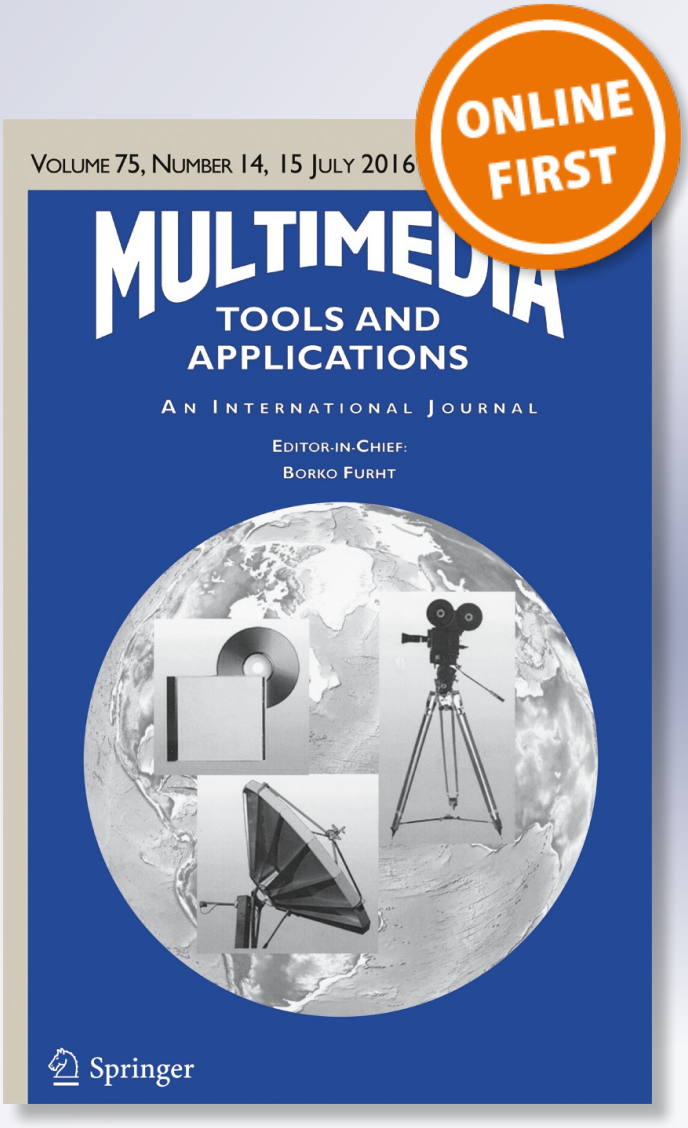
X3DOM volume rendering component for web content developers

Ander Arbelaiz, Aitor Moreno, Luis Kabongo & Alejandro García-Alonso

Multimedia Tools and Applications
An International Journal

ISSN 1380-7501

Multimed Tools Appl
DOI 10.1007/s11042-016-3743-1



Springer

Your article is protected by copyright and all rights are held exclusively by Springer Science +Business Media New York. This e-offprint is for personal use only and shall not be self-archived in electronic repositories. If you wish to self-archive your article, please use the accepted manuscript version for posting on your own website. You may further deposit the accepted manuscript version in any repository, provided it is only made publicly available 12 months after official publication or later and provided acknowledgement is given to the original source of publication and a link is inserted to the published article on Springer's website. The link must be accompanied by the following text: "The final publication is available at link.springer.com".

Author's personal copy

Multimed Tools Appl
DOI 10.1007/s11042-016-3743-1

X3DOM volume rendering component for web content developers

Ander Arbelaiz[1] · **Aitor Moreno**[1] · **Luis Kabongo**[1,2] · **Alejandro García-Alonso**[3]

Received: 2 December 2014 / Revised: 4 April 2016 / Accepted: 30 June 2016
© Springer Science+Business Media New York 2016

Abstract We present a real-time volume rendering component for the Web, which provides a set of illustrative and non-photorealistic styles. Volume data is used in many scientific disciplines, requiring the visualization of the inner data, features for enhancing extracted characteristics or even coloring the volume. The Medical Working Group of X3D published a volume rendering specification. The next step is to build a component that realizes the functionalities defined by the specification. We have designed and built a volume rendering component integrated in the X3DOM framework. This component allows content developers to use the X3D specification. It combines and applies multiple rendering styles to several volume data types, offering a suitable tool for declarative volume rendering on the Web. As we show in the result section, the proposed component can be used in many fields that requires the visualization of multi-dimensional data, such as in medical and scientific fields. Our approach is based on WebGL and X3DOM, providing content developers with an easy and flexible declarative way of sharing and visualizing volumetric content over the Web.

Keywords Volume rendering · WebGL · Declarative 3D · X3DOM · X3D

✉ Ander Arbelaiz
aarbelaiz@vicomtech.org

Aitor Moreno
amoreno@vicomtech.org

Luis Kabongo
lkabongo@vicomtech.org

Alejandro García-Alonso
alex.galonso@ehu.es

[1] Vicomtech-IK4, 20009 Donostia / San Sebastián, Spain

[2] Biodonostia Health Research Institute, Donostia / San Sebastián, Spain

[3] University of the Basque Country, Donostia / San Sebastián, Spain

Published online: 15 July 2016

Author's personal copy

1 Introduction

Scientific visualization aims to offer a better understanding of complex data like multi-dimensional data. This goal requires specialized rendering techniques to visualize data from many scientific areas and topics, such as biomolecular systems in biology, particle collisions in quantum physics, fluid flow in physics and geographic information in geoinformatics.

When it comes to 3-dimensional discrete sampled data, direct volume rendering visualization techniques can be used (see Fig. 1). Volumetric data often needs to be processed in order to enhance and extract specific information or characteristics within the data. In the medical field, the visualization of enhanced features eases the identification and differentiation of objects such as tumorous tissues, functional activities, organs and morphological characteristics. We provide users with operations to search specific information imperceptible during the raw data exploration. Therefore, in terms of the user's interactions, interactive volume rendering techniques can be compared to the traditional interactive tools in Visual Analytics [41]. These actions are goal-oriented and the user's knowledge of the domain is what drives the discovery process.

The Web is the greatest platform for knowledge and content distribution. In this environment, new software distribution paradigms have been introduced. Ideally, a web application will be universally accessible by its web address, and it will run in any computer or device. This is the opposite to the Desktop environment, where a program is meant to run in the specific computer it is installed on.

Current web applications lack the capabilities for rendering volumetric datasets interactively in compliance with the Extensible 3D (X3D) standard [40]. The Medical Working Group of X3D published a volume rendering component specification. Following these definitions, content developers may easily define a distributable and replicable 3D volume rendering scene in a declarative way.

Our objective is to provide interactive rendering of volumetric datasets in the Web, which places two main challenges: In first place, all the rendering requirements described by the X3D specification should be fulfilled. In second place, the challenge should be solved using only standardized web-based tools: HTML, JavaScript and WebGL. Our approach adapts the traditional volume rendering techniques [20, 36] in the Web environment. We have solved the challenges placed by X3D when they defined the nodes that should make possible to interactively display volumetric datasets.

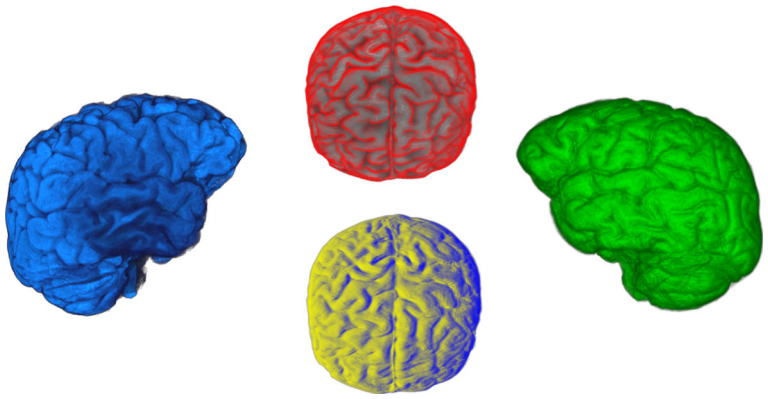


Fig. 1 Volume rendering outputs of the `brain` [39] dataset with the proposed volume rendering component

Author's personal copy

The presented component solves the challenges placed by web technologies for data segmentation, volume blending, visual enhancements on edges, boundaries and silhouettes. Web content developers can take advantage of our software solution that allows them to avoid programing complex computer graphics specialized techniques: creating a volume rendering canvas can be as easy as using X3D and HTML. With a declarative approach, the graphics rendering process is transparent to content creators and developers.

The paper is structured as follows. Section 2 briefly introduces the status of the different volume rendering algorithms and non-photorealistic styles. Section 3 gives an overall picture of the presented work. Section 4 presents the software component and algorithms we have developed for X3DOM, including the necessary adaptations of the volume rendering methods to the Web environment. Section 5 presents our implementation of the X3D nodes. Section 6 shows the results obtained with the implemented rendering styles. Finally, Section 7 summarizes our work with the conclusions and future work.

2 Related work

This section presents the related work. Firstly, it introduces the necessary context to understand where our proposed component is positioned. Secondly, we review the literature that provides the background required to solve the challenges of the presented component.

2.1 Context

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* [4], *Babylon.js* [5] and *OSG.js* [31]. However, they are not focused on the rendering of volumetric data.

Previously, there were two common approaches for volume rendering in the Web browser. The first one makes use of third party plug-ins and the second one, renders the volumetric data on the server side. The first method follows the same approach as desktop solutions using OpenGL or DirectX APIs, but it has been discarded over time due to browsers sand-boxing policies of third-party software and applications for security reasons. The second method is an effective way of rendering big volumetric datasets using high-performance servers. Possibly, this approach is not always suitable for interactive applications, due to the connection lag between client and server. Also, it is not a scalable solution and requires more investment on server side computational power.

In 2012, WebGL-based approaches like *goXTK* [17] and X3DOM [12] provide new tools to address these problems. They benefit from the use of Graphics Processing Units (GPU) on the client side and cross-platform support, including support for mobile devices.

X3D is a royalty-free and matured ISO standard [40]. Conceived for interchangeable 3D content on the Web, it aims at representing a 3D real-time scene with a standard eXtensible Markup Language (XML) based file format. X3DOM [1] is a document object model based tool that allows the integration of the X3D nodes into the HTML DOM tree. It adds the capability of declaring 3D scenes under the X3D format and directly modifying the X3D tree through DOM events.

X3D defines several profiles. Each profile is composed by a set of components. Some of these components are extensions added by collaborative committees. The Medical Working

Author's personal copy

Group of X3D defined the volume rendering component [32]. However, current literature does not contain an implementation of the volume rendering nodes for the Web.

2.2 Background

In the state of the art, several volume rendering algorithms can be found. Indirect methods try to extract the surface data in a pre-processing step, and then, the surfaces are rendered. In contrast, direct methods generate a 2D image directly from the volume data. Our approach is based on a well-known direct method: volume ray-casting. It was presented by Kajiya and Von Herzen [18] and formalized by Levoy [21]. Rays are cast from the viewer position through the volume data and they are sampled at regular intervals along these rays. Each sampled point along the same ray is blended by accumulating color and opacity which makes it computationally expensive. It became more popular when Kruger and Westermann [20] used the graphics hardware computational power and presented a GPU-based ray-casting algorithm, achieving real-time frame rates with nowadays consumer hardware. Several techniques have been addressed to gain performance with ray-casting, such as early ray termination [20], which finishes the accumulation process when the contribution of the sample is irrelevant, and empty space skipping [22], which optimizes the ray traversal through empty regions. In general, 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 was denoted by Stegmaier et al. [36] when they presented a single-pass volume rendering framework for GPU-based ray-casting.

Web volumetric visualization has been researched from two points of view. Rendering the volume on the server side (Gutenko et al. [16]) and rendering the volume on the client side with WebGL. In first instance, Congote et al. [7] presented a WebGL volume ray-casting algorithm based on Kruger and Westermann's multi-pass approach. Later, Mobeen et al. [26, 27] revisited the algorithm presenting a volume rendering WebGL platform based on Stegmaier et al. [36] single-pass approach. Also, with WebGL's ubiquitous characteristic, Noguera et al. [28, 29] have analyzed volume rendering on mobile devices, comparing ray-casting with a texture slicing rendering technique. Currently, among the 3D graphics frameworks available for the Web, only *goXTK* [17] and *X3DOM* [6] support volume rendering for scientific data visualization.

Volume visualization can be enhanced by visual effects or feature extraction [42, 43]. Originally conceived for traditional image rendering and artistic effects, illustrative and non-photorealistic renderings can be adapted for volume rendering. They enhance the feature perception within the volume data. Decaudin [10] introduced cartoon style rendering for 3D scenes and Gooch et al. [15] 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 [11]. Cluster-based, GPU hardware accelerated, non-photorealistic renderings were studied by Lum and Ma [25]. More recently, Bruckner and Gröller [3] presented a novel technique to apply an illustrative style with the use of transfer functions.

Our proposal extends the WebGL-based volume rendering work by Congote et al. [7] and Mobeen and Feng [26] to support the rendering styles specified in the volume rendering component of the X3D standard [40].

Regarding the standard, Polys et al. [33] described and evaluated the usability and feasibility of their volume rendering component implementation. Their tests were focused on the desktop visualization of several datasets acquired from different fields. Their results demonstrated how the X3D specification meets the requirements of repeatable multi-dimensional

Author's personal copy

Table 1 Summary of the related work in categories: The volume rendering algorithm (Texture slicer, Ray-casting and Single-pass ray-casting), hardware acceleration (GPU or WebGL), supported platform (Desktop and Mobile) and multiple rendering styles (Styles)

References	Text. slicer	Ray-casting	Single-pass	GPU	WebGL	Desktop	Mobile	Styles
[18, 21]	x	✓	x	x	x	✓	x	x
[20, 22]	x	✓	x	✓	x	✓	x	x
[36]	x	✓	✓	✓	x	✓	x	✓
[16]	x	✓	x	✓	x	✓	✓	?
[6, 7]	x	✓	x	✓	✓	✓	✓	x
[26, 27]	✓	✓	✓	✓	✓	✓	✓	x
[28, 29]	✓	✓	x	✓	✓	✓	✓	x
[17]	✓	x	x	✓	✓	✓	?	x
[3, 25, 33, 34, 42, 43]	x	✓	x	✓	x	✓	x	✓

volume image presentation across domains. Additionally, Polys et al. [34] described to what extend the X3D standard meets the requirements for immersive volume rendering and some examples were provided. Our proposal implements the X3D standard volume rendering component for the X3DOM framework, following some guidelines and suggestions from their previous works. Table 1 summarizes the presented bibliographic works, providing the relevant supported features.

3 Overview

The objective of the *volume rendering component defined* by X3D is to provide volume rendering support in a declarative manner. Our *implementation of the component* integrates web technologies: CSS, HTML, JavaScript, WebGL and DOM scripting. The new component allows 3D volume rendering content and interactions to coexist in the web ecosystem.

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 volume rendering component. The document includes the declared X3D scene using HTML markup language (see Fig. 2).

Secondly, on the user device, when the browser loads the HTML/X3D document, our component and the volume data are loaded. At runtime the browser creates a 3D canvas and renders the 3D volume making use of the local computer GPU. As a result, when the user interacts with the 3D canvas new images are rendered in real-time (see Fig. 3).

The volume rendering component along with the X3DOM framework, provides support for volume rendering in a declarative manner at the client browser. Web content developers that are familiar with the X3D standard specification can integrate a volume rendering canvas within a web page. The component developed in this work makes it easier to create volumetric content for developers without specific knowledge on computer graphics rendering.

Our component approach 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.

Author's personal copy

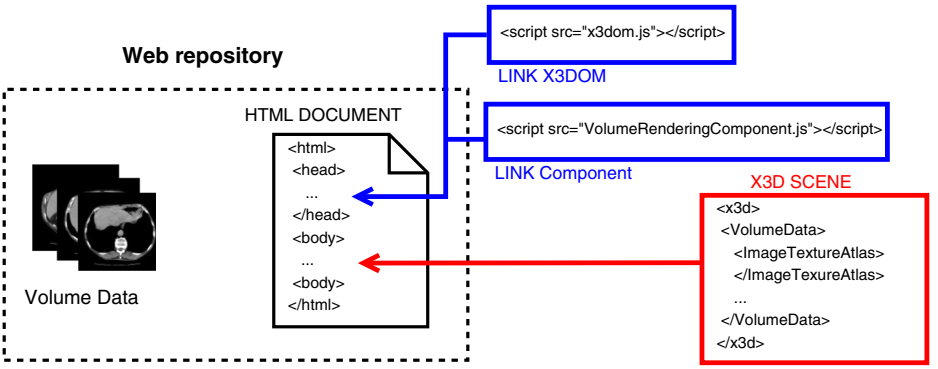


Fig. 2 Resources that must be created by a web content developer on the web repository

On Fig. 3 we have placed our component in the X3DOM repository, however as the component is a JavaScript library, it can be stored in any web repository. For large scale infrastructures, it can be duplicated on several servers or distributed through a content delivery network (CDN).

Moreover, the component provides a set of rendering styles in compliance with the X3D standard that have not been rendered by Web browsers before. These rendering styles provide the tools to improve the visualization of volumetric data at many fields. Examples of the use of the rendering styles at different fields will be shown in Section 6.

4 Methodology

This section presents our solution for the volume rendering component. We start introducing the pre-processing step required by our approach. Afterwards, the architecture and methodology used is presented. Later, the declarative nature of the X3DOM framework together with the component is detailed with an example. Finally, the last subsection deals with the rendering parameters that can be modified interactively by the user.

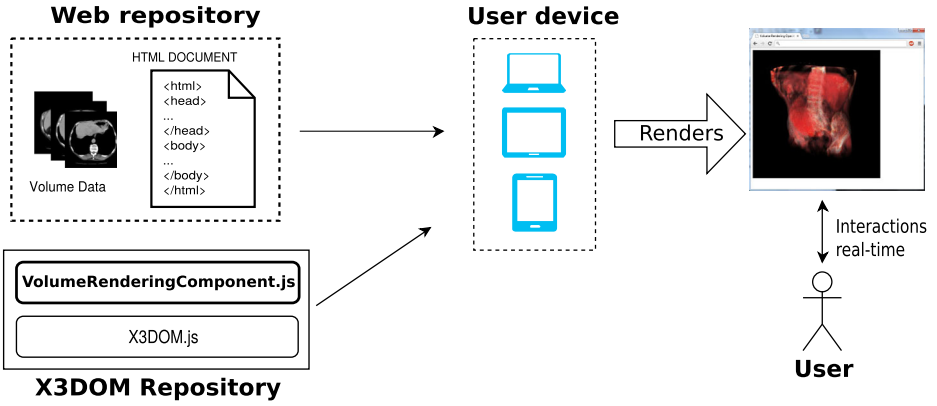
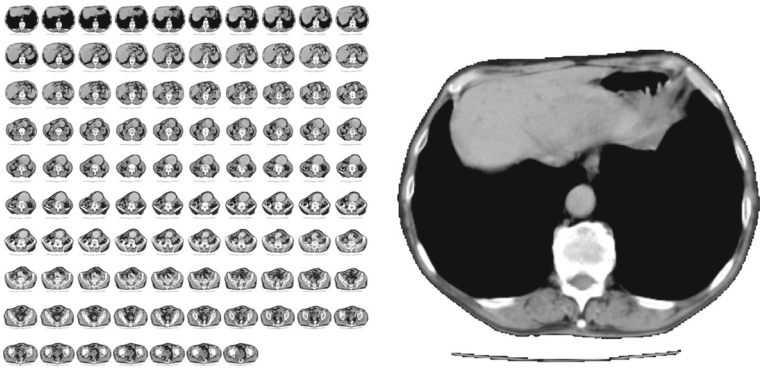


Fig. 3 Overview of direct volume rendering using the proposed component

Author's personal copy



(a) Volume data atlas (b) First volume data slice

Fig. 4 Volume data of the `aorta` dataset with transparent background. **a** Volume slices stored in one texture (matrix configuration). **b** The first slice of the volume data

4.1 Pre-processing

Volume data can be seen as a 3D array composed of cubic elements. The unit space that each cubic element represents is called voxel. Usually, in GPU-accelerated volume rendering, the volume data is stored as a 3D texture. However, WebGL does not support yet this data structure. This limitation can be overcome using the volume atlas method introduced by Congote et al. [7].

4.1.1 Volume data atlas

Our proposal requires an offline pre-processing of the volume data before it can be interactively visualized using WebGL. All the slices of the volume data are tiled into a single image on a matrix configuration. This image is called atlas (see Fig. 4).

4.1.2 Gradient data atlas

In order to enhance features or illuminate the volume, surface normals are needed. Unlike regular polygonal meshes, volumetric data does not have implicit normals. Our approach adds the normal to each voxel by calculating the voxel gradient from the volume data values. The computed gradient data is used as surface normals for the volume rendering nodes. We have implemented two different ways to manage the gradient data.

In the first method, the gradient is computed at the pre-processing step. The choice of the gradient operator is up to the user. A Sobel or a Gauss filter are suitable operators, but user-defined ones could also be applied.

The computed gradient data is encoded into an atlas texture with the same matrix configuration as the volume data atlas. We call gradient data atlas to the information stored in this structure (Fig. 5 shows a single slice of the gradient data atlas). Our approach stores each gradient vector in the RGB channels: {R: X, G: Y, B: Z} at its corresponding voxel.

The access to the gradient data atlas uses the same coordinates as the volume data. Therefore, the gradient vector is obtained on the shaders with only an additional texture fetch,

Author's personal copy

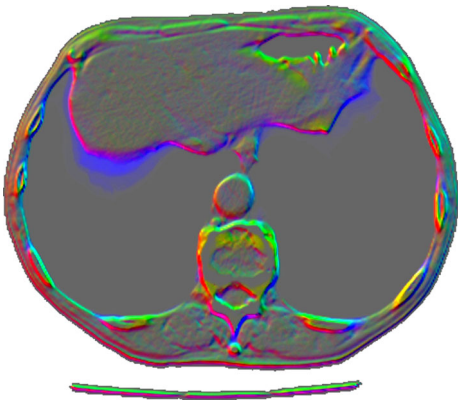


Fig. 5 A color-enhanced version of a gradient data slice of the `aorta` dataset with transparent background

being a good approach to preserve the overall performance. However, the amount of GPU memory increases due to the need of an extra texture.

In the second method, the gradient is computed on-the-fly using a central differences operator.

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

The gradient computation (∇) is a neighborhood operator. The (1) shows that, on each sampled voxel, six additional neighbor texture fetches are needed to compute the gradient.

Summing up, with the first method better performance is achieved than with the second method, but the extra memory required by the gradient data atlas could be a drawback for some specific hardware like mobile devices.

4.2 Atlas resolution

There is one consideration that must be taken into account regarding this approach: the volume reduction 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 require out-of-core algorithms as stated by [9, 14]. Congote et al. [8] showed the render quality achieved with the ray-casting algorithm under different steps and atlas resolutions. Using the statistics collected by WebGLStats in 2014, we assume 4096×4096 as the texture size limit supported by the majority of PCs, and 2048×2048 for mobile devices. The Table 2 summarizes the overall size reduction, down-sampling percent of the generated atlas, for the most supported texture sizes. We have used bicubic interpolation with Gimp in our tests to perform the down-sampling.

Table 2 Overall resolution reduction using atlases

Dataset	Atlas	Valid Atlas	8192^2	4096^2	2048^2	1024^2
128^3	1536^2	2048^2	0 %	0 %	0 %	33, 3 %
256^3	4096^2	4096^2	0 %	0 %	50 %	75 %
512^3	11776^2	16384^2	30, 4 %	65, 2 %	82, 6 %	91, 3 %

Author's personal copy

4.3 X3D volume node hierarchy

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 behavior.

The node hierarchy defined by X3D is composed of three abstract node types. The root node describes the volume data to be rendered. 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 *X3DComposableVolumeRenderStyleNode*, and they are declared as children of the *X3DVolumeDataNode*.

Style nodes that inherit from the *X3DComposableVolumeRenderStyleNode* can be composed: the output of a style can be the input of the next applied style.

The *ImageTextureAtlas* is an additional node not defined by the X3D standard. This node is used by our approach to provide the volume data or the gradient data previously defined in Section 4.1. Depending on the scene, the gradient data can be provided as an *ImageTextureAtlas*, and declared as a child node of the *X3DVolumeDataNode* or child node of the *X3DComposableVolumeRenderStyleNode*.

4.4 Shaders

Our 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 executed by shaders on the GPU. Shaders are a set of text strings that are passed to the graphics hardware driver for compilation and execution. Our approach is based on Congote et al. [7] and Mobeen and Feng [26]. A single shader (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, content developers can define a X3D scene with the desired rendering styles to get a specific visualization of such dataset. For instance, they could use different illustrative styles in two segments within a volume or they could compose a set of styles to enhance the contours of the volume.

Thus, the number of possible scenes is unbounded, and each scene requires its specific shader to implement the volume rendering. In this regard, to fulfill the dynamic requirements of X3D, we avoid 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 Fig. 6). This process starts when a new web page with X3D content is loaded.

The ray-casting loop is implemented in our fragment shader. Our solution generates automatically the shader code required for each X3D scene. We use 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 made in two steps: an *initialization phase* and a *shader code generation phase*.

Author's personal copy

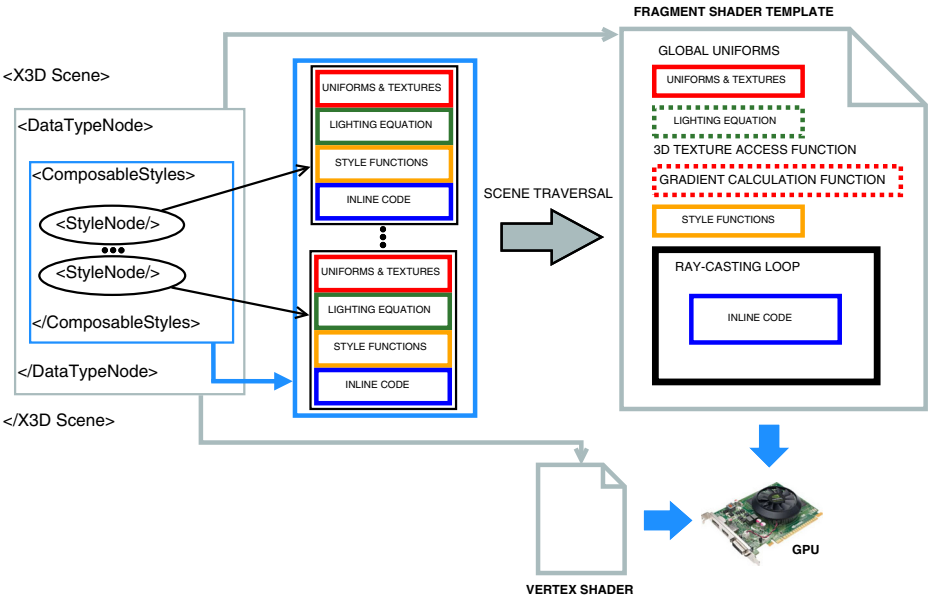


Fig. 6 Template-based shader code generation

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 assignation of a free texture image unit to each texture sampler must be managed and, *iv)* each variable must be correctly linked to its node parameter at the DOM tree, to allow dynamic changes on the shader variables (see Section 4.6).

In the shader code generation phase, the complete shader code is composed, compiled and linked in the GPU. The volume data type node (*X3DVolumeDataNode*) defines the base template of the fragment shader (see Fig. 6, 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, X3DComposableVolumeRenderStyleNode*) 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 Fig. 6, 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 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 may be added to the template if the user declares a light on the scene or it is mandatory to the style, e.g. *ShadedVolumeStyle* (Section 5.3).

Author's personal copy

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.

4.5 Declarative scene

The node architecture proposed by the X3D standard offers flexibility to compose a scene graph. With the X3DOM framework, X3D content can be integrated into a HTML document [1]. 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.

Figure 7 shows an example of a volumetric scene tree of the `backpack` [37] dataset, where each node type is shown with a distinctive color. In this example, a scene is defined with a *ComposedVolumeStyle* which includes two rendering styles. First, edges are enhanced with red color using the *EdgeEnhancementVolumeStyle* and then, the *SilhouetteEnhancementVolumeStyle* highlights the areas where the surface normals are perpendicular to the view direction. Figure 8 shows the rendering output of the X3DOM scene tree (see Fig. 7).

In a volume rendering scene, multiple volume data nodes can be declared. A custom shader will be generated for each declared *X3DVolumeDataNode*.

4.6 Interactivity

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 event handlers and listeners that change the attributes of a volume rendering node tag at the DOM tree.

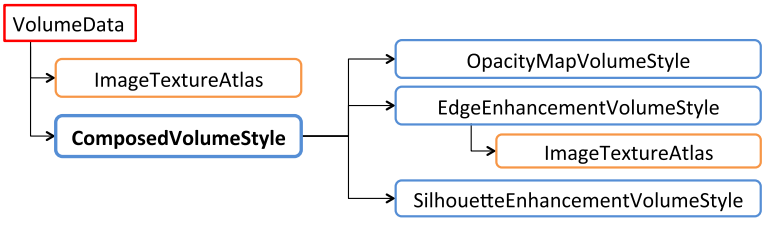
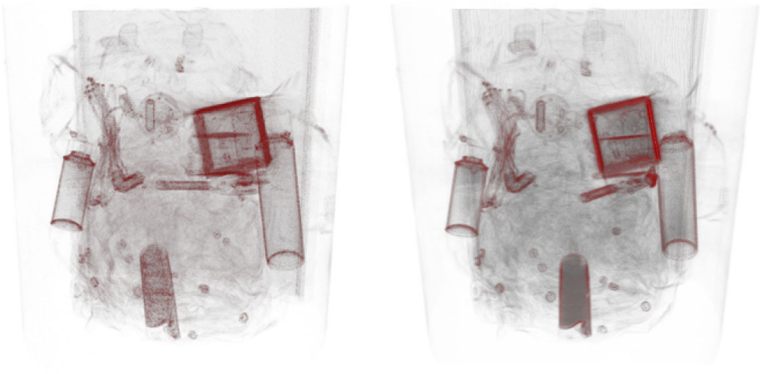


Fig. 7 X3DOM partial tree of a composed scene

Author's personal copy



(a) X3D example (b) Our implementation

Fig. 8 Two rendering outputs of the `backpack` dataset using the *ComposedVolumeStyle* with the *SilhouetteEnhancementVolumeStyle* and the *EdgeEnhancementVolumeStyle*. **a** Rendering output taken from the X3D standard. **b** Rendering output from our implementation

Input parameters of the rendering style nodes are usually shader uniform variables (see Fig. 6). 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.

Our 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 Fig. 6). The modification of such parameters, will require to regenerate the shaders again. For these cases, the scene must be reloaded to compile and link the new updated shader.

5 Implementation

This section describes our implementation of the nodes defined by X3D.

5.1 X3DVolumeDataNode

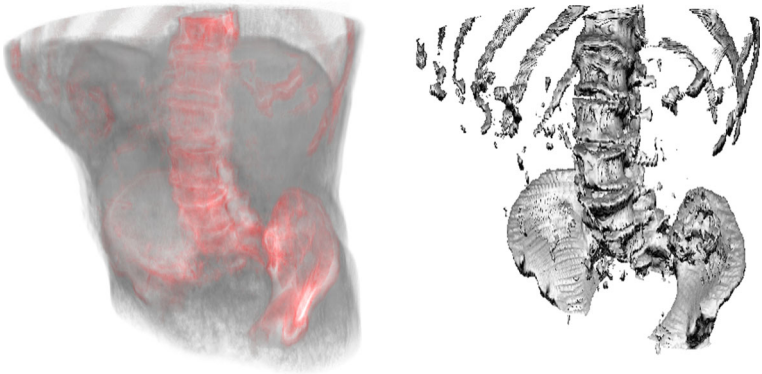
The *X3DVolumeDataNode* has three derived nodes: *VolumeData*, *SegmentedVolumeData* and *IsoSurfaceVolumeData*.

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.

The *SegmentedVolumeData* takes a segmented volume data as input. The segment identifier assigned to each voxel is not stored in the volume data. So, when required in the rendering process, we assign a segment identifier using (2).

$$id = \lfloor f(x) \times maxSegment - 0.5 \rfloor \tag{2}$$

Author's personal copy



(a) *Segmented Volume Data* (b) *Iso Surface Volume Data*

Fig. 9 Two rendering outputs of the `aorta` dataset using the *X3DVolumeDataNode*. **a** Two segments tissue and bones, the first rendered using a *BoundaryEnhancementVolumeStyle* and the second with an *EdgeEnhancementVolumeStyle*. **b** A single 0.92 isosurface value rendered with the *CartoonVolumeStyle*

In (2), $f(x)$ 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 Fig. 9a). In our implementation we have added the $maxSegment$ parameter for this node in order to adjust the way the segment identifiers are computed from the input segmented volume data.

The *IsoSurfaceVolumeData* allows the visualization of one or more surfaces extracted from the volume data (see Fig. 9b). *"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"* [40].

$$\begin{cases} C_g = styleNode(C_v, O_v) \wedge O_g = 1, \\ \quad \text{if } (f(x) \geq iso_v \vee f(x-1) < iso_v) \wedge \parallel \bigtriangledown f(x) \parallel \geq s_t \\ C_g = styleNode(C_v, O_v) \wedge O_g = 1, \\ \quad \text{if } (f(x) \leq iso_v \vee f(x-1) > iso_v) \wedge \parallel \bigtriangledown f(x) \parallel \geq s_t \\ C_g = C_v \wedge O_g = 0, \qquad \text{otherwise} \end{cases} \tag{3}$$

Multiple isovalues can be given as parameters to the style. Equation (3) shows the conditional statement we use to check if a voxel belongs to a given isosurface. C_v and O_v are the original voxel color and opacity. C_g and O_g are the generated output color and opacity. When multiple isovalues are given, a rendering style is associated to each isovalue, following the rules of the X3D specification.

5.2 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 color based on the voxels values traversed by a ray. *Maximum Intensity Projection* (MIP) stores the greatest value along the ray (see Fig. 10). It is widely used in the medical field. It was originally proposed by Wallis et al. [38] for

Author's personal copy

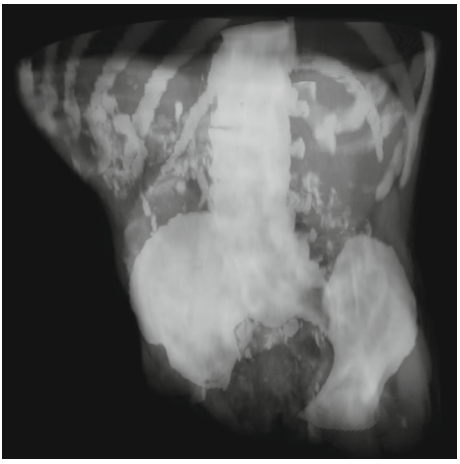


Fig. 10 The rendering output of the `aorta` dataset using the *ProjectionVolumeStyle* with the *max* method (MIP)

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 [30].

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.

5.3 X3DComposableVolumeRenderStyleNode

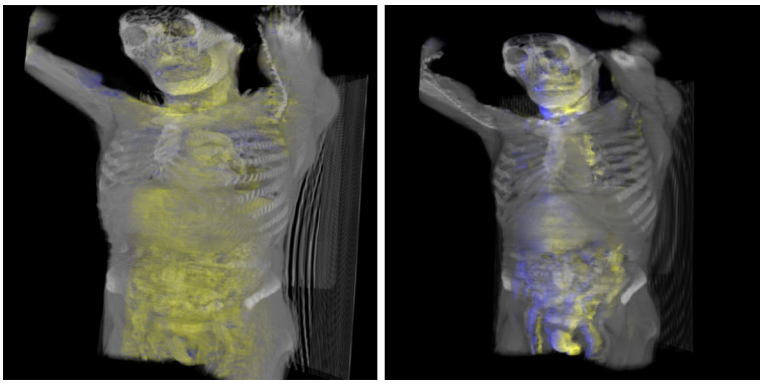
Nodes derived from the *X3DComposableVolumeRenderStyleNode* can be composed resulting in richer renderings. The *X3DComposableVolumeRenderStyleNode* has the following derived nodes: *ComposedVolumeStyle*, *BlendedVolumeStyle*, *CartoonVolumeStyle*, *OpacityMapVolumeStyle*, *BoundaryEnhancementVolumeStyle*, *EdgeEnhancementVolumeStyle*, *SilhouetteEnhancementVolumeStyle*, *ToneMappedVolumeStyle* and *ShadedVolumeStyle*.

The *ComposedVolumeStyle* allows compositing multiple *X3DComposableVolumeRenderStyleNode* 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. In our implementation, the styles are applied 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 up to 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 Fig. 11). The main dataset is the parent *X3DVolumeDataNode* 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 colors as input parameters. The final rendering will depend on the local surface normals and the view direction. The result is a cartoon-style non-photorealistic rendering [10]. Our implementation differs slightly from the specification.

Author's personal copy



(a) X3 Dexample (b) Our implementation

Fig. 11 Two rendering outputs using the *BlendedVolumeStyle* with the `body` [39] and `internals` [39] 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`

We do not take into account the alpha channel from the input colors. Instead, the opacity values are obtained from the volume data.

The *OpacityMapVolumeStyle* maps the opacity and color values for each voxel from a function stored as a texture. This texture is called transfer function. The creation of this transfer function is up to the designer and is created in an offline preliminary step. Extensive work has been done regarding this topic. Kniss et al. [19] denoted the use of multi-dimensional transfer functions. Bruckner and Gröller implemented illustrative styles through transfer functions [3]. We have followed the X3D specification regarding this style, supporting 1D transfer function textures.

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.

$$O_g = O_v \times (K_{gc} + K_{gs} \times (\| \bigtriangledown f(x) \|^{K_{ge}})) \tag{4}$$

The (4) 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* stands out the edges of the volume with an input color parameter. Edges are volume data values where the gradient is perpendicular to the view direction. The input color is blended with the volume data color in function of the angle between both vectors (see (5)).

$$C_g = \begin{cases} C_v \times \mid \bigtriangledown f(x) \cdot V \mid + edgeColor \times (1 - \mid \bigtriangledown f(x) \cdot V \mid), \\ \quad \text{if} \mid \bigtriangledown f(x) \cdot V \mid > \cos(gradThreshold) \\ C_v, \quad \text{otherwise.} \end{cases} \tag{5}$$

$$O_g = O_v \tag{6}$$

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

Author's personal copy

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 color.

$$O_s = O_v \times (K_{sc} + K_{ss} \times (1 - \mid \bigtriangledown f(x) \cdot V \mid^{K_{se}})) \tag{7}$$

The (7) 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. [15] were the first to propose an illumination model following this approach. This tone shading technique defines two colors: warm and cool. The warm color represents surfaces facing towards the light direction, and the cool color is used for surfaces facing away the light. The interpolation between these colors is assigned using the dot product between the angles of the gradient and the light direction to each sampled voxel.

Currently, our implementation supports local illumination following the Blinn-Phong illumination model [2]. Additionally, the *ShadedVolumeStyle* node allows to specify the fog and material properties. Due to the extra computational cost that imply global illumination models, they have been considered for future work.

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 12 shows the rendering output of our implementation for each described *X3DComposableVolumeRenderStyleNode*.

6 Results

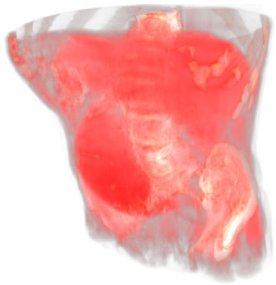
This section shows the advantages, flexibility and utility of the declarative approach for volume rendering. It shows the powerful and easy-to-use tool for web content developers. Four volumetric datasets from different thematic areas are presented: medical, engineering, physics and life sciences. For 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 visualization is introduced, then a basic render of the volume is shown. Afterwards, a partial X3D scene for each use case is presented. Finally, we show a table that resumes the performance achieved on each of the presented figures.

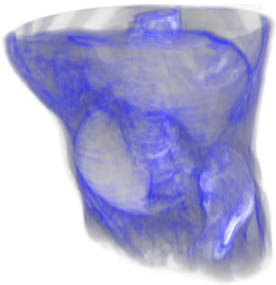
6.1 Medical use case

Undoubtedly, a useful tool in the medical field is the segmentation, i.e. the partitioning of the volume data into different segments. 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 visualize 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. We use the `Head MRI` [37] dataset. It consists of a Magnetic Resonance Imaging

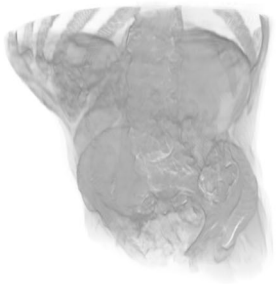
Author's personal copy



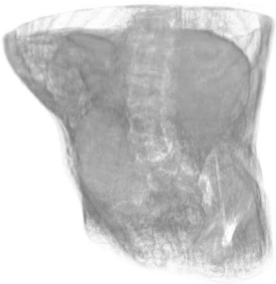
(a) *Opacity Map Volume Style*



(b) *Edge Enhancement Style*

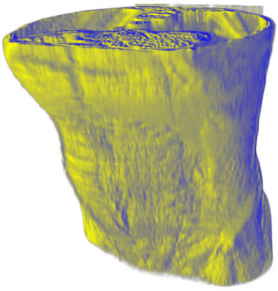


(c) *Boundary Enhancement-Volume Style*

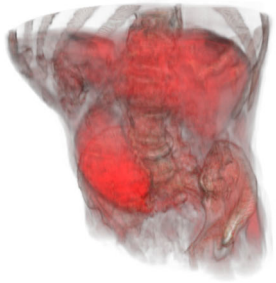


(d) *Silhouette Enhancement-Volume Style*

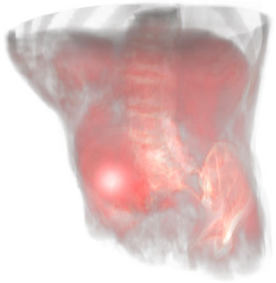
(e) *Cartoon Volume Style*



(f) *Tone Mapped Volume Style*



(g) *Composed Volume Style*



(h) *Shaded Volume Style*

Fig. 12 The rendering output of each *X3DComposableRenderStyleNode* using the `aorta` dataset

Author's personal copy

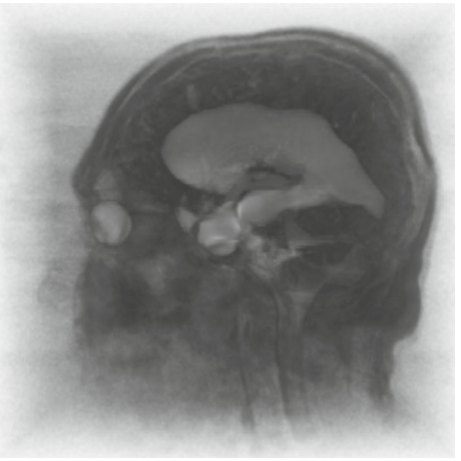


Fig. 13 Direct volume rendering visualization of the `Head MRI` dataset (2048 × 2048 atlas), using the *OpacityMapVolumeStyle* without a transfer function

(MRI) scan of the head and a segmentation of the ventricles of the brain. With the use of the *SegmentedVolumeData* node, we enhance the ventricles shape from the rest of the volume data. Figure 13 shows a basic visualization of the *Head MRI* dataset, without the use of the segmentation data.

Using the segments information, we can apply a rendering style to each one. Figure 15 shows two rendering outputs that highlight the ventricles of the brain. Both use the *SegmentedVolumeData* node with two different rendering styles. Figure 14 is a partial X3D scene tree of Fig. 15a. First, the volume is declared as a *SegmentedVolumeData*. In this case, two atlases must be provided to the component: the volume data atlas 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.

In the scene tree defined in Fig. 14, the first segment has been rendered using the *OpacityMapVolumeStyle* with a low *opacityFactor* making more visible the insides of the head. The second segment, the ventricles of the brain, has been rendered with a composition of two rendering styles: the *OpacityMapVolumeStyle* and the *EdgeEnhancementVolumeStyle*. Making the segment more opaque and highlighting the shape of the ventricles. The difference between Fig. 15a and b is the render style used on the second segment. In Fig. 15b we have used the *CartoonVolumeStyle* instead of the *EdgeEnhacementVolumeStyle*.

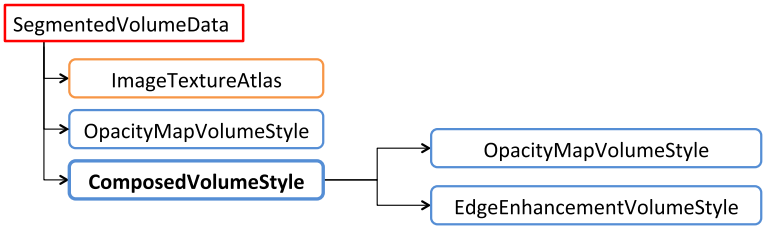
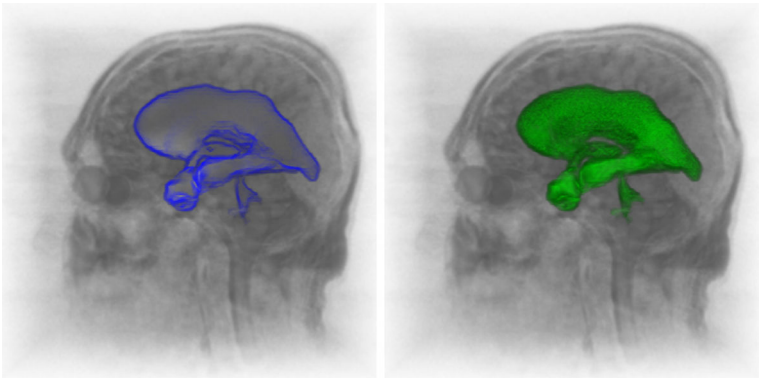


Fig. 14 Partial X3D scene tree of the `Head MRI` dataset, using the *SegmentedVolumeData* node

Author's personal copy



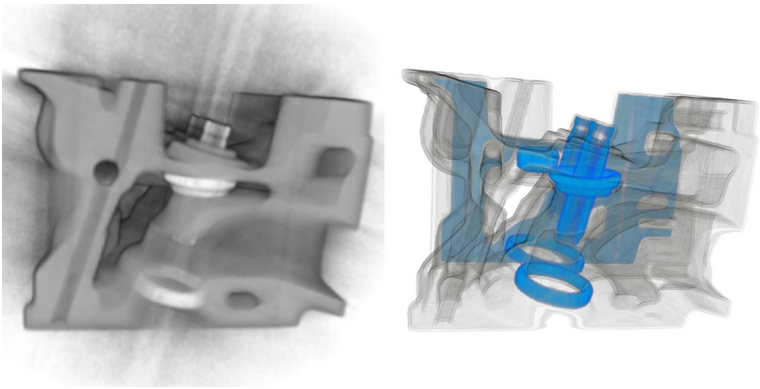
(a) *Edge Enhancement Volume Style* (b) *Cartoon Volume Style*

Fig. 15 Direct volume rendering visualization of the `Head MRI` dataset (2048 × 2048 atlas), using the *SegmentedVolumeData* to enhance the ventricles of the brain

6.2 Engineering use case

In the engineering field, we show a use case using the `engine` [37] dataset. This dataset consists of a Computed Tomography (CT) scan of an engine block. In this use case, we aim to visually enhance a region of interest: the two cylinders inside the engine. We will achieve this objective showing two different resulting scenes: firstly, with the aid of a transfer function, and secondly, the cylinders are extracted with the visualization of an isosurface. Figure 16a shows the `engine` dataset, with the default rendering style: the *OpacityMapVolumeStyle*.

Using a 1D *transfer function* on the *OpacityMapVolumeStyle*, we can map each opacity value from the volume data to a color and opacity, enhancing and illustrating the volume (see Figs. 16b and 17).



(a) Engine dataset (b) Engine with TF

Fig. 16 Direct volume rendering visualization of the `engine` dataset with a 2048 × 2048 atlas. **a** Basic visualization (by default *OpacityMapVolumeStyle*). **b** Engine dataset with a 1D transfer function

Author's personal copy

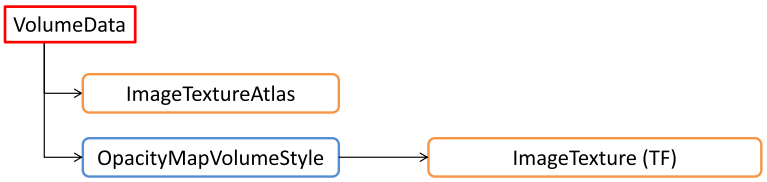


Fig. 17 Partial X3D scene tree declaration of the `engine` dataset, using the *OpacityMapVolumeStyle* with a *transfer function*

The creation of the transfer function is out of the scope of the volume rendering component. 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 Figs. 18 and 19).

In Fig. 18 we have used the *CartoonVolumeStyle* to illustrate the extracted isosurfaces, showing a comparison between different *colorSteps*. A more *cartoonish* effect can be achieved when the number of *colorSteps* is low (Fig. 18a and b), whereas a higher value of *colorSteps* can be used to get a more solid appearance (see Fig. 18e). Note that in this case, we do not see the whole volume as before (Fig. 16b). Both presented solutions are able to

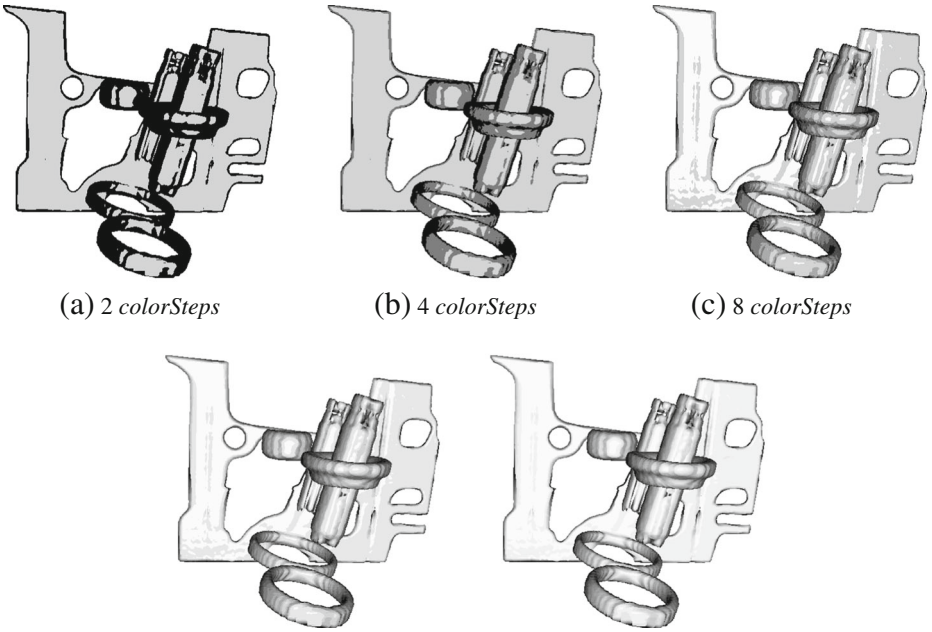


Fig. 18 Direct volume rendering visualization of the `engine` dataset with a 2048 × 2048 atlas. Dataset declared as an *IsoSurfaceVolumeData* with a set of *surfaceValues* of [0.7, 0.75, 0.8] and illustrated with the *CartoonVolumeStyle* at different *colorSteps*

Author's personal copy

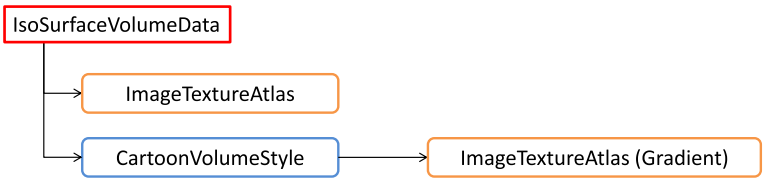


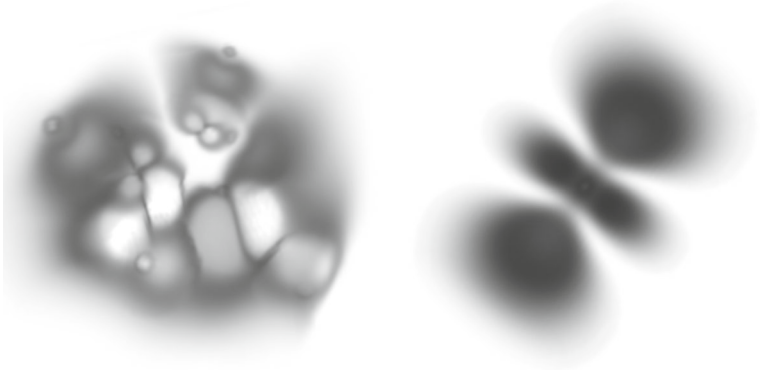
Fig. 19 Partial X3D scene tree of the `engine` dataset, declaring the volume as a *IsoSurfaceVolumeData* and using a *CartoonVolumeStyle*

visualize the cylinders, which denotes the flexibility of the proposed component and the X3D specification (Fig. 19).

6.3 Physics use case

Volume rendering is useful for scientific volume data visualization. We have selected the `neghip` [37] and `hydrogen-atom` [37] datasets to show examples of the utility of our volume rendering component in the nuclear physics field. The `neghip` dataset ($64 \times 64 \times 64$) 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, for example, to better understand the relation between molecules and the organism [23]. 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. In both cases, we aim to observe the shape of the density distributions, by visualizing a selection of isovalues from the datasets. A basic visualization of both datasets (Fig. 20) 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.

To explore the distribution, we present a X3D example (see X3D nodes in Fig. 21) on how to visualize a set of isovalues and color them (see Fig. 22). The use of the



(a) Neghip dataset (b) Hydrogen atom dataset

Fig. 20 Direct volume rendering visualization of the `neghip` (512×512 atlas) and `hydrogen atom` (1024×1024 atlas) datasets with no styles applied

Author's personal copy

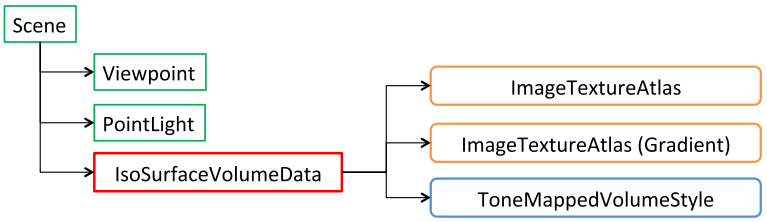


Fig. 21 Partial X3D scene tree for the `neghip` dataset illustrated with the *ToneMappedVolumeStyle*

IsoSurfaceVolumeData allows us to apply a rendering style on each specified *surfaceValue*. The illustration of the volume is necessary to enhance the perception of depth in the volume.

Figures 22 and 23 are defined with the same tree (Figs. 21 and 24). In both cases, an isosurface of the volume is being visualized with the *IsoSurfaceVolumeData* node and then, illustrated with the *ToneMappedVolumeStyle*. We can select the desired isosurface by specifying a *surfaceValue*, and limit the surface detection with the *surfaceTolerance* parameter (Fig. 23). The *PointLight* node is declared in both scenes, because it is necessary for the ToneMappedVolumeStyle to know the light location or direction.

6.4 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 understatement of its composition and morphology is obtained. Thus, our approach can be used to complement web articles and teaching material. As an example, we have selected the `orange`[1] dataset ($256 \times 256 \times 64$) to visualize its inner structure. Figure 25a shows a basic rendering of the `orange` dataset. This can be easily declared in a few lines of HTML and X3D.

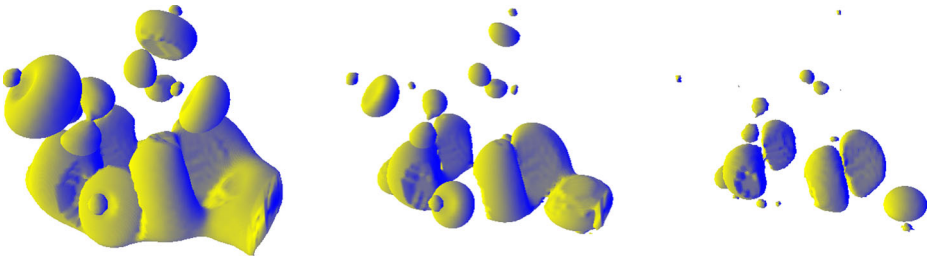
Figure 25a shows a cartoon rendering of the orange illustrated with orange and yellow colors to make it closer in appearance to the real fruit. As an alternative, a transfer function could be used to achieve a similar result, but it would be a more time consuming approach if the transfer function has to be edited manually. Figure 26 shows the scene tree used to illustrate the volume with the *CartoonVolumeStyle*.

In this use case, we aim to highlight the composition of the orange. A quick and effective way to achieve our objective is to enhance the boundaries and silhouette considerably. Figure 27 shows the composition of several rendering style nodes, which allows us to visualize the orange sections and seeds.

The final rendering (Fig. 27) is produced using the scene at Fig. 28. The *ComposedVolumeStyle* provides a way to combine different style nodes. The first stage of this case is to use an *OpacityMapVolumeStyle* to adjust the amount of opacity accumulated on each sample. Then, we have applied a *BoundaryEnhancementVolumeStyle* to make more noticeable the changes between regions inside the orange. Afterwards, we have applied the *SilhouetteEnhancementVolumeStyle* retaining very little opacity from the original volume by making slightly visible the contours of the volume. Finally, we have used the *EdgeEnhancementStyle* to fill with color the previous filtered contour.

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

Author's personal copy



(a) *surface Value* 0.2 (b) *surface Value* 0.4 (c) *surface Value* 0.9

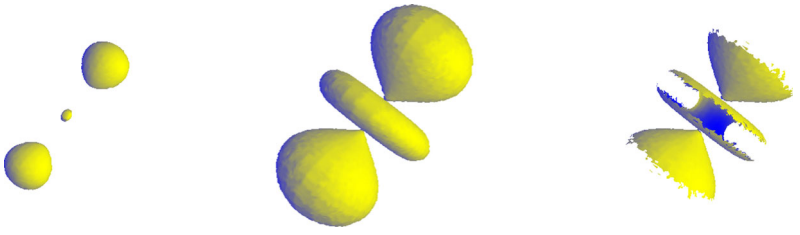
Fig. 22 Volume rendering visualization of the `neghip` dataset (512 × 512 atlas). Using the *IsoSurfaceVolumeData* node with a single isovalue on the *surfaceValues* parameter and illustrated with the *ToneMappedVolumeStyle*

6.5 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 38 and Firefox 32. Both Firefox and Chrome 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 3 summarizes the performance obtained on each of the cases described before.

For the creation of the figures and performance tests, we have used 120 steps, i.e., each ray is sampled 120 times at maximum in the ray-casting method. If the ray comes out of the volume, or the accumulated opacity reaches the value 1, the ray sampling is stopped. By default, the release 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.

In general, we can affirm that the results are good to be used in consumer oriented desktop computers with domestic PC graphics cards. Table 4 summarizes the advantages and disadvantages of our Web based methodology in comparison with the desktop based solutions presented in Section 2. As a Web based approach, the X3DOM framework is



(a) *surface Value* 0.20 (b) *surface Value* 0.05 (c) *surface Value* 0.05 and *surface Tolerance* 0.035

Fig. 23 Volume rendering visualization 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

Author's personal copy

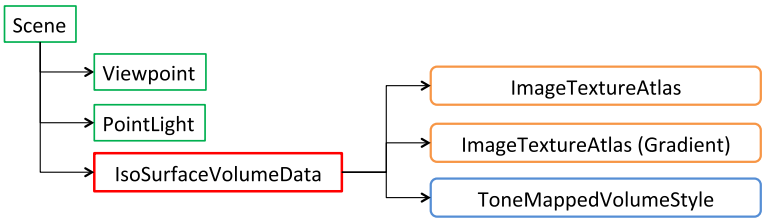
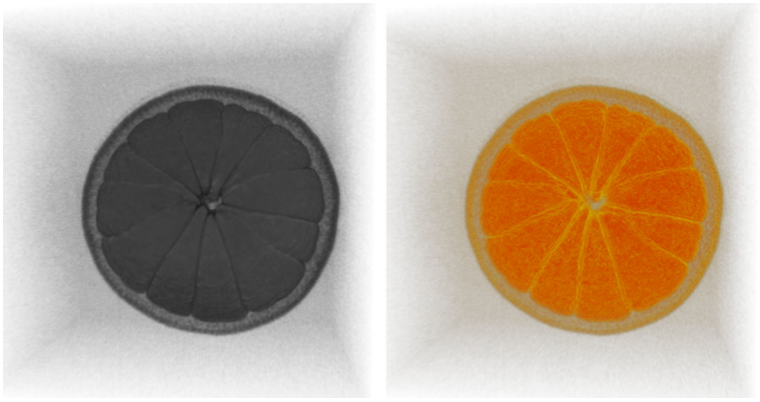


Fig. 24 Partial X3D scene tree of the `hydrogen-atom` dataset illustrated with the *ToneMappedVolumeStyle*



(a) Orange dataset

(b) Composed with Opacity Map and Cartoon styles

Fig. 25 Volume rendering of the `orange` dataset with a 1024 × 1024 atlas. **a** Basic volume visualization (by default *OpacityMapVolumeStyle*). **b** `Orange` dataset rendered with the *OpacityMapVolumeStyle* and *CartoonVolumeStyle*

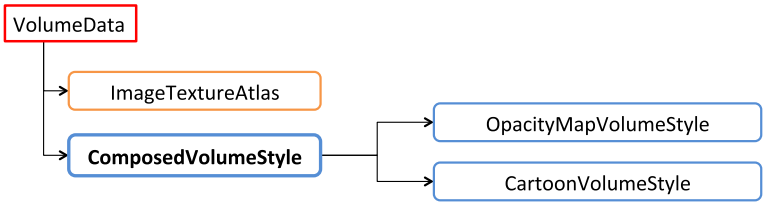


Fig. 26 Partial X3D scene tree of the `orange` dataset to illustrate the volume

Author's personal copy

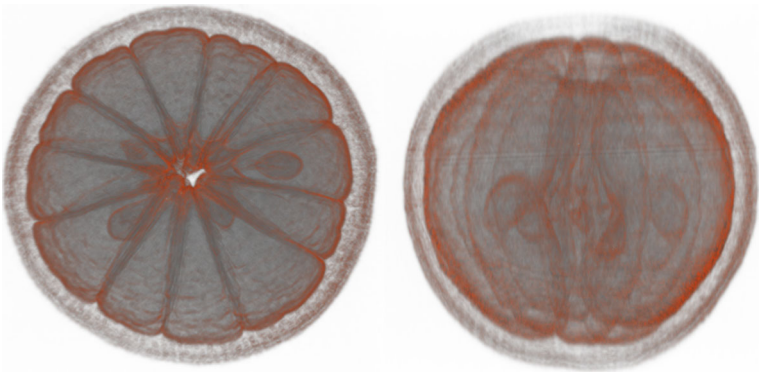


Fig. 27 Volume rendering of the `orange` dataset with a 1024×1024 atlas from two point of views. Applying several composable styles to highlight the sections and seeds of the orange

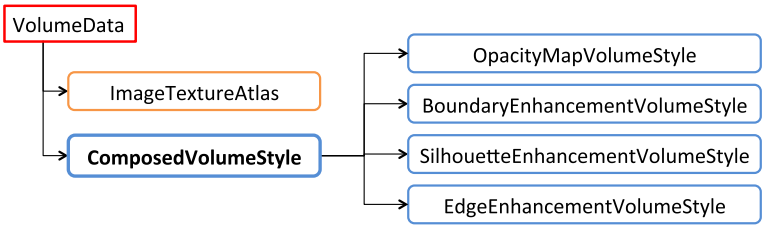


Fig. 28 Partial X3D scene tree, enhancing and highlighting the insides of the `orange` dataset

Table 3 Performance, frames per second (FPS) on each use case example at different resolutions

Use case	Figure	Dataset	Resolution	Gradient atlas	FPS
6.1	Fig. 13	`Head MRI`	2048 × 2048	no	50-55
6.1	Fig. 15a	`Head MRI`	2048 × 2048	no	25-35
6.1	Fig. 15b	`Head MRI`	2048 × 2048	no	40-50
6.2	Fig. 16a	`engine`	2048 × 2048	no	50-60
6.2	Fig. 16b	`engine`	2048 × 2048	no	50-55
6.2	Fig. 18	`engine`	2048 × 2048	no	30-39
6.3	Fig. 22a	`neghip`	512 × 512	yes	40-45
6.3	Fig. 22b	`neghip`	512 × 512	yes	40-45
6.3	Fig. 22c	`neghip`	512 × 512	yes	38-45
6.3	Fig. 23a	`hydrogen-atom`	1024 × 1024	yes	38-45
6.3	Fig. 23b	`hydrogen-atom`	1024 × 1024	yes	40-45
6.3	Fig. 23c	`hydrogen-atom`	1024 × 1024	yes	38-45
6.4	Fig. 25a	`orange`	2048 × 2048	no	50-60
6.4	Fig. 25b	`orange`	2048 × 2048	no	40-45
6.4	Fig. 27	`orange`	2048 × 2048	no	25-35

As a help to the reader, the section and figures are referred

Author's personal copy

Table 4 Summary of the key advantages and disadvantages of our Web based approach against other desktop based approaches

Our Web based approach	Desktop based approaches
Datasets up to 512 × 512 × 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

available for all compatible WebGL devices, being the interactivity rate limited by the GPU computational power of the device.

7 Conclusions

The X3D Medical Working Group defined a volume rendering component and its nodes for 3D visualization of volumetric data. This paper shows how the challenge placed by these definitions has been solved. Section 6 shows, how web content developers can easily declare these 3D interactive visualization canvases. It also presents graphical results that show different applications and renders that can be interactively displayed. Our work demonstrates the power of X3D definitions once their implementation challenges have been solved. Its performance achieves interactive rates in domestic PC web browsers.

We have presented the first volume rendering component based on WebGL for real-time volume rendering that supports multiple illustrative and non-photorealistic styles in a declarative manner. In compliance with the X3D standard, the set of styles available with the component is useful to enhance features from the volume data. The render styles are suitable for any volumetric visualization. Among them, some have direct applications in the medical field, like the MIP. The integration of the component in X3DOM does not only provide the advantages of a declarative approach, but it also provides an opportunity to create web-based applications on top of it.

We have implemented all the nodes defined by the X3D volume rendering component specification, as described in the Section 5. The flexibility and utility of the framework has been proved in Section 6, where we have showed the rendering outputs of several datasets that go beyond a basic visualization. A few lines of HTML and X3D are enough to enhance or highlight the data hidden within the volumetric datasets. A user evaluation of the proposed component will provide deeper insights and conclusions of the capabilities of the volume rendering component in real world scenarios.

The supported volume datasets are limited by the target GPU texture size limit and the dataset resolution (see Section 4.2). Even if the proposed implementation can not compete in terms of performance and rendering quality to current desktop solutions, the presented

Author's personal copy

results are good enough for the visualization of volumes up to 512 × 512 × 512. Bigger datasets will require the utilisation of out-of-core ray-casting algorithms and novel volume data streaming techniques.

The gap between desktop and web graphics may close in future versions of WebGL as more functionality of the GPU will be available to the browser. Until then, we consider appropriate the inclusion of the *ImageTextureAtlas* node in the X3D standard to support volume rendering in WebGL 1.0 compatible browsers and devices. In addition, we see necessary a new parameter at the *X3DVolumeDataNode* level that regulates the *qualityLevel* of the rendering output. The cross-platform capabilities of WebGL open volume rendering to devices with far less GPU computational power than desktop computers. Allowing to regulate the *qualityLevel* will help to target different devices and maintain performance across them. Also, we consider that it would be interesting to add more illustrative rendering styles on top of the X3D specification like hatching [35] and stippling [24].

Although limited by GPU power on some mobile devices, the rapid growth of these devices makes them appropriate for real-time graphics applications as its computational power is expected to improve over the years, making the volume rendering component available to a bigger number of devices. The presented implementation shows the possibilities of 3D graphics on the Web. The component with all the nodes described at Section 5 are already available for public use at the public X3DOM repository [13].

References

1. Behr J, Eschler P, Jung Y, Zöllner M (2009) X3DOM: A DOM-based HTML5/X3D integration model. In: Proceedings of the 14th International Conference on 3D Web Technology, ACM, New York, NY, USA, Web3D '09, pp 127–135, doi:10.1145/1559764.1559784
2. Blinn JF (1977) Models of light reflection for computer synthesized pictures. SIGGRAPH Comput Graph 11(2):192–198. doi:10.1145/965141.563893
3. Bruckner S, Gröller ME (2007) Style Transfer Functions for Illustrative Volume Rendering. Comput Graph Forum 26(3):715–724. doi:10.1111/j.1467-8659.2007.01095.x
4. Cabello R (2014) Three.js a JavaScript 3D library. http://www.threejs.org
5. Catuhe D, Rousseau M, Lagarde P, Rousset D (2014) Babylon.js a 3D engine based on webgl and javascript. http://www.babylonjs.com
6. Congote J (2012) MEDX3DOM: MEDX3D for X3DOM. In: Proceedings of the 17th International Conference on 3D Web Technology, ACM, NY, USA, Web3D '12, pp 179–179, doi:10.1145/2338714.2338746
7. Congote J, Seguram A, Kabongom L, Morenom A, Posadam J, Ruizm O (2011) Interactive Visualization of Volumetric Data with WebGL in Real-time. In: Proceedings of the 16th International Conference on 3D Web Technology, ACM, NY, USA, Web3D '11, pp 137–146, doi:10.1145/2010425.2010449
8. Congote J, Kabongo L, Moreno A, Segura A, Beristain A, Posada J, Ruiz O (2012) Volume Ray Casting in WebGL. InTech, doi:10.5772/34878
9. Crassin C, Neyret F, Lefebvre S, Eisemann E (2009) Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In: Proceedings of the 2009 symposium on Interactive 3D graphics and games. ACM, pp 15–22
10. Decaudin P (1996) Cartoon Looking Rendering of 3D Scenes. Research Report 2919, INRIA, http://phildec.users.sf.net/Research/RR-2919.php
11. Ebert D, Rheingans P (2000) Volume illustration: non-photorealistic rendering of volume models. In: Proceedings of the Conference on Visualization '00, IEEE Computer Society Press, CA, USA, VIS '00, pp 195–202, http://dl.acm.org/citation.cfm?id=375213.375241
12. Fraunhofer IGD (2014) X3DOM. http://www.x3dom.org
13. Fraunhofer IGD (2016) X3DOM Github repository. https://github.com/x3dom/x3dom
14. Gobbetti E, Marton F, Guitián JAI (2008) A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. Vis Comput 24(7-9):797–806

Author's personal copy

15. Gooch A, Gooch B, Shirley P, Cohen E (1998) A Non-photorealistic Lighting Model for Automatic Technical Illustration. In: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, ACM, NY, USA, SIGGRAPH '98, pp 447–452, doi:10.1145/280814.280950
16. Gutenko I, Petkov K, Papadopoulos C, Zhao X, Park JH, Kaufman A, Cha R (2014) Remote volume rendering pipeline for mHealth applications. In: SPIE Medical Imaging, International Society for Optics and Photonics, pp 903,904–903,904
17. Hähn D, Rannou N, Ahtam B, Grant E, Pienaar R (2012) Neuroimaging in the Browser using the X Toolkit. In: Frontiers Neuroinformation Conference Abstract: 5th INCF Congress of Neuroinformatics. doi:10.3389/conf. fninf, Neuroinformatics
18. Kajiya JT, Von Herzen BP (1984) Ray Tracing Volume Densities. In: Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, ACM, New York, NY, USA, SIGGRAPH '84, pp 165–174, doi:10.1145/800031.808594
19. Kniss J, Kindlmann G, Hansen C (2002) Multidimensional transfer functions for interactive volume rendering. IEEE Trans Vis Comput Graph 8(3):270–285. doi:10.1109/TVCG.2002.1021579
20. Kruger J, Westermann R (2003) Acceleration techniques for GPUbased volume rendering. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), IEEE Computer Society, DC, USA, VIS'03, pp 38, doi:10.1109/VIS.2003.10001
21. Levoy M (1988) Display of Surfaces from Volume Data. IEEE Comput Graph Appl 8(3). doi:10.1109/38.511
22. Li W, Mueller K, Kaufman A (2003) Empty space skipping and occlusion clipping for texture-based volume rendering. In: Proceedings of the 14th IEEE Visualization 2003 (VIS'03), IEEE Computer Society, DC, USA, VIS '03, pp 42–, doi:10.1109/VISUAL.2003.1250388
23. Linsen L, Hagen H, Hamann B, Hege H (2012) Visualization in medicine and life sciences ii: progress and new challenges. Mathematics and Visualization, Springer. http://link.springer.com/book/10.1007%2F978-3-642-21608-4
24. Lu A, Morris CJ, Ebert DS, Rheingans P, Hansen C (2002) Non-photorealistic volume rendering using stippling techniques. In: Proceedings of the Conference on Visualization '02, IEEE Computer Society, DC, USA, VIS '02, pp 211–218, http://dl.acm.org/citation.cfm?id=602099.602131
25. Lum EB, Ma KL (2002) Hardware-accelerated parallel non-photorealistic volume rendering. In: Proceedings of the 2nd international symposium on Non-photorealistic animation and rendering. ACM, pp 67–ff
26. Mobeen M, Feng L (2012) 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-ICESS), pp 381–388. doi:10.1109/HPCC.2012.58
27. Movania MM, Chiew WM, Lin F (2014) On-site volume rendering with GPU-enabled devices. Wirel Pers Commun 76(4):795–812
28. Noguera JM, Jiménez JR (2012) Visualization of very large 3D volumes on mobile devices and webGL. WSCG Communication Proceedings pp 105–112
29. Noguera JM, Jiménez JR, Ogáyar CJ, Segura RJ (2012) Volume rendering strategies on mobile devices. In: GRAPP/IVAPP, pp 447–452
30. Perandini S, Faccioli N, Zaccarella A, Re T, Mucelli R (2010) The diagnostic contribution of CT volumetric rendering techniques in routine practice. Indian J Radiol Imaging 20(2):63043. doi:10.4103/0971-3026
31. Pinson C (2014) OSG.JS WebGL framework based on OpenSceneGraph concepts. http://www.osgjs.org
32. Polys N, Wood A (2012) New platforms for health hypermedia. Issues Inf Syst 13(1):40–50
33. Polys N, Wood A, Shinpaugh P (2011) Cross-platform presentation of interactive volumetric imagery. Technical Report Departmental Technical Report 1177. Virginia Technology, Advanced Research Computing
34. Polys NF, Ullrich S, Evestedt D, Wood AD, Aratow M (2013) A fresh look at immersive Volume Rendering: Challenges and capabilities. IEEE VR Workshop on Immersive Volume Rendering, Orlando
35. Praun E, Hoppe H, Webb M, Finkelstein A (2001) Real-time hatching. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, ACM, NY, USA, SIGGRAPH '01, pp 581–, doi:10.1145/383259.383328
36. Stegmaier S, Strengert M, Klein T, Ertl T (2005) A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In: Proceedings of the Fourth Eurographics / IEEE VGTC Conference on Volume Graphics, Eurographics Association, Aire-la-Ville, Switzerland, VG'05, pp 187–195, doi:10.2312/VG/VG05/187-195
37. University of Tübingen WSI/GRIS (2014) Collection of volumetric datasets. http://www.volvis.org

Author's personal copy

38. Wallis J, Miller TR, Lerner C, Kleerup E (1989) Three-dimensional display in nuclear medicine. IEEE Trans Med Imaging 8(4):297–230. doi:10.1109/42.41482
39. Web3DConsortium (2014a) Extensible 3D (X3D) basic example archives. http://www.web3d.org/x3d-resources/content/examples/Basic/VolumeRendering/
40. Web3DConsortium (2014b) Extensible 3D (X3D) specifications. http://www.web3d.org/x3d/specifications/
41. Wong PC, Thomas J (2004) Visual analytics. IEEE Comput Graph Appl 24(5):20–21. doi:10.1109/MCG.2004.39
42. Yang F, Yang F, Li X, Tian J (2014) Ray feature analysis for volume rendering. Multimedia Tools and Applications pp 1–21, doi:10.1007/s11042-014-1994-2
43. Zhou Z, Tao Y, Lin H, Dong F, Clapworthy G (2014) Occlusion-free feature exploration for volume visualization. Multimedia Tools and Applications pp 1–16, doi:10.1007/s11042-014-2162-4

Ander Arbelaiz obtained his Computer Science degree in 2013 from the University of the Basque Country (UPV/EHU) and in 2014, he received a Masters Degree on "Computational Engineering and Intelligent Systems". In 2013 he joined Vicomtech-IK4 (http://www.vicomtech.org) as a research assistant. His research interests include computer graphics, volume rendering, simulation and GPU computing.

Aitor Moreno received a Ph.D degree in Computer Science in 2013 from the University of the Basque Country. His thesis titled "Urban and forest fire propagation and extinguishment in Virtual Reality training scenarios" received the "Cum Laude" designation. In 2002, he received a degree in Computer Science from the University of the Basque Country. In October 2002, Aitor Moreno joined Vicomtech-IK4 (http://www.vicomtech.org) as a full time researcher. He is involved in several regional, national and international projects focusing mainly on Computer Graphics, VR and interactive simulations. He has published several papers in international conferences and journals. His main interests are Computer Graphics, Simulation, Virtual Reality and advanced interaction techniques (Human-Computer Interaction).

Author's personal copy

Dr Luis Kabongo is researcher in the Biomedical Applications Department at Vicomtech-IK4 (Spain) since May 2008 after obtaining his Computer Science PhD in the University of Bordeaux 1 (France). He is also holding an Applied Cognitive Science Masters Degree which led him to work into a Medical Imaging laboratory and a Medical Devices development company around the city of Bordeaux. He participated there to a project of visualisation and therapy planning platform using a new technology of cancer treatment with focalised ultrasound heating technique. His recent activities are the management and execution of applied research projects dealing with medical imaging related technologies like visualization, processing, human computer interfaces and high performance distributed computation.

Alejandro García-Alonso received on 1990 a Ph.D. degree in mechanical engineering from the University of Navarre. From 1983 to 1995 he contributed to projects developed at CEIT (Spain) for different companies, among them the European Space Agency and Mechanical Dynamics (Ann Arbor, MI). Since 1996 he is an associate professor of computer graphics at the UPV. His research interests include computer graphics, geometrical computation, and haptic systems.