and Applications. In *ACM SIGGRAPH ASIA 2009 Courses*, SIGGRAPH ASIA '09, pages 3:1–3:197, New York, NY, USA, 2009. ACM. doi: 10.1145/1665817.1665820. URL `http://doi.acm.org/10.1145/1665817.1665820`.

Chrome. Chrome developer builds with WebVR support, 2016. URL `https://webvr.info/get-chrome/`.

J. Comba, J. T. Klosowsk, N. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams. Fast Polyhedral Cell Sorting for Interactive Rendering of

Unstructured Grids. *Computer Graphics Forum*, 18(3):369–376, 1999. doi: 10.1111/1467-8659.00357. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00357`.

J. Congote. MEDX3DOM: MEDX3D for X3DOM. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 179–179, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1432-9. doi: 10.1145/2338714.2338746. URL `http://doi.acm.org/10.1145/2338714.2338746`.

J. Congote, A. Segura, L. Kabongo, A. Moreno, J. Posada, and 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, pages 137–146, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0774-1. doi: 10.1145/2010425.2010449. URL `http://doi.acm.org/10.1145/2010425.2010449`.

J. Congote, A. Moreno, L. Kabongo, Pérez, J.-L., San-José, R., and O. Ruiz. Web based hybrid volumetric visualisation of urban gis data, 2012. URL `https://doi.org/10.1051/3u3d/201203001`.

Cornerstone. JavaScript library to display interactive medical images including but not limited to DICOM, 2016. URL `https://github.com/chafey/cornerstone`.

C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22. ACM, 2009.

M. F. A. P. Dani Tost, Sergi Grau. Ray-casting time-varying volume data sets with frame-to-frame coherence, 2006. URL `https://doi.org/10.1117/12.649814`.

P. Decaudin. Cartoon Looking Rendering of 3D Scenes. Research Report 2919, INRIA, June 1996. URL `http://phildec.users.sf.net/Research/RR-2919.php`.

J. Díaz-García, P. Brunet, I. Navazo, and P.-P. Vázquez. Progressive ray casting for volumetric models on mobile devices. *Computers & Graphics*, 73:1 – 16, 2018. ISSN 0097-8493. doi: https://doi.org/10.1016/j.cag.2018.02.007. URL `http://www.sciencedirect.com/science/article/pii/S009784931830030X`.

D. Ebert and P. Rheingans. Volume Illustration: Non-photorealistic Rendering of Volume Models. In *Proceedings of the Conference on Visualization '00*, VIS '00, pages 195–202, Los Alamitos, CA, USA, 2000. IEEE Computer Society Press. ISBN 1-58113-309-X. URL `http://dl.acm.org/citation.cfm?id=375213.375241`.

T. T. Elvins. A Survey of Algorithms for Volume Visualization. *SIGGRAPH Comput. Graph.*, 26(3):194–201, Aug. 1992. ISSN 0097-8930. doi: 10.1145/142413.142427. URL `http://doi.acm.org/10.1145/142413.142427`.

K. Engel, M. Kraus, and 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, pages 9–16, New York, NY, USA, 2001. ACM. ISBN 1-58113-407-X. doi: 10.1145/383507.383515. URL `http://doi.acm.org/10.1145/383507.383515`.

J. Fernandez-Bayó, O. Barbero, C. Rubies, M. Sentís, and L. Donoso. Distributing Medical Images with Internet Technologies: A DICOM Web Server and a DICOM Java Viewer. *RadioGraphics*, 20(2):581–590, 2000. doi: 10.1148/radiographics.20.2.g00mc18581. URL `http://dx.doi.org/10.1148/radiographics.20.2.g00mc18581`. PMID: 10715352.

Fraunhofer IGD. X3DOM, 2014. URL `http://www.x3dom.org`.

E. Gobbetti, F. Marton, and J. A. I. Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

A. Gooch, B. Gooch, P. Shirley, and E. Cohen. A Non-photorealistic Lighting Model for Automatic Technical Illustration. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, pages 447–452, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280950. URL `http://doi.acm.org/10.1145/280814.280950`.

Google. The polymer project, 2018. URL `https://www.polymer-project.org/`.

A. Guetat, A. Ancel, S. Marchesin, and J. Dischler. Pre-Integrated Volume Rendering with Non-Linear Gradient Interpolation. *IEEE Trans-*

actions on Visualization and Computer Graphics, 16(6):1487–1494, Nov 2010. ISSN 1077-2626. doi: 10.1109/TVCG.2010.187.

I. Gutenko, K. Petkov, C. Papadopoulos, X. Zhao, J. H. Park, A. Kaufman, and R. Cha. Remote volume rendering pipeline for mHealth applications. In *SPIE Medical Imaging*, pages 903904–903904. International Society for Optics and Photonics, 2014.

M. Hadwiger, J. M. Kniss, C. Rezk-salama, D. Weiskopf, and K. Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. ISBN 1568812663.

D. Hähn, N. Rannou, B. Ahtam, E. Grant, and R. Pienaar. Neuroimaging in the Browser using the X Toolkit. In *Front. Neuroinform. Conference Abstract: 5th INCF Congress of Neuroinformatics. doi: 10.3389/conf. fninf*, Neuroinformatics, 2012.

Q. Ho and M. Jern. Interacting with 4D oceanographic volume data using GeoAnalytics tools. *National Center for Visual Analytics NCVA*, 2008.

M. Hofmann. Virtual Natural History Museum (VNHM) website, 2018. URL `http://vnhm.de/`.

C. Hurt, N. F. Polys, and H. A. Burgess. Zebrafish Brain Browser, 2018. URL `http://metagrid2.sv.vt.edu/~chris526/zbb/`.

N. John, M. Aratow, J. Couch, D. Evestedt, A. Hudson, N. Polys, R. Puk, A. Ray, K. Victor, and Q. Wang. MedX3D: standards enabled desktop medical 3D. *Studies in health technology and informatics*, 132:189–194, 2007.

H. J. Johnson, M. McCormick, L. Ibáñez, and T. I. S. Consortium. *The ITK Software Guide*. Kitware, Inc., third edition, 2013. URL `http://www.itk.org/ItkSoftwareGuide.pdf`.

D. Jönsson, E. Sundén, A. Ynnerman, and T. Ropinski. A Survey of Volumetric Illumination Techniques for Interactive Volume Rendering. *Computer Graphics Forum*, 33(1):27–51, 2014. doi: 10.1111/cgf.12252. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.12252`.

Y. Jung, R. Recker, M. Olbrich, and U. Bockholt. Using X3D for Medical Training Simulations. In *Proceedings of the 13th International Symposium on 3D Web Technology*, Web3D '08, pages 43–51, New York, NY, USA,

2008. ACM. ISBN 978-1-60558-213-9. doi: 10.1145/1394209.1394221. URL `http://doi.acm.org/10.1145/1394209.1394221`.

L. Kabongo and A. Arbelaiz. Mirro4all application, 2018a. URL `http://mirror4all.vicomtech.org`.

L. Kabongo and A. Arbelaiz. Mirro4all open source repository, 2018b. URL `https://github.com/VolumeRC/MIRROR4all`.

J. T. Kajiya and B. P. Von Herzen. Ray Tracing Volume Densities. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 165–174, New York, NY, USA, 1984a. ACM. ISBN 0-89791-138-5. doi: 10.1145/800031.808594. URL `http://doi.acm.org/10.1145/800031.808594`.

J. T. Kajiya and B. P. Von Herzen. Ray Tracing Volume Densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, Jan. 1984b. ISSN 0097-8930. doi: 10.1145/964965.808594. URL `http://doi.acm.org/10.1145/964965.808594`.

M. Kaspar, N. M. Parsad, and J. C. Silverstein. An optimized web-based approach for collaborative stereoscopic medical visualization. *Journal of the American Medical Informatics Association*, 20(3):535–543, 2013.

T. L. Kay and J. T. Kajiya. Ray Tracing Complex Scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278, Aug. 1986a. ISSN 0097-8930. doi: 10.1145/15886.15916. URL `http://doi.acm.org/10.1145/15886.15916`.

T. L. Kay and J. T. Kajiya. Ray Tracing Complex Scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 269–278, New York, NY, USA, 1986b. ACM. ISBN 0-89791-196-2. doi: 10.1145/15922.15916. URL `http://doi.acm.org/10.1145/15922.15916`.

Khronos. WebGL: OpenGL ES 2.0 for the Web., 2016. URL `https://www.khronos.org/webgl/`.

Kitware. VTK.js JavaScript library for scientific visualization, 2019. URL `https://kitware.github.io/vtk-js/index.html`.

J. Kniss, G. Kindlmann, and C. Hansen. Multidimensional Transfer Functions for Interactive Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, July 2002. ISSN 1077-2626.

doi: 10.1109/TVCG.2002.1021579. URL http://dx.doi.org/10.1109/TVCG.2002.1021579.

J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003a. IEEE Computer Society. ISBN 0-7695-2030-8. doi: 10.1109/VIS.2003.10001. URL https://doi.org/10.1109/VIS.2003.10001.

J. Kruger and R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 38–, Washington, DC, USA, 2003b. IEEE Computer Society. ISBN 0-7695-2030-8. doi: 10.1109/VIS.2003.10001. URL http://dx.doi.org/10.1109/VIS.2003.10001.

H. Kye, B.-S. Shin, and Y. G. Shin. Interactive classification for pre-integrated volume rendering of high-precision volume data. *Graphical Models*, 70(6):125 – 132, 2008. ISSN 1524-0703. doi: https://doi.org/10.1016/j.gmod.2008.05.001. URL http://www.sciencedirect.com/science/article/pii/S1524070308000088.

P. Lacroute. Real-time Volume Rendering on Shared Memory Multiprocessors Using the Shear-warp Factorization. In *Proceedings of the IEEE Symposium on Parallel Rendering*, PRS '95, pages 15–22, New York, NY, USA, 1995. ACM. ISBN 0-89791-774-X. doi: 10.1145/218327.218331. URL http://doi.acm.org/10.1145/218327.218331.

P. Lacroute and 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, pages 451–458, New York, NY, USA, 1994. ACM. ISBN 0-89791-667-0. doi: 10.1145/192161.192283. URL http://doi.acm.org/10.1145/192161.192283.

D. Laur and P. Hanrahan. Hierarchical Splatting: A Progressive Refinement Algorithm for Volume Rendering. *SIGGRAPH Comput. Graph.*, 25(4): 285–288, July 1991. ISSN 0097-8930. doi: 10.1145/127719.122748. URL http://doi.acm.org/10.1145/127719.122748.

v. Lesar, C. Bohak, and M. Marolt. Real-time Interactive Platform-agnostic Volumetric Path Tracing in webGL 2.0. In *Proceedings of the 23rd International ACM Conference on 3D Web Technology*, Web3D '18, pages

7:1–7:7, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5800-2. doi: 10.1145/3208806.3208814. URL `http://doi.acm.org/10.1145/3208806.3208814`.

M. Levoy. Display of Surfaces from Volume Data. *IEEE Comput. Graph. Appl.*, 8(3):29–37, May 1988. ISSN 0272-1716. doi: 10.1109/38.511. URL `http://dx.doi.org/10.1109/38.511`.

M. Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3): 245–261, 1990. doi: 10.1145/78964.78965. URL `https://doi.org/10.1145/78964.78965`.

T. Li, M. Xie, W. Zhao, and Y. Wei. Shear-warp rendering algorithm based on radial basis functions interpolation. In *2010 Second International Conference on Computer Modeling and Simulation*, volume 2, pages 425–429. IEEE, 2010.

W. Li, K. Mueller, and A. Kaufman. Empty Space Skipping and Occlusion Clipping for Texture-based Volume Rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, VIS '03, pages 42–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2030-8. doi: 10.1109/VISUAL.2003.1250388. URL `http://dx.doi.org/10.1109/VISUAL.2003.1250388`.

L. Linsen, H. Hagen, B. Hamann, and H. Hege. *Visualization in Medicine and Life Sciences II: Progress and New Challenges*. Mathematics and Visualization. Springer, 2012. ISBN 9783642216084. URL `http://link.springer.com/book/10.1007%2F978-3-642-21608-4`.

P. Ljung, J. Krüger, M. E. Gröller, M. Hadwiger, C. Hansen, and A. Ynnerman. State of the Art in Transfer Functions for Direct Volume Rendering. *Computer Graphics Forum (2016)*, 35(3):669–691, 2016. URL `https://www.cg.tuwien.ac.at/research/publications/2016/Groeller_2016_P3/`.

E. B. Lum and K.-L. Ma. Hardware-accelerated parallel non-photorealistic volume rendering. In *Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering*, pages 67–ff. ACM, 2002.

N. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, 1995.

A. Maximo, R. Marroquim, and R. Farias. Hardware-Assisted Projected Tetrahedra. *Computer Graphics Forum*, 29(3):903–912, 2010. doi: 10.1111/j.1467-8659.2009.01673.x. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1467-8659.2009.01673.x`.

M. McCann, R. Puk, A. Hudson, R. Melton, and D. Brutzman. Proposed Enhancements to the X3D Geospatial Component. In D. W. Fellner, A. Sourin, J. Behr, and K. Walczak, editors, *International Conference on 3D Web Technology*. The Eurographics Association, 2009. ISBN 978-1-60558-432-4. doi: 10.1145/1559764.1559788.

M. Meißner, U. Hoffmann, and W. Straßer. Enabling Calssification and Shading for 3D Texture Mapping Based Volume Rendering Using OpenGL and Extensions. In *IEEE Visualization*, 1999.

M. Meißner, S. Guthe, and W. Straßer. Interactive lighting models and pre-integration for volume rendering on PC graphics accelerators. In *Graphics Interface*, volume 2, pages 209–218, 2002.

J. Mischek. Clipping with caps, 2018. URL `https://github.com/daign/clipping-with-caps`.

M. M. Mobeen and L. Feng. Ubiquitous medical volume rendering on mobile devices. In *International Conference on Information Society (i-Society 2012)*, pages 93–98, June 2012a.

M. M. Mobeen and L. Feng. High-Performance Volume Rendering on the Ubiquitous WebGL Platform. In *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 381–388, Washington, DC, USA, June 2012b. IEEE Computer Society. ISBN 978-0-7695-4749-7. doi: 10.1109/HPCC.2012.58. URL `http://dx.doi.org/10.1109/HPCC.2012.58`.

A. Moreno, A. G. A. Mujika, and A. Segura. Visual analytics of multi-sensor weather information georeferenciation of Doppler weather radar and weather stations. In *2014 International Conference on Information Visualization Theory and Applications (IVAPP)*, pages 329–336, Jan 2014.

M. M. Movania and F. Lin. Mobile visualization of biomedical volume datasets. *J. Internet. Technol. Secured Trans*, 1(2):52–60, 2012.

M. M. Movania, W. M. Chiew, and F. Lin. On-Site Volume Rendering with GPU-Enabled Devices. *Wireless Personal Communications*, 76(4): 795–812, 2014. ISSN 1572-834X. doi: 10.1007/s11277-013-1354-y. URL `http://dx.doi.org/10.1007/s11277-013-1354-y`.

F. Mwalongo, M. Krone, G. Reina, and T. Ertl. Web-based Volume Rendering using Progressive Importance-based Data Transfer. In F. Beck, C. Dachsbacher, and F. Sadlo, editors, *Vision, Modeling and Visualization*. The Eurographics Association, 2018. ISBN 978-3-03868-072-7. doi: 10.2312/vmv.20181264.

N. Neophytou and K. Mueller. GPU accelerated image aligned splatting. In *Fourth International Workshop on Volume Graphics, 2005.*, pages 197–242, June 2005. doi: 10.1109/VG.2005.194115.

J. M. Noguera and J.-R. Jiménez. Visualization of very large 3D volumes on mobile devices and WebGL. *WSCG Communication Proceedings*, pages 105–112, 2012.

J. M. Noguera and J. R. Jiménez. Mobile Volume Rendering: Past, Present and Future. *IEEE Transactions on Visualization and Computer Graphics*, 22(2):1164–1178, Feb 2016. ISSN 1077-2626. doi: 10.1109/TVCG.2015.2430343.

J. M. Noguera, J.-R. Jiménez, C. J. Ogáyar, and R. J. Segura. Volume Rendering Strategies on Mobile Devices. In *GRAPP/IVAPP*, pages 447–452, 2012.

OsiriX. OsiriX DICOM Image Library, 2018. URL `http://www.osirix-viewer.com/resources/dicom-image-library/`.

T. Pan, T.-Y. Lee, E. Rietzel, and G. T. Y. Chen. 4D-CT imaging of a volume influenced by respiratory motion on multi-slice CT. *Medical Physics*, 31(2):333–340, 2004. doi: 10.1118/1.1639993. URL `https://aapm.onlinelibrary.wiley.com/doi/abs/10.1118/1.1639993`.

S. Perandini, N. Faccioli, A. Zaccarella, T. Re, and R. Mucelli. The diagnostic contribution of CT volumetric rendering techniques in routine practice. *The Indian Journal of Radiology & Imaging*, 20(2):92–97, 2010. doi: 10.4103/0971-3026.63043.

C. Pinson. OSG.JS WebGL framework based on OpenSceneGraph concepts, 2014. URL `http://www.osgjs.org`.

N. Polys and A. Wood. New platforms for health hypermedia. *Issues in Information Systems*, 13(1):40–50, 2012.

N. Polys, A. Wood, and P. Shinpaugh. Cross-platform presentation of interactive volumetric imagery. Technical report, Departmental Technical Report 1177, Virginia Tech, Advanced Research Computing, 2011a.

N. Polys, A. D. Wood, and P. Shinpaugh. Cross-platform Presentation of Interactive Volumetric Imagery. 2011b.

N. F. Polys, S. Ullrich, D. Evestedt, A. D. Wood, and M. Aratow. A fresh look at immersive Volume Rendering: Challenges and capabilities. In *IEEE VR Workshop on Immersive Volume Rendering, Orlando*, 2013a.

N. F. Polys, S. Ullrich, D. Evestedt, A. D. Wood, and M. Aratow. A Fresh Look at Immersive Volume Rendering: Challenges and Capabilities. *IEEE VR Workshop on Immersive Volume Rendering 2013: Orlando.*, 2013b.

L. Rebenitsch. Managing Cybersickness in Virtual Reality. *XRDS*, 22(1): 46–51, Nov. 2015. ISSN 1528-4972. doi: 10.1145/2810054. URL `http://doi.acm.org/10.1145/2810054`.

L. Ren, H. Pfister, and M. Zwicker. Object Space EWA Surface Splatting: A Hardware Accelerated Approach to High Quality Point Rendering. *Computer Graphics Forum*, 21(3):461–470, 2002. doi: 10.1111/1467-8659.00606. URL `https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-8659.00606`.

S. Roettger and T. Ertl. A two-step approach for interactive pre-integrated volume rendering of unstructured grids. In *Symposium on Volume Visualization and Graphics, 2002. Proceedings. IEEE / ACM SIGGRAPH*, pages 23–28, Oct 2002. doi: 10.1109/SWG.2002.1226506.

S. Rottger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings Visualization 2000. VIS 2000 (Cat. No.00CH37145)*, pages 109–116, Oct 2000. doi: 10.1109/VISUAL.2000.885683.

J. P. Schulze, M. Kraus, U. Lang, and T. Ertl. Integrating pre-integration into the shear-warp algorithm. In *Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics*, pages 109–118. ACM, 2003.

P. Shirley and A. Tuchman. A Polygonal Approximation to Direct Scalar Volume Rendering. *SIGGRAPH Comput. Graph.*, 24(5):63–70, Nov. 1990. ISSN 0097-8930. doi: 10.1145/99308.99322. URL `http://doi.acm.org/10.1145/99308.99322`.

C. T. Silva, J. S. B. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *IEEE Symposium on Volume Visualization (Cat. No.989EX300)*, pages 87–94, Oct 1998. doi: 10.1109/SVV.1998.729589.

C. T. Silva, J. L. D. Comba, S. P. Callahan, and F. F. Bernardon. A survey of GPU-based volume rendering of unstructured grids. *Revista de informática teórica e aplicada. Porto Alegre, RS. Vol. 12, n. 2 (out. 2005), p. 9-29*, 2005.

Z. Stankovic, B. D. Allen, J. Garcia, K. B. Jarvis, and M. Markl. 4D flow imaging with MRI. *Cardiovascular Diagnosis and Therapy*, 4(2): 173–192, 04 2014. doi: 10.3978/j.issn.2223-3652.2014.01.02. URL `http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3996243/`.

S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-hardware-based Raycasting. In *Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics*, VG'05, pages 187–195, Aire-la-Ville, Switzerland, Switzerland, 2005. Eurographics Association. ISBN 3-905673-26-6. doi: 10.2312/VG/VG05/187-195. URL `http://dx.doi.org/10.2312/VG/VG05/187-195`.

J. Sweeney and K. Mueller. Shear-warp deluxe: the shear-warp algorithm revisited. In *VisSym*, pages 95–104, 2002.

K. M. Tabor, G. D. Marquart, C. Hurt, T. S. Smith, A. K. Geoca, A. A. Bhandiwad, A. Subedi, J. L. Sinclair, N. F. Polys, and H. A. Burgess. Brain-wide cellular resolution imaging of Cre transgenic zebrafish lines for functional circuit-mapping. *bioRxiv*, 2018. doi: 10.1101/444075. URL `https://www.biorxiv.org/content/early/2018/10/15/444075`.

H. K. Tuy and L. T. Tuy. Direct 2-D display of 3-D objects. *IEEE Computer Graphics and Applications*, 4(10):29–34, Oct 1984. ISSN 0272-1716. doi: 10.1109/MCG.1984.6429333.

University of Tübingen WSI/GRIS. Collection of volumetric datasets, 2014. URL `http://www.volvis.org`.

C. Upson and M. Keeler. V-buffer: visible volume rendering. In *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1988, Atlanta, Georgia, USA, August 1-5, 1988*, pages 59–64, 1988. doi: 10.1145/54852.378482. URL `https://doi.org/10.1145/54852.378482`.

R. Vlasov, K.-I. Friese, and F.-E. Wolter. Haptic rendering of volume data with collision determination guarantee using ray casting and implicit surface representation. In *Cyberworlds (CW), 2012 International Conference on*, pages 91–98. IEEE, 2012.

Volvis. Volumetric dataset archive, 2017. URL `http://volvis.org`.

J. Wallis, T. R. Miller, C. Lerner, and E. Kleerup. Three-dimensional display in nuclear medicine. *Medical Imaging, IEEE Transactions on*, 8(4):297–230, Dec 1989. ISSN 0278-0062. doi: 10.1109/42.41482.

K. Wangkaoom, P. Ratanaworabhan, and S. S. Thongvigitmanee. High-quality web-based volume rendering in real-time. In *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, pages 1–6, June 2015. doi: 10.1109/ECTICon.2015.7207091.

Web3DConsortium. Extensible 3D (X3D) basic example archives, 2014. URL `http://www.web3d.org/x3d-resources/content/examples/Basic/VolumeRendering/`.

Web3DConsortium. Extensible 3D (X3D) Volume rendering component: ISO/IEC 19775-1, 2017a. URL `http://www.web3d.org/files/specifications/19775-1/V3.3/Part01/components/volume.html`.

Web3DConsortium. Extensible 3D (X3D) v3.2, 2017b. URL `http://www.web3d.org/standards/version/V3.2`.

Web3DConsortium. Extensible 3D (X3D) specifications, 2014. URL `http://www.web3d.org/x3d/specifications/`.

Web3DConsortium. Web3DConsortium Medical Working Group (MWG), 2017a. URL `http://www.web3d.org/working-groups/medical`.

Web3DConsortium. Extensible 3D (X3D) specifications, 2017b. URL `http://www.web3d.org/x3d/specifications/`.

WebVR. Bringing virtual reality to the web, 2016. URL http://webvr.info/.

M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based view-independent cell projection. *IEEE Transactions on Visualization and Computer Graphics*, 9(2):163–175, April 2003. ISSN 1077-2626. doi: 10.1109/TVCG.2003.1196004.

D. Weiskopf. *GPU-based interactive visualization techniques*. Springer, 2007.

R. Westermann and 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, pages 169–177, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: 10.1145/280814.280860. URL http://doi.acm.org/10.1145/280814.280860.

L. Westover. Footprint Evaluation for Volume Rendering. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 367–376, New York, NY, USA, 1990a. ACM. ISBN 0-89791-344-2. doi: 10.1145/97879.97919. URL http://doi.acm.org/10.1145/97879.97919.

L. Westover. Footprint Evaluation for Volume Rendering. *SIGGRAPH Comput. Graph.*, 24(4):367–376, Sept. 1990b. ISSN 0097-8930. doi: 10.1145/97880.97919. URL http://doi.acm.org/10.1145/97880.97919.

P. L. Williams. Visibility-ordering Meshed Polyhedra. *ACM Trans. Graph.*, 11(2):103–126, Apr. 1992. ISSN 0730-0301. doi: 10.1145/130826.130899. URL http://doi.acm.org/10.1145/130826.130899.

B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral Projection Using Vertex Shaders. In *Proceedings of the 2002 IEEE Symposium on Volume Visualization and Graphics*, VVS '02, pages 7–12, Piscataway, NJ, USA, 2002. IEEE Press. ISBN 0-7803-7641-2. URL http://dl.acm.org/citation.cfm?id=584110.584112.

X3DOM Community. onehalfmv2 - Moving inside a volume, 2015a. URL https://github.com/x3dom/x3dom/issues/537.

X3DOM Community. Johannes Schröder-Schetelig - Rendering of volumetric and polygonal data together, 2015b. URL https://sourceforge.net/p/x3dom/mailman/x3dom-users/thread/

CAC7R8D5gFBTeKMk36Pb0ayY_g0qE0hHpok2ioO5C339mr9Akdg@mail.gmail.com.

X3DOM Community. Paul - MPR multiple arbitrary planes, 2016a. URL https://sourceforge.net/p/x3dom/mailman/x3dom-users/thread/d9cd0469-497f-03ac-fe72-b6909b2a9b7f%40web.de.

X3DOM Community. pgruenbacher-TSUS - MPR not include tranfer function, 2016b. URL https://github.com/x3dom/x3dom/issues/613.

X3DOM Community. PCH3DPrintLab - Section Caps for Clipping Planes, 2017. URL https://github.com/x3dom/x3dom/issues/718.

R. Xu, A. Sugiyama, K. Hasegawa, K. Tagawa, S. Tanaka, and H. T. Tanaka. *Innovation in Medicine and Healthcare 2015*, chapter Remote Transparent Visualization of Surface-Volume Fused Data to Support Network-Based Laparoscopic Surgery Simulation, pages 345–352. Springer International Publishing, Cham, 2016. ISBN 978-3-319-23024-5. doi: 10.1007/978-3-319-23024-5_31. URL http://dx.doi.org/10.1007/978-3-319-23024-5_31.

F. Yang, F. Yang, X. Li, and J. Tian. Ray feature analysis for volume rendering. *Multimedia Tools and Applications*, pages 1–21, 2014. ISSN 1380-7501. doi: 10.1007/s11042-014-1994-2. URL http://dx.doi.org/10.1007/s11042-014-1994-2.

Y. Yang, A. Sharma, and A. Girier. Volumetric Texture Data Compression Scheme for Transmission. In *Proceedings of the 20th International Conference on 3D Web Technology*, Web3D '15, pages 65–68, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3647-5. doi: 10.1145/2775292.2775323. URL http://doi.acm.org/10.1145/2775292.2775323.

Y. Yang, A. Sharma, and A. Girier. X3DOM RadarVolumeStyle example demo, 2018. URL https://examples.x3dom.org/example/RadarVolumeStyle/.

J. Zhou, , J. Zhou, and K. D. Tönnies. State of The Art for Volume Rendering, 2018.

Z. Zhou, Y. Tao, H. Lin, F. Dong, and G. Clapworthy. Occlusion-free Feature Exploration for Volume Visualization. *Multimedia Tools Appl.*, 74(23):10243–10258, Dec. 2015. ISSN 1380-7501. doi: 10.1007/s11042-014-2162-4. URL http://dx.doi.org/10.1007/s11042-014-2162-4.

M. Zwicker, H. Pfister, J. van Baar, and M. Gross. EWA splatting. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):223–238, July 2002. ISSN 1077-2626. doi: 10.1109/TVCG.2002.1021576.