Real-time ray casting of volumetric data

Žiga Lesar
Faculty of Computer and Information Science, University of Ljubljana, Slovenia
Email: ziga.lesar@lgm.fri.uni-lj.si

***Abstract*—In this paper we present acceleration structures and techniques for real-time ray casting of large volumetric data sets, such as those obtained by CT or MRI scans. The techniques used include adaptive sampling and sparse casting. To improve rendering quality we use the regula falsi method and Monte-Carlo ambient occlusion estimation. To enhance the final rendering we use visual effects - screen-space ambient occlusion and depth of field. The algorithms have been parallelized with OpenCL. We compare the results obtained with different methods - isosurface extraction, maximum intensity projection and alpha compositing. We tested our methods on medical data sets, specifically angiograms.**

I. Introduction

Volume rendering is a term for different approaches used to visualize volumetric data - 3D discretely sampled images usually obtained by medical imaging or numerical simulation. A number of different methods are used for the job, including splatting [16], texture-based rendering [15], conversion to triangular meshes (e.g. marching cubes [9]) etc. One of the oldest and most widely-used is ray casting, introduced in 1982 by Scott Roth [13] and applied to volume rendering in 1988 by Marc Levoy [7]. All of these methods rely on more or less direct evaluation of the rendering integral, described in Section III-E. Some of them strive for visual quality of the rendered image (ray casting), while others aim for speed and performance (splatting, marching cubes). Although ray casting is a computationally intensive method, it can be easily parallelized for execution on the GPU, allowing us to render high-quality images relatively quickly. With the widespread use of medical imaging as support in making diagnoses came the need for fast and quality visualization of acquired data. For this purpose many acceleration structures have been developed over the years, most of them exploiting object-space and screen-space coherence, as well as neglecting irrelevant information.

II. Related work

One of the methods presented in this paper is isosurface extraction. Our implementation is based on Hart's approach [3], where he used ray casting methods to render implicit surfaces. For intersection refinement numerical iterative methods, such as regula falsi, were used. Sphere tracing [4] was used to efficiently skip empty spaces. Although fast, it requires complex preprocessing to compute the distance transform. This is the main reason why more recent applications use octrees to encode data and serve as an acceleration structure. A common approach is to use sparse octrees [6] to reduce memory consumption. A fast parametric algorithm for octree traversal is described in [12].

978-1-4799-8569-2/15/$31.00

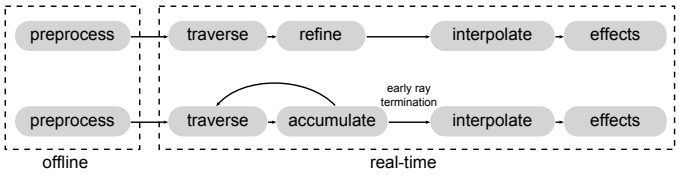

Figure 1. A diagram showing how the methods are interconnected to form a complete rendering framework. The upper half shows the process for isosurface reconstruction, and the bottom half shows how the case with transparency is handled. The preprocessing step is done offline, then the data is offloaded onto the graphics card, where it is rendered in real time. In the traversal step we traverse the octree with sparse casting and find an intersection, which is refined with the regula falsi method. The resulting rendering is interpolated and additional rays are cast where needed. Visual effects are applied at the end. In the case of transparency, the traversal is repeated using the kd-restart algorithm.

Often isosurface rendering is insufficient for displaying relevant information. Transparency-based approaches, described in [11], may be used to map volume values to optical properties. These properties are then used in optical models to accurately simulate light transport in transparent materials. Different methods have been developed to speed up the process, including early ray termination [5] and sparse casting [8].

III. Methods

The following sections describe the acceleration structures and methods used to speed up the rendering process. Figure 1 shows how they are interconnected to form a complete rendering framework. The methods are suitable for execution on a graphics card.

A. Sampling

Volumetric data can be thought of as sampling of a continuous signal $\Phi : \mathbb{R}^3 \to \mathbb{R}$. The signal has to be band-limited prior to discretization to avoid aliasing artifacts. Usually we cannot affect the sampling process, but we can make the best use of the acquired data by correctly reconstructing the original signal from the samples.

With a high enough sampling rate - the Nyquist rate - we are able to reconstruct the original signal exactly by using a 3D analogue of a sinc filter. This is the fundamental process of Whittaker-Shannon interpolation. However, the interpolation formula requires taking all the samples into consideration, since the sinc filter has infinite extent and is non-zero almost everywhere. This shows to be computationally too expensive. In practice we use simple approximations of the sinc filter to achieve *good-enough* results. Box filter (nearest neighbour function) and tent filter (trilinear interpolation) are used in our implementation, as it turns out they are a good trade-off between speed and resulting image quality. The three

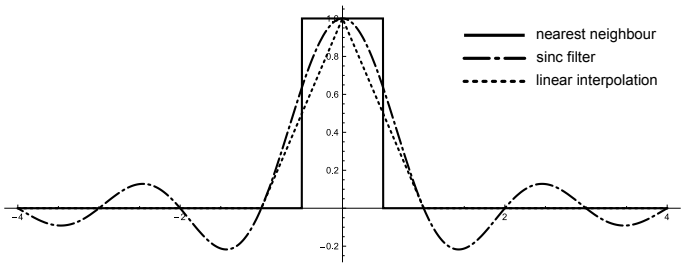

Figure 2. Reconstruction filters, in order of increasing reconstruction accuracy: nearest neighbour, linear interpolation and the sinc filter. The filters shown are for the 1D case, but 3D analogues can be obtained with a tensor product.

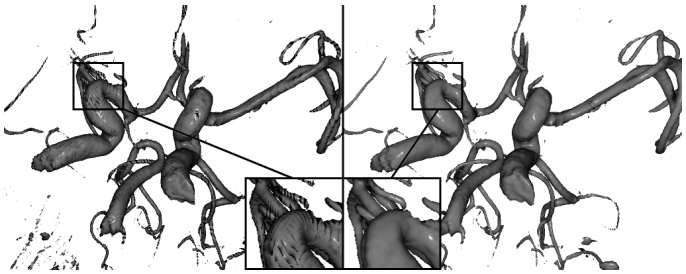

Figure 3. An example of a rendered image with (right) and without (left) intersection point refinement. Regula falsi with 5 iterations was used.

reconstruction filters are shown in Figure 2. For even smoother representation of the volumetric structures we preprocess the volume with a Gaussian kernel to smooth out any unwanted features in the data.

In the following text the term *plain sample* is used to denote sampling with nearest neighbour filtering, as this just means reading a single value from a location in the volume data array. In the process of trilinear interpolation sampling we therefore need 8 plain samples. Because surface normals are often used for shading, we also need to sample gradients $\nabla\Phi(\mathbf{p})$. These are estimated with the central difference formula that takes 6 plain samples into account. Used with trilinear interpolation and eliminating duplicate plain samples this makes for 32 plain samples, which makes normal sampling a very expensive process. Gradients could be estimated with a forward difference formula with fewer plain samples, but it can be shown that this estimation is inferior in terms of convergence rate.

B. Ray casting

Equipped with volume data and signal reconstruction methods we are able to recover signal values from arbitrary points in space. However, we have to project those values onto the projection plane. In our implementation we used the ray casting method for this purpose. To project a volume onto the projection plane we need a camera object $(\mathbf{P}, O, \varphi, \alpha)$ which holds a position $\mathbf{P}$ and orientation O in space. Field of view φ and aspect ratio α are used to define the region on the projection plane to be displayed on the screen. We use this information to cast rays into the scene, sampling the volumetric data along the way. These samples may be used to find μ-isosurface intersections - the set of points $\mathbf{p} \in \mathbb{R}^3$, for which $\Phi(\mathbf{p}) = \mu$. The value μ is called isosurface value and can be set at runtime by the user. Isosurface extraction problem is reduced to a one-dimensional problem of finding zero-crossings when we use parametrized light rays. Light rays are parametrized in our implementation as

$$\mathbf{r}(\lambda) = \mathbf{P} + \lambda\hat{\mathbf{D}}(i,j). \tag{1}$$

The vector $\hat{\mathbf{D}}(i,j)$ is the unit directional vector of the ray and can be computed from screen coordinates (i,j). Since it is unit length, the parameter λ represents the depth and can be directly written to the depth image after the intersection is found. The parametrization can be substituted into the signal representation to obtain

$$\Psi(\lambda) \stackrel{\text{def.}}{=\!=} (\Phi \circ \mathbf{r})(\lambda) - \mu = 0, \tag{2}$$

the function used in the intersection refinement step.

While we sample the volume at discrete steps of the parameter λ, we check whether the sampled value exceeds the set threshold μ. When this condition is satisfied, we know that the isosurface intersection point lies between the last two sampling positions and we can stop the ray marching procedure. However, the intersection point is vaguely defined and can be further refined using conventional iterative numerical methods for finding zero-crossings. We have chosen regula falsi, since its assumption of linearity of the function makes it an ideal complement to our trilinear interpolation sampling choice. It is symbolically described in equation (3), where we labeled the interval containing the intersection point with (a_k, b_k). The other (albeit less suitable) choice was Newton's method, but it could diverge with imprecise initial estimates. Application of the method to the isosurface extraction problem is symbolically described in equation (4). Figure 3 shows the difference in image quality with and without intersection point refinement.

$$c_k = b_k - \frac{\Psi(b_k)(b_k - a_k)}{\Psi(b_k) - \Psi(a_k)} \tag{3}$$

$$\lambda_{k+1} = \lambda_k - \frac{(\Phi \circ \mathbf{r})(\lambda_k) - \mu}{(\nabla\Phi \circ \mathbf{r})(\lambda_k) \cdot \hat{\mathbf{D}}(i,j)} \tag{4}$$

After the intersection point is found it can be shaded using any one of the standard illumination models. We have implemented the Phong illumination model, as it is easy to compute and is relatively computationally inexpensive. A gradient estimate must still be obtained from the intersection point of every ray to be used in the shading process.

C. Octree

Since ray casting is computationally a very intensive process many acceleration structures and algorithms have been developed over the years. Octrees turned out to be simple and efficient, but still easy to implement. An octree is a space subdivision method which recursively subdivides the volume into 8 smaller parts by 3 subdivision planes. Level k of the octree with n elements therefore contains $\frac{n}{8^k}$ nodes. Our implementation does not subdivide the volume to the lowest level, as the levels use exponentially more memory. We decided to build the octree structure down to level 3 and traverse the

rest of the structure linearly. This way the memory footprint is insignificant compared to that of the volumetric data array. Tree nodes can optionally contain additional information of the volume part. In our implementation we store minimum and maximum values, which are then used during traversal to skip the parts of the volume which do not contain the isovalue. Additionally, the octree nodes store the average value of the volume part to speed up the rendering process while using the emission-absorption model, described in Section III-E. To be able to efficiently search and traverse the octree structure we have implemented a full octree, as opposed to a sparse octree. This is also a better choice in the case of volumetric data with high entropy. The volume bounding box used in octree construction is scaled so that each of the three dimensions is a power of 2. This makes octree construction much easier.

The octree structure built from level 3 upward takes up

$$\sum_{k=3}^{\infty} \left(\frac{1}{8}\right)^k = \frac{1}{448} \approx 0.002232 \tag{5}$$

of the memory used by the volumetric data array. As each node contains 3 times as much data as a single volume sample, the ratio is 3 times as large. Assuming that in the worst case the octree is 8 times bigger due to scaling, this makes for about 5% of the memory used by the volumetric data array. Most of the time the actual ratio is a lot lower.

To be able to traverse the tree structure we implemented the algorithm kd-restart [1]. The algorithm finds a point in space with recursive subdivision based on the point's coordinates. When a new point is needed to be located, the process is repeated starting from the root node, hence the name of the algorithm. Striving for visualization speed, this method presents a fairly good trade-off between theoretical and computational complexity.

Since angiography data contains a lot of empty space, octree structures are a very efficient method to speed up the ray traversal procedure, both in terms of space and time complexity.

D. Sparse casting

The main bottleneck of the ray casting algorithm is the number of rays needed to produce the final image. This number increases linearly with each dimension of the final image, making high-definition rendering in real time quite a daunting task. Sparse casting is a method for simple and effective reduction of the number of required light rays. It is an exploitation of screen-space coherence, taking into account the fact that adjacent pixels hold similar colors most of the time, or the color changes gradually. The method consists of two phases. In phase 1 the rays are cast for every k-th pixel on screen and the colors are obtained for those pixels. In phase 2 the colors are interpolated inbetween those pixels. Since this produces somehow blurry images (a side effect caused by interpolation of colors in areas where otherwise sharp features should be preserved), some of the interpolated pixels must be replaced with actual light rays. Those pixels can be found by looking at the magnitude of the local color gradient. Where the magnitude exceeds a certain threshold, the interpolated pixels are replaced with light rays.

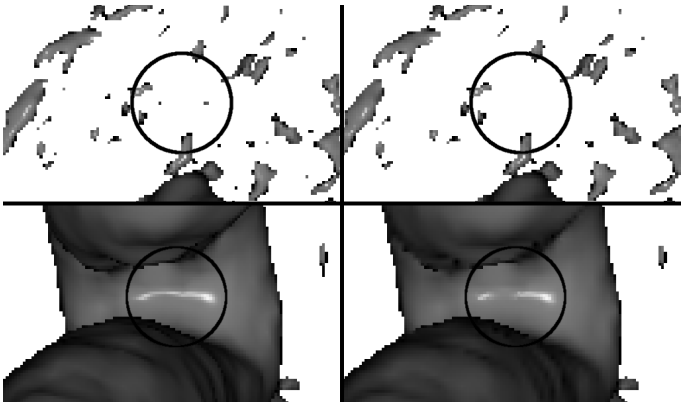

Figure 4. Rendered image with (right) and without (left) sparse casting. Objects not wider than k pixels may be not be visible when using sparse casting, since they lie inbetween the adjacent rays (top). Shading is also affected due to interpolation (bottom).

In theory the method produces a speed-up of $O(k^2)$, but the additional rays in phase 2 and additional processing required during the process significantly reduce its benefits. Another downside of this approach is that we may miss fine features not wider than k pixels when projected onto the screen. This may present a problem in some specific domains where fine precision of the rendering is of great importance, but users are generally not interested in pixel-perfect images.

A simplified method using every second pixel (using $k = 2$) was implemented for the purposes of our application. A comparison of sparse casting with full-scale ray casting can be seen in Figure 4.

E. Emission-absorption model

Apart from isosurface rendering we can also show different volume structures on the same image. We can achieve that by introducing an optical model which simulates transparency. We can think of a transparent volume as a cloud of particles that absorb and emit light, affecting the light intensity I of the light ray passing through it. This interaction can be represented in the form of a differential equation (6) which for every point in space models absorption $\kappa(\lambda)$ and emission $g(\lambda)$ as two continuous processes, where λ denotes position along the ray.

$$\frac{dI(\lambda)}{d\lambda} = g(\lambda) - \kappa(\lambda)I(\lambda) \tag{6}$$

The solution to differential equation (6) is the *volume rendering integral*, shown in equation (7). The value τ used in the integral and defined in equation (8) is known as the *optical depth* and represents a point's visibility as viewed from the camera.

$$I(\lambda) = I_0 e^{-\tau(0,\lambda)} + \int_0^{\lambda} g(t) e^{-\tau(t,\lambda)} dt \tag{7}$$

$$\tau(a,b) = \int_a^b \kappa(u) du \tag{8}$$

For visualization purposes the integral (7) must be discretisized. Approximation using a Riemann sum for integral evaluation is a process known as *alpha compositing* and is described in-depth in [11]. Its accuracy is directly related to step size, which

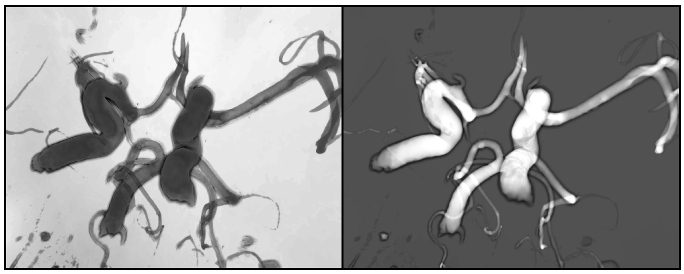

Figure 5. Comparison of different classification gradients (transfer functions) applied to the same volume. The transfer function can be used to enhance different parts of the volume, e.g. vessel walls or the interior of the vessels.

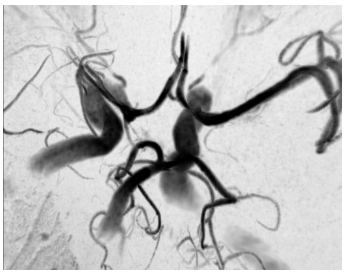

Figure 6. An example rendering using maximum intensity projection. The maximum value for each ray is projected onto the screen using a linear ramp to map intensity to color. The vessels with greater intensities are projected over the vessels with lesser intensities, despite the latter being closer to the camera.

makes this a rather inaccurate method, yet highly efficient for interactive applications. Image quality issues arising from inappropriate step size choices can be overcome by using a better numerical tool, such as Romberg's method for numerical integration or the Runge-Kutta method for solving the associated differential equation. However, the Riemann sum is fast and easy to compute, which makes it a popular choice in real-time implementations. In our application it is implemented with front-to-back compositing, which enables us to employ a technique called early ray termination. Since the samples underlying the parts of the image of high opacity contribute little to the visual quality and accuracy of the final rendering, ray marching can be terminated when a high-enough opacity has been reached.

The optical properties used in the equations can be obtained by a *transfer function*, which maps volume data values to optical properties, such as the absorption and emission coefficients κ and g. This process is known as classification and can be executed efficiently using a precomputed lookup table. Figure 5 shows the resulting image using different transfer functions.

Octrees can be used to help skip empty and homogeneous regions of the volume. Homogeneity is estimated in our application as the difference between the maximum and the minimum value of the octree node in question. If the difference is below some user-specified threshold, the node is treated as homogeneous and is approximated with the average value stored in the same octree node. However, if the transfer function changes rapidly between the node's minimum and maximum values, the approximation may be far-off.

F. Maximum intensity projection

A special case of the emission-absorption model is *maximum intensity projection* (MIP). It is a very fast compositing scheme where we take the maximum value of the function along the viewing ray and project it on the screen using a color, which corresponds to the maximum value. Basically, we are evaluating the function

$$I = c(\max_{\lambda}(\Phi \circ \mathbf{r})(\lambda)). \tag{9}$$

In most cases a simple linear ramp between black and white is the best choice for the function c. Using interpolation for sampling does not contribute greatly to the final image quality, so nearest neighbour sampling is enough to get a good rendering. Optionally we can segment the data - in our implementation we used simple thresholding to hide irrelevant information and low amplitude noise. Compared to other compositing schemes MIP conveys the least ammount of depth and surface topology information, but due to its simplicity and ease of implementation it is still one of the most popular ways to visualize volumetric data. One may think shading could possibly enhance the rendering, but since the projected light intensity is altered during the process, this essentially defeats the purpose of the method, therefore we are left with the limited information the method provides us with. In the case of orthographic projection it is even impossible to distinguish between left or right and front or back, but this may be partially overcome by projecting local maxima only. This modification is known as LMIP and is described in [14]. Nevertheless, MIP may be the perfect method for identifying the regions of the volume with large magnitude differences (e.g. vessel walls), especially in automated processing with computer vision techniques.

G. Visual effects

Visual effects, such as gamma corrention, tone mapping, brightness and contrast adjustments, often enhance the rendering by making it more visually appealing. More complex effects, such as screen-space ambient occlusion and depth of field, which were implemented in our application, may emphasize different features in the image facilitating depth perception and spatial comprehention.

Ambient occlusion is a way of approximating global illumination effects of local surface proximity. In object-space it can be computed by approximating the local accessibility integral, shown in equation (10), for example with a Monte-Carlo estimator. In equation (10), $\hat{N}$ is the surface normal, $\hat{\omega}$ is the solid angle, and $V_{\mathbf{p},\omega}$ is the visibility function, which equals 1 when the point $\mathbf{p}$ is visible from ω, and 0 otherwise.

$$A_{\mathbf{p}} = \frac{1}{\pi}\int_{\Omega} V_{\mathbf{p},\omega}(\hat{N}\cdot\hat{\omega})d\omega \tag{10}$$

Nevertheless, it is still a computationally expensive task to estimate the integral accurately at every intersection point, hence a screen-space version has been developed to reduce complex object-space operations and sampling to simpler queries in image space. The screen-space ambient occlusion (SSAO) algorithm has been developed with speed in mind, with its main

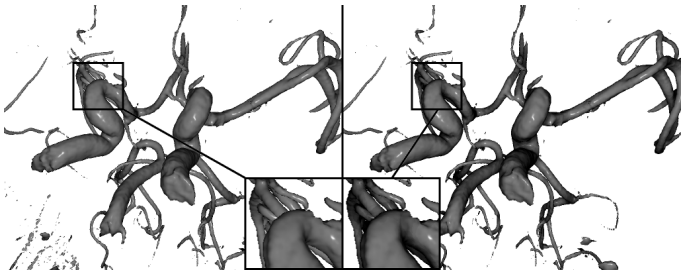

Figure 7. An example rendering with (right) and without (left) screen-space ambient occlusion. Parts of the image which are occluded by the structures in the foreground appear darker. The effect is more noticeable with camera animation.

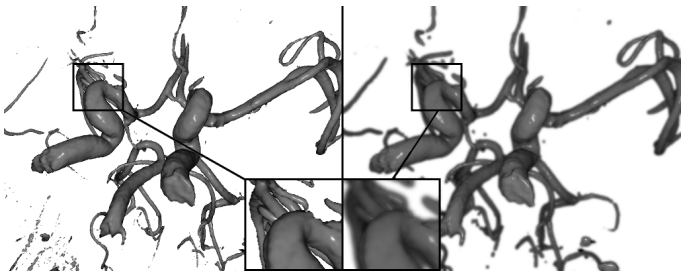

Figure 8. An example rendering with (right) and without (left) depth of field. The structures which are far from the focal distance appear blurred, mimicking the effect of a camera lens. The effect should redirect viewer's attention to focused regions of the rendering.

advantage being independence of scene complexity. It is though directly dependent on the size of the final rendering. It works by estimating the accessibility integral (10) by examining the local neighbourhood of the pixel in question. As the algorithm requires the depth of the samples, a depth image must be provided, which is often the case in deferred rendering approaches. If, according to the depth image, the nearby pixels are closer to the camera than the pixel currently being processed, the pixel shoud appear darker as if it's occluded. To avoid black spots on overlapping objects that are otherwise far appart a threshold for the queries should be provided by the user. An example rendering with the SSAO effect is shown in Figure 7.

Depth of field (DOF) is a technique that further enhances depth perception and spatial comprehension by simulating the lens of a camera. A lens can precisely focus only objects at focal distance from the camera, so other parts of the scene appear blurred in the final rendering. Since a human eye naturally seeks differences in the percieved image, blurred out-of-focus parts of the image should redirect user's attention to focused parts, holding relevant information. Focusing may be additionally emphasized by darkening defocused objects. Furthermore, the focal distance can be dynamically adjusted based on the depth image.

Our implementation uses depth information acquired from the depth image to determine the distace from the focal point. In phase 1 of the algorithm, the rendered image is blurred. The resulting image is then used in phase 2 where it is interpolated with the original (sharp) image with the interpolation factor being the distance from the focal point. Figure 8 shows an example of the effect.

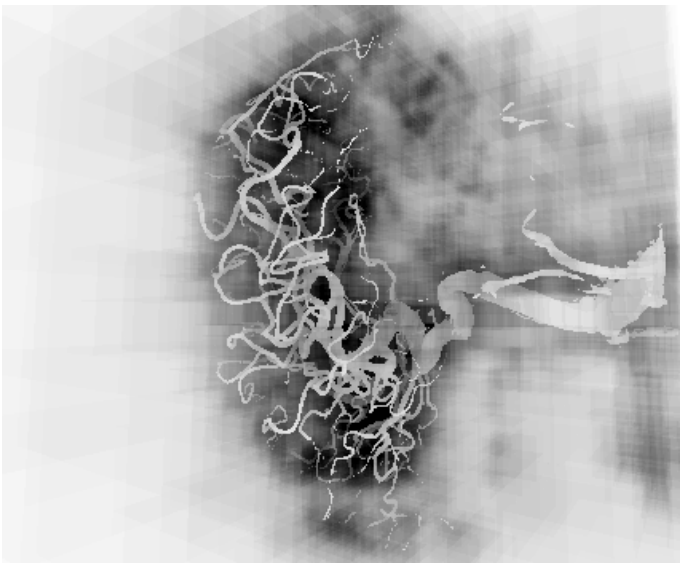

Figure 9. Octree traversal loop iteration count. A darker color means more iterations. Clearly the rays in close proximity of the isosurface require many more iterations than those passing through it. Rectangular artifacts are visible in the figure, as a result of octree traversal.

IV. Results

We have implemented the algorithms described in Section III with Java for the host application and OpenCL for image synthesis. While we are striving for portability, efficiency is still a big issue, especially with the code written in OpenCL which is generally slower than native solutions (e.g. CUDA).

Our implementation is capable of rendering volumes of dimensions up to 512^3 in real time. We achieve speeds of up to 20 frames per second, depending on the volume and the transfer function. Using sparse casting together with an octree and kd-restart traversal we can achieve around 20% speedup in comparison with uniform ray marching. The octree is in fact a major contribution to visualization speed, but traversing the structure takes quite a lot of time and the whole process might slow down the application when there is a lot of geometry to be rendered on the screen. The slowdown is most noticeable when light rays travel close to (but not through) the surface, as this causes the algorithm to traverse the tree down to the lowest levels. This phenomenon can be seen in Figure 9. We are planing on employing a better traversal algorithm in the future, for example the one found in [12], since finding the closest intersection quickly is still the worst bottleneck in our implementation. Intersection point refinement and shading actually take a lot less time than tree traversal, but our implementations of those algorithms are still unoptimized.

We also find visual effects quite visually appealing. The computation takes only a fraction of a second, but still the results are extremely positive. This encourages us to implement a wide variety of effects in the future, hopefully further improving the final rendering in terms of better perception as well as easier customizability.

To assess the performance of our application and the quality of the implementations of individual algorithms we have conducted a performance analysis and evaluation. We have

done two separate experiments: (1) an experiment with a static view, where camera position was fixed for 45 seconds at a resolution of 1024×1024; (2) an experiment with view animation, where the camera was in a circular trajectory around the volume for 100 seconds at a resolution of 512×512. We have used volumetric data from an open on-line data set `http://www.volvis.org/`. The experiments were executed on a machine with two Intel Xeon E5-2620 2.0 GHz processors, 32 GB RAM and an NVIDIA Quadro K5000 4 GB RAM graphics card, with the Microsoft Windows Server 2008 R2 operating system. Average FPS rates for individual tests in this setup are presented in Table I.

Table I
PERFORMANCE ANALYSIS

	dof	sparse	ssao	recon
Static				
with	15.70	14.48	15.64	16.75
without	16.15	16.11	16.20	16.04
Dynamic				
with	14.25	16.84	14.33	15.93
without	14.63	14.23	15.02	14.74

It is evident that adding visual effects screen-space ambient occlusion (*ssao*) and depth of field (*dof*) does not take much additional computational time while improving the final rendering of the image significantly. Sparse casting (*sparse*) seems to be beneficial in most cases, but under certain conditions it may perform poorly (e.g. in the static test). Another great improvement is using a simpler reconstruction filter in regions where accurate reconstruction is not a concern (*recon*). The effect of switching between linear interpolation and nearest neighbour sampling seems to have a positive impact on performance.

V. CONCLUSION AND FUTURE WORK

In this work we presented the possibilities for accelerating the ray casting procedure. GPU execution itself is a major improvement over CPU implementations, taking the parallel nature of ray casting into account. This comes at a price though, as GPUs are not designed for executing complex branching and control flow statements, meaning we have to avoid those as much as possible. Despite these limitations we can still make GPU ray casting efficient by exploiting coherence and neglecting irrelevant information, for example by early ray termination, octrees and sparse casting. While this slightly affects rendering quality it effectively reduces the amount of required samples and operations needed to produce the image.

We are in fact only scratching the surface of what is possible. Future improvements in the form of advanced hierarchical structures (e.g. hashed octrees, [10]) could further minimize the number of iterations in the ray marching loop. We could also use the same hierarchy for adaptive sampling as described in [2]. Screen-space coherence could further be improved by beam optimization [6]. Employing a C-buffer [17] can enable us to benefit from temporal coherence to increase performance of ray casting with smooth camera animations. Better numerical methods for integral evaluation and differential equation solving would directly improve image quality in exchange for rendering speed.

Ray casting of volumetric data in real-time therefore remains a difficult problem, requiring complex methods and algorithms to produce solutions capable of quality interactive visualization. With ever-improving graphics hardware we are confident that more advanced illumination techniques will soon be suitable for real-time applications even on low-end hardware. It is in fact already possible to achieve impressive results with top-edge hardware.

ACKNOWLEDGMENT

The angiography data sets are courtesy of Özlem Gürvit, Institute for Neuroradiology, Frankfurt, Germany, and are available for download at `http://www.volvis.org/`. We also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Quadro K5000 graphics card for this research.

REFERENCES

[1] Tim Foley and Jeremy Sugerman. KD-tree acceleration structures for a GPU raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, volume 45, pages 15–22, 2005.

[2] Enrico Gobbetti, Fabio Marton, and José Antonio Iglesias Guitián. A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806, 2008.

[3] John C. Hart. Ray tracing implicit surfaces. In *SIGGRAPH '93 Course Notes: Design, Visualization and Animation of Implicit Surfaces*, volume 1, 1993.

[4] John C. Hart. Sphere tracing: a geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[5] Jens Krüger and Rüdriger Westermann. Acceleration techniques for GPU-based volume rendering. In *Proceedings of the 14th IEEE Visualization 2003*, pages 287–292, 2003.

[6] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8):1048–1059, 2011.

[7] Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):22–37, 1988.

[8] Marc Levoy. Volume rendering by adaptive refinement. *The Visual Computer*, 6(1):2–7, 1990.

[9] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987.

[10] Daniel Madeira, Esteban Clua, and Thomas Lewiner. GPU octrees and optimized search. In *Proceedings of the 8th Brazilian Symposium on Games and Digital Entertainment*, pages 73–76, 2009.

[11] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[12] Jorge Revelles, Carlos Ureña, and Miguel Lastra. An efficient parametric algorithm for octree traversal. In *Journal of WSCG*, volume 8, pages 212–219, 2000.

[13] Scott D. Roth. Ray casting for modeling solids. *Computer Graphics and Image Processing*, 18(2):109–144, February 1982.

[14] Yoshinobu Sato, Nobuyuki Shiraga, Shin Nakajima, Shinichi Tamura, and Ron Kikinis. Local maximum intensity projection (LMIP): A new rendering method for vascular visualization. *Journal of Computer Assisted Tomography*, 22(6):912–917, 1998.

[15] Allen Van Gelder and Kwansik Kim. Direct volume rendering with shading via three-dimensional textures. In *Proceedings of the 1996 symposium on Volume visualization*, pages 23–30. ACM, 1996.

[16] Lee Alan Westover. *Splatting: a parallel, feed-forward volume rendering algorithm*. Doktorska disertacija, University of North Carolina, 1991.

[17] Ilmi Yoon, Joe Demers, Taeyong Kim, and Ulrich Neumann. Accelerating volume visualization by exploiting temporal coherence. In *Proceedings of IEEE Visualization*, pages 21–24, 1997.