CEIG - Spanish Computer Graphics Conference (2016)
Alejandro García-Alonso and Belen Masia (Editors)

Deployment of volume rendering interactive visualizations in Web platforms with intersected 3D geometry

A. Arbelaiz[1], A. Moreno[1] and L. Kabongo[1]

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

Abstract

Volume rendering is a visualization technique focused in volumetric datasets, which requires a great amount of computational power and memory resources. This situation is even more severe when HTML5 and WebGL implementations are used in ubiquitous platforms to render the virtual scene. This paper presents a WebGL based volume rendering algorithm to support 3D geometries (provided as a set of 3D triangles) inside the volumetric datasets. Implementation details to achieve interactive rates along the volume rendering methods are also discussed.

Categories and Subject Descriptors (according to ACM CCS): I.3.0 [Computer Graphics]: General—I.3.3 [Computer Graphics]: Picture/Image Generation—Display algorithms

1. Introduction

Volumetric datasets are widely used in the medical field. A large variety of medical procedures count on these datasets to provide accurate and useful evaluation of the patients' health. Volumetric datasets are acquired via CT or MRI scans, producing a set of 2D slices. Volume rendering techniques were a milestone in the medical field, as these methods were used to obtain 3D visualizations from the set of 2D slices.

The integration of 3D polygonal elements in volume rendering is a key functionality, specially in the medical field, but also important in others fields like the simulation of engineering processes (thermal combustions). This paper presents an interactive volume rendering algorithm to support 3D geometries (provided as a set of 3D triangles) inside the volume within the constraints of the WebGL API.

The Web as a 3D interactive visualization platform presents restrictions as it tries to cover a wide range of devices maintaining cross-platform support from mobile devices to desktop PCs. For volume rendering, challenges such as, the lack of 3D and depth textures need to be addressed.

The paper is organized as follows, first we present the related work at Section 2. Section 3 describes the modifications needed to allow the coexistence of 3D geometries and volumetric data. Finally, in Section 4 some conclusions and future work are discussed.

2. Related Work

Volume rendering visualizations are being used in various fields to visualize volumetric data. As a consequence, several volume rendering algorithms have been researched over the years. Hadwiger et al. [HKRs*06] made a survey of real-time volume rendering graphics.

This paper is centered in the ray-casting Direct Volume Rendering (DVR) approach. DVR approaches create a rendering directly fetching the volume data. Ray-casting was first introduced using the CPU by Kajiya and Von Herzen [KVH84]. Nowadays, due to the parallel nature of this technique, it has been adapted to work with the GPU. Krüger and Westermann [KW03] presented a multi-pass hardware accelerated approach.

In despite of the efforts made to improve the visualization algorithms and techniques, the combination of highly specialized software and hardware products has made difficult to share volume visualizations across different devices. In contrast, the Web postulates as an ubiquitous deployment platform for volumetric data thanks to GPU accelerated graphics available through the WebGL API.

Based on the Web a few have contributed to the WebGL based ray-casting. The first multi-pass method was introduced by Congote et al. [CSK*11]. Later Mobeen et al. [MF12] presented a single ray-casting approach. Noguera and Jiménez [NJ12] have also addressed the limitations of volume data size of these approaches. The presented work extends these efforts exploring the possibilities of the coexistence of 3D triangular meshes along volume data within the constraints of the Web platform.

DOI: 10.2312/ceig.20161312

2.1. Volume rendering ray-casting algorithm

The ray-casting technique consists in the generation of rays from the camera position which traverse the volumetric dataset mimicking the light absorption and emission physical model. For each pixel of the image, a ray is casted into the scene, which contains a geometric 3D cube. This cube acts as the container of the volumetric 3D dataset. Each ray traverses the cube, sampling the volume data at equi-distant intervals. At each sampling interval, a scalar value is obtained. This operation is usually done by re-sampling the volume data with trilinear interpolation. This calculated value is accumulated along the ray using *alpha blending* in front-to-back or back-to-front order.

The obtained scalar value at each sample interval can be mapped to a given color and opacity by providing a look up table (transfer function) or alternative methods to alter the accumulation composition and in this way, give color or enhance characteristics in the volume data. When the ray finishes the bounding box traversal, the accumulated color and opacity is set to the pixel from which the ray has been originally generated from.

The pseudo-code shown in Listing 1 summarizes the ray-casting algorithm that can be found in the literature.

Listing 1: Ray-casting pseudo-algorithm

```
For each pixel in the screen
  Initialize number of steps S
  Compute the ray_pos, ray_direction
                AND maximun distance D
  Compute interval step s
  For i=0;i<S;i=i+1
    Interpolate sample at current ray_pos
    Compute color with transfer function
    Accumulate color
    Accumulate opacity α
    If α >= 1.0 OR ray_pos > bounding box
        break;
    Increment ray_pos with s
  End For
End For
```

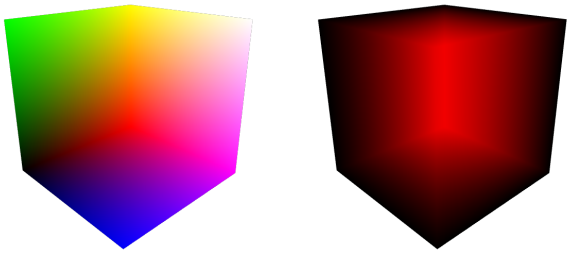
2.2. WebGL volume rendering

The WebGL multi-pass approach was presented by Congote et al. [CSK*11]. In this method, the ray-casting is performed in two rendering passes.

First pass (Back cube coordinates): The back faces of the cube are rendered in a separate Frame Buffer Object (FBO). In this pass, the 3D texture coordinates are assigned as a color on each vertex of the cube. In the rasterization process, the interpolated per-vertex colors represent the volume data 3D texture coordinates.

Second pass (Ray direction and traversal): The 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 color of the front faces and the back faces coordinates obtained in the previous pass, the ray direction and length is computed. Finally, the ray traversal samples the volume data with the method presented in Section 2.3 and using the ray-casting algorithm described in Listing 1.

The pipeline of this approach is depicted in Figure 1. The second pass calculates the ray direction and actually performs the ray traversal.



(a) Ray exit points (b) Ray direction

Figure 1: WebGL ray-casting multi-pass approach. a) First pass, ray termination coordinates encoded as RGB. b) Second pass, ray direction (back-front) as RGB.

2.3. Image texture atlas

The biggest caveat of WebGL in relation to desktop volume rendering techniques is the lack of support for 3D textures. Congote et al. [CSK*11] proposed a workaround to allow sampling the 3D volume data using a single 2D texture. Volume data is usually composed by a set of 2D slices (2D images) stacked in the Z axis. The slices can be arranged as a matrix configuration in the same 2D plane or *atlas*.

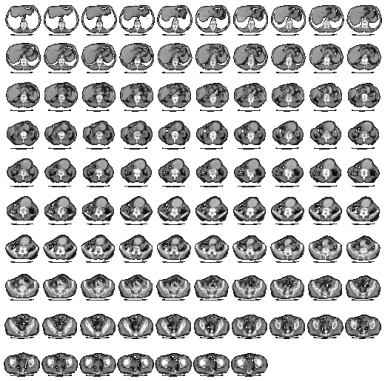


Figure 2: Example of an *image texture atlas* of the *aorta* dataset.

Figure 2 shows an example of a volumetric dataset represented as a 2D texture atlas. The stack of slices have been converted into an atlas representation which allows to store the volume data as a 2D texture. The ray-casting fragment shader will use this 2D texture to sample the volumetric dataset. Using a *transfer function*, the scalar values obtained from the volume data are mapped with colors, an example with the *aorta* dataset will be shown in Figure 3e

3. Intersection with 3D geometry

Volume rendering can be considered as a specific rendering method that focuses mainly in volumetric data. But in some cases, it is of interest to render volumetric and geometrical data together in the same scene. For example, in medicine flow streamlines are typically visualized combined with MRI scan data [SAG*14]. Also, medical surgical training simulations require the interaction between 3D modeled objects and real volumetric data in order to simulate the operations with the use of haptic devices [VFW12, XSH*16]. This section describes additional procedures that must be performed in the ray-casting algorithm to support the integration of 3D geometry and 3D volumetric datasets.

Volumetric data can be considered as semi-transparent objects, and therefore, there are two cases to take into account when rendering both 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 implemented to support the rendering of 3D objects inside or intersecting the volume. The following subsections present solutions to overcome these two cases. The blending process is explained in Section 3.2 and the intersection case will be presented in Section 3.3.

3.1. Multi-pass rendering

To support the rendering of 3D polygonal meshes and volumetric datasets, the presented ray-casting method at Section 2.1 must be modified with additional rendering passes. These passes are required to gather additional information needed during the ray casting traversal in the last pass of the volume ray-casting.

For the rendering of the complete scene, when mixing opaque and transparent objects, the order of which objects are rendered is important. Similarly to the traditional Computer Graphics techniques that render transparent and opaque polygonal meshes, to render volumes and 3D opaque meshes they must be sorted in z-depth order and rendered from back to front.

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

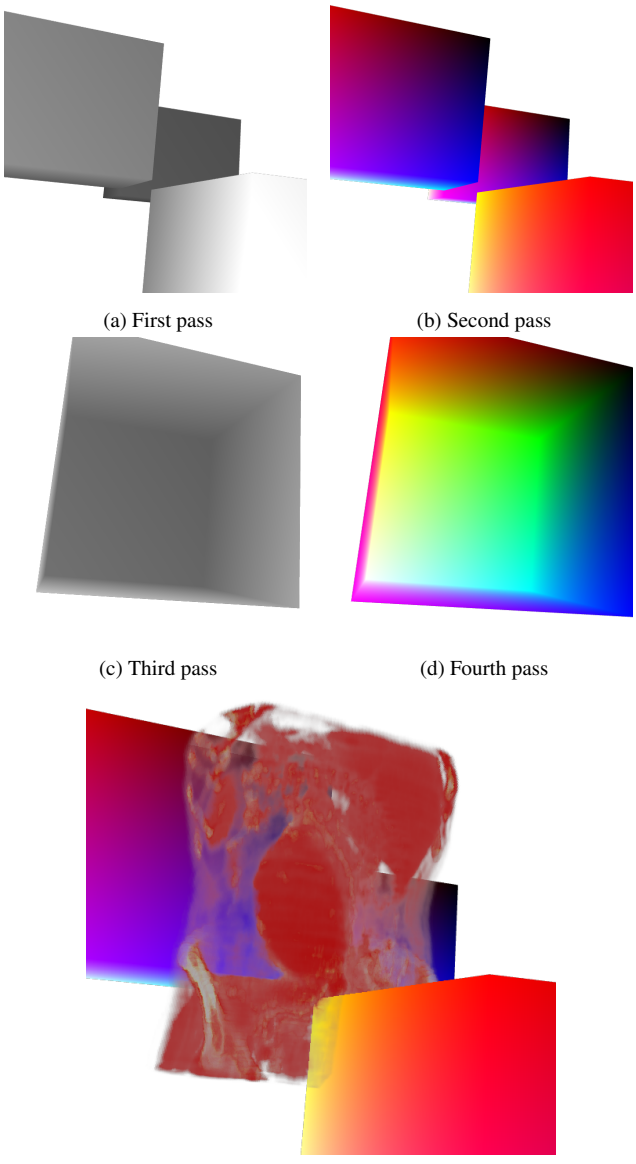
First pass (Depth pass): In the first pass, the depth of all the 3D meshes in the scene are rendered into a Frame Buffer Object (FBO). A simple shader is executed per 3D object in which the fragment shader outputs the current depth encoded in the color channels as RGBA.

Second pass (Color of surface objects): In the second pass, all the 3D meshes are rendered into a separate FBO to store their color information.

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

Fourth pass (Back coordinates of the cube): In this pass, the back faces of the volume bounding box are rendered into a separate FBO. This is same as the first pass in Section 2.2. As result, the the output color components are the exit point of the rays for the ray-casting algorithm.

Fifth pass (Ray traversal and 3D surfaces): In this pass, 3D objects are rendered in the screen along volume objects computing the ray-casting traversal. Objects must be rendered sorted in Z in back-to-front order. In the fragment shader rays are casted towards the volume accumulating color and opacity to finally be rendered in the screen buffer. In this pass, all the outputs of the previous passes are 2D texture inputs for the ray-casting fragment shader. The additional computation applied in the ray-casting shader is explained in Section 3.3.



(a) First pass (b) Second pass (c) Third pass (d) Fourth pass (e) Fifth pass

Figure 3: Multi-pass ray-casting pipeline to render volume and 3D geometries.

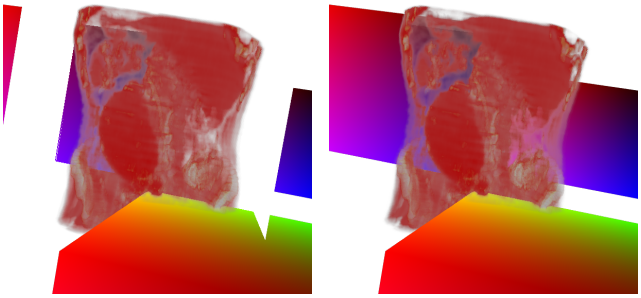
As an immediate conclusion of the presented passes, the objects

must be rendered in certain order to correctly obtain the final composition. Additionally, a downside of this multi-pass approach is the number of the FBO required.

3.2. Blending

As stated in the multi-pass rendering approach presented in Section 2.2, 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 partially contributes to the final rendering output. This step is traditionally known as *alpha blending*.

When the ray-casting shaders are executed in the final rendering pass, the color and alpha output of the computed pixels are being rendered in the screen. For our approach, we have used a back-to-front blending composition.



(a) Blend off (b) Blend on

Figure 4: Alpha blending of volumetric objects and 3D geometries. a) Rendering the volume with blending disabled. b) Front-to-back alpha blending enabled.

Figure 4 shows that the alpha blending is necessary to render correctly any object behind the volume object. The following code shows how to enable blending and to perform a back-to-front composition using WebGL directives.

```
gl.enable(gl.BLEND);
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```

The back-to-front blending is performed only in the fifth pass. For simplicity we have only considered opaque 3D objects in our experiments. Therefore, we are assuming that there are not transparent 3D polygonal objects in the scene. But minor adjustments of the presented methodology can be implemented to support 3D transparent objects.

3.3. Ray-casting with 3D geometry

The ray-casting fragment shader must be modified to support blending of the 3D meshes and the volume. In this section, these necessary additions are presented as GLSL code samples.

The first problem we have to overcome 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 there is a 3D object or not.

To achieve this goal, we need to access the depth data produced by passes 1 and 3 (see Section 3.1) 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 no support for *Float* textures. An extension has been provided to WebGL 1.0 in order to support depth textures, but only implemented by some browsers. As a fallback solution to all WebGL compatible devices packing and unpacking functions can be used. The depth information is packed as RGBA in the passes 1 and 3 and this RGBA information is then unpacked to a float 24-bit value in the fifth pass using the following GLSL code:

```
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
}

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));
}
```

At the beginning, parameters are initialized per pixel fragment to initialize the ray-casting: *ray_origin, ray_direction, ray_step...* Additionally, depth data from previous passes is also fetched to initialize the ray depth step in the same space as the scene objects. The following GLSL code stores the depth information in independent variables that will be used in the final comparison:

```
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)/n_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 are iteratively incremented.

The depth test compares the depth of the 3D object, including its thickness, and the z position of the ray-casting (*ray_depth*). If the 3D object is hit, the accumulated value is updated with the color of the object and the corresponding z position is updated. The following GLSL code shows that procedure and that the ray-casting loop is stopped with the *break* instruction since an opaque object has been hit:

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

```
  break;
}
```

The ray-casting algorithm can finish in different ways:

- The ray traverses the volume completely
- The ray fills the alpha value during the ray traversal
- The ray hits an opaque object

The first case uses the traditional ray-casting algorithm, but the possibility of having a 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 color and depth is returned as the color of pixel in the final rendered image. The third case is when the ray hits a 3D object inside the volume. The scenarios are depicted in the following subsections. The first one addresses the case of a ray traversing the volume completely. The second subsection addresses the early termination techniques used to detect if a ray has hit a 3D object or the accumulated opacity value has reached an opaque value and consequently, abort the ray-casting.

3.3.1. Complete ray traversal, accumulated alpha less than 1

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

3.3.2. Early ray termination, accumulated alpha equal 1

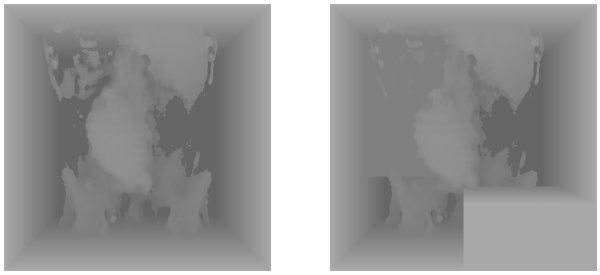
A common ray-casting acceleration technique is known as *early ray termination* (see Krüger et al. [KW03]). In this technique the ray traversal is interrupted when the accumulated opacity reaches an opaque value (accumulated alpha equal 1.0) before the ray gets out of the volume bounding box boundaries.

The *early ray termination* checks for the state of the accumulated opacity inside the ray traversal loop before the computing next ray step. The following GLSL code 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*.

```
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 ray traversal. As previously stated in this section, the essence of this technique is also being applied with 3D geometries inside the volume. When the ray intersects the 3D mesh, the accumulated color and opacity is blended with the 3D geometry surface. As a result the accumulated opacity reaches a totally opaque value, and thus, the ray is terminated.

Figure 5 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 closeness of the 3D object to the camera.



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

Figure 5: 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.

4. Results and discussion

The presented WebGL implementation has been tested in desktop and mobile devices. Figure 6 shows how 3D colored and textured boxes can be integrated with volumetric datasets.

Figure 7 shows different situations regarding the positions of the volume and the boxes. There are pixels that corresponds to a box placed before, behind and in the middle of the volume. All these situations produce visual renderings that are visually correct.

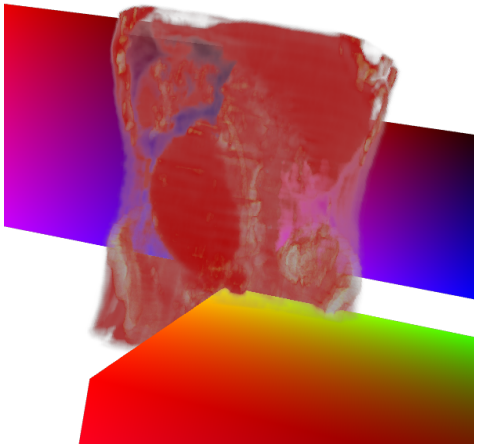
The implementation of the GLSL code is targeting WebGL 1.0. But WebGL 1.0 provides some official extensions that can be used to produce better results and to simplify the GLSL code.

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 3.3 can be replaced by a direct access to the depth texture.

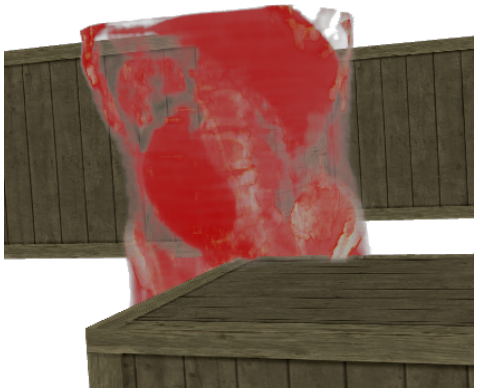
The *WebGL EXT_frag_depth* extension provides the possibility of modifying the depth value in the fragment shader. Using this extension provides more flexibility in the depth buffer reading, testing and writing. Ultimately, it could lead to a reduction in the number of passes in the presented methodology.

The *WebGL_draw_buffers* extension provides multiple color buffers and color 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.

In the future, a non-constant step size in the ray-casting algorithm will be researched. The introduction of such modification might require additional passes of the geometry and the volume in order to calculate the optimal step size for each pixel. The impact of this modification has to be studied, specially in the quality of the results and the impact in the rendering performance.

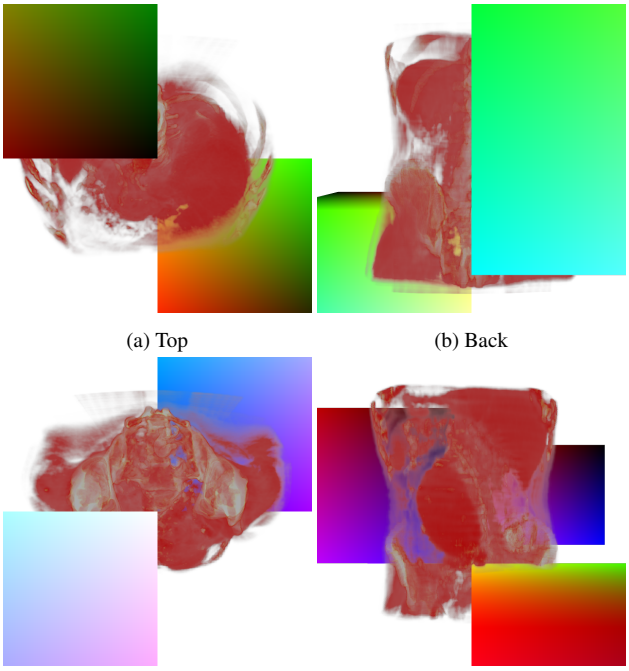


(a) Per vertex color



(b) Texture mapping

Figure 6: WebGL volume rendering of volumetric data and 3D geometry. a) 3D box geometries with per vertex color. b) 3D box geometries with textures.



(a) Top (b) Back

(c) Bottom (d) Front

Figure 7: Volume rendering and 3D geometry intersection from various views using the aorta dataset.

References

[CSK*11] CONGOTE J., SEGURA A., KABONGO L., MORENO A., POSADA J., RUIZ O.: Interactive visualization of volumetric data with webgl in real-time. In *Proceedings of the 16th International Conference on 3D Web Technology* (New York, NY, USA, 2011), Web3D '11, ACM, pp. 137–146. URL: http://doi.acm.org/10.1145/2010425.2010449, doi:10.1145/2010425.2010449. 1, 2

[HKRs*06] HADWIGER M., KNISS J. M., REZK-SALAMA C., WEISKOPF D., ENGEL K.: *Real-time Volume Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2006. 1

[KVH84] KAJIYA J. T., VON HERZEN B. P.: Ray tracing volume densities. *SIGGRAPH Comput. Graph. 18*, 3 (Jan. 1984), 165–174. URL: http://doi.acm.org/10.1145/964965.808594, doi:10.1145/964965.808594. 1

[KW03] KRUGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)* (Washington, DC, USA, 2003), VIS '03, IEEE Computer Society, pp. 38–. URL: http://dx.doi.org/10.1109/VIS.2003.10001, doi:10.1109/VIS.2003.10001. 1, 5

[MF12] MOBEEN M. M., FENG L.: High-performance volume rendering on the ubiquitous webgl platform. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems* (Washington, DC, USA, 2012), HPCC '12, IEEE Computer Society, pp. 381–388. URL: http://dx.doi.org/10.1109/HPCC.2012.58, doi:10.1109/HPCC.2012.58. 1

[NJ12] NOGUERA J. M., JIMÉNEZ J.-R.: Visualization of very large 3d volumes on mobile devices and webgl. In *20th WSCG International Conference on Computer Graphics, Visualization and Computer Vision 2012* (2012). 1

[SAG*14] STANKOVIC Z., ALLEN B. D., GARCIA J., JARVIS K. B., MARKL M.: 4d flow imaging with mri. *Cardiovascular Diagnosis and Therapy 4*, 2 (04 2014), 173–192. URL: http://www.ncbi.nlm.nih.gov/pmc/articles/PMC3996243/, doi:10.3978/j.issn.2223-3652.2014.01.02. 3

[VFW12] VLASOV R., FRIESE K.-I., WOLTER F.-E.: Haptic rendering of volume data with collision determination guarantee using ray casting and implicit surface representation. In *Cyberworlds (CW), 2012 International Conference on* (2012), IEEE, pp. 91–98. 3

[XSH*16] XU R., SUGIYAMA A., HASEGAWA K., TAGAWA K., TANAKA S., TANAKA H. T.: *Innovation in Medicine and Healthcare 2015*. Springer International Publishing, Cham, 2016, ch. Remote Transparent Visualization of Surface-Volume Fused Data to Support Network-Based Laparoscopic Surgery Simulation, pp. 345–352. URL: http://dx.doi.org/10.1007/978-3-319-23024-5_31, doi:10.1007/978-3-319-23024-5_31. 3