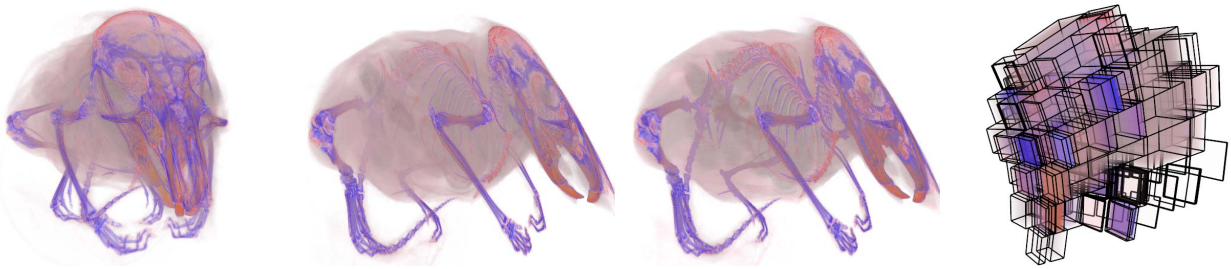
Explorable Volumetric Depth Images from Raycasting

Steffen Frey, Filip Sadlo, and Thomas Ertl
Visualization Research Institute, University of Stuttgart, Germany
Email: {steffen.frey, filip.sadlo, thomas.ertl}@visus.uni-stuttgart.de



(a) VDI generation, 280ms overhead (b) Raycasting takes 950ms (c) VDI rendering takes 110ms (d) VDI frustums (low res.)

Fig. 1: (a) Generation of a 512^2 Volumetric Depth Image (VDI) of the Mouse data set ($1024 \times 1024 \times 975$ at 16 bit), resulting in approximately 2M supersegments, at small overhead w.r.t. the original raycasting. (b) Raycasting from different view. (c) Same view as (b) using VDI rendering is faster and requires less memory. (d) Illustration of frustum-based VDI rendering.

***Abstract*—View-dependent image-based rendering techniques have become increasingly popular as they combine the high quality of images with the explorability of interactive techniques. However, in the context of volume rendering, previous approaches suffer from various shortcomings, including the limitation to surfaces, expensive generation, and insufficient occlusion and motion parallax impairing depth perception. In this paper, we propose Volumetric Depth Images (VDI) to overcome these issues for view-dependent volume visualization by an extension of the Layered Depth Image (LDI) approach. Instead of only saving for each view ray of one camera configuration the depth and color values for a set of surfaces, as in LDIs, VDIs store so-called supersegments, each consisting of a depth range as well as composited color and opacity. This compact representation is independent from the structure of the original data and can be generated by slight modification of raycasters with very low overhead. VDIs can be rendered efficiently at high quality with arbitrary camera configurations by means of proxy frustum geometry and an efficient depth ordering scheme. When viewing the scene from the initial view point, VDIs produce results identical to the original raycasting. As demonstrated by means of a prototype implementation and data from different fields, our approach can be useful for preview rendering and a-priori analysis in in-situ contexts among others.**

***Keywords*-Volume Raycasting; Explorable Image; Preview and Offline Techniques;**

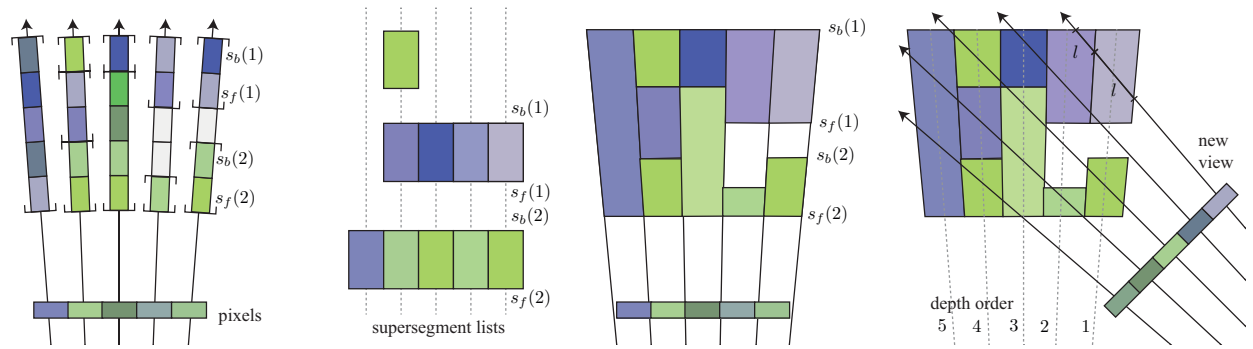
I. Introduction

An explorable image is a compact intermediate view-dependent representation of data that allows for deferred interaction. It has many different applications, like providing preview rendering in a local or remote rendering setting in the context of an expensive rendering process or low bandwidth, or lowering the overall load on a render server. In contrast to one plain image, an explorable image (e.g., a Layered Depth Image (LDI) [1]) allows for interaction until the next image is available. Another area of application stems from large-scale, high-fidelity simulations by making use of the high-performance facilities at a supercomputing center to transform the data into a compact explorable image. For instance, this could be used in combination with techniques suggesting informative views in volume visualization (e.g., [2]).

In this work, we propose Volumetric Depth Images (VDI), a generalization of LDIs for volume data. It allows one to capture a volume from a certain camera configuration in a fast and efficient way for subsequent rendering (see Fig. 2 for an overview). A VDI can easily be generated during volumetric raycasting by partitioning the samples along rays (Fig. 2a) according to their similarity, providing the additional possibility to skip "empty" regions. These partitions can then be stored as lists of so-called supersegments containing the bounding depth pair (s_f, s_b) and partial color accumulation values (Fig. 2b). Each (quadrilateral) pixel in the image plane with the used camera parameters forms a pyramid. The supersegments can then be rendered as frustums of these pyramids (Fig. 2c). Frustums generated by the same ray are conceptually grouped into a frustum list. Color and opacity are modeled constant within a frustum, and determined during VDI-generation. During rendering, frustum lists are depth-ordered and composited (Fig. 2d). For this, the length l a ray passes through each frustum needs to be considered in order to correctly adjust its opacity contribution.

In particular, we contribute the following:

1530-1834/13 $26.00 © 2013 IEEE
DOI 10.1109/SIBGRAPI.2013.26



(a) Supersegment generation (Sec. IV) (b) VDI (Sec. III-B) (c) Frustums (Sec. V-A) (d) Rendering (Secs. V-B,V-C)

Fig. 2: (a),(b) VDI generation and (c),(d) rendering. (b) Supersegments are tuples that are organized in a 2D array of lists. (c) Frustums constitute the 3D representations of supersegments. (d) These are eventually used to generate new views.

- Volumetric Depth Image representation extending Layered Depth Images to continuous instead of a discrete depth.
- A technique to generate Volumetric Depth Images that is easy to integrate with existing raycasting codes and introduces only minor overhead.
- An efficient rendering of VDIs allowing for interactive viewpoint changes with fast and efficient depth sorting and opacity correction.

The paper is structured as follows. Sec. II discusses related work, while Sec. III explains basic fundamentals. Secs. IV and V elaborate on the generation and the rendering of Volumetric Depth Images, respectively. Sec. VI discusses the implementation and Sec. VII the results of our approach. Sec. VIII concludes the paper.

II. Related Work

Direct volume rendering techniques can roughly be classified as image-based, most notably ray-casting, and object-based, such as cell projection, shear-warp, or splatting. Nowadays, GPU-based raycasting [3] is the state-of-the art technique for interactive volume rendering [4] and we use it in our approach. Our supersegment generation exhibits some similarity to adaptive object-space importance sampling techniques in that entropy needs to be taken into account. However, in contrast to our approach, these techniques typically rely on distinguished data structures (e.g., LOD volumes [5]). Ament et al. [6] introduce a unified model for generalized isosurfaces allowing for scale-invariant opacity. Viola et al. [7] present an importance-driven automatic focus and context display technique. As an alternative to raycasting, texture-based rendering via 3D textures slices the texture block in back-to-front order with planes oriented parallel to the view plane [8]. Similarly, our implementation for rendering VDIs utilizes the rasteriziation pipeline and blending, but we use frustums instead of planes and do not require the original data during rendering. In contrast, splatting [9] accumulates data points by projecting flat disc-like kernels for each voxel to the image plane. Many adjustments have been proposed to improve its quality and speed (e.g., [10][11]). The projected tetrahedra approach renders partially transparent polygons to render volume data [12][13], based on the projected profile of tetrahedral cells.

Image-based rendering infers new images from existing ones, e.g., with changed lighting or camera configuration [14]. A number of techniques has been proposed to construct different representations from multiple views, like view-dependent texture maps [15], warping [16][17], light fields [18], or Lumigraphs [19]. Meyer et al. [20] use opacity light fields for image-based volume rendering based on multiple renderings by means of proxy surfaces. Rezk-Salama et al. [21] employ depth layers to generate high quality light field representations from volumetric data. Further techniques using multiple images to synthesize new surface-based views of volume data include the works of Choi et al. [22] and Chen et al. [23]. Such techniques allow the adaptation of color and lighting parameters [24], or transfer functions [25]. In contrast, our technique only uses a single view, thus minimizing the generation overhead, and employs a volumetric instead of a surface-based representation.

Shade et al. [1] introduced LDIs representing one camera view with multiple pixels along each line of sight. Multi-layered representations have since been popularized in commercial rendering software to simulate complex materials like skin on synthetic objects [26]. In volume rendering, layer-based representations have been used to defer operations such as lighting and classification [27][28]. This has also been proven effective to cache results [29][30] or certain volumetric properties along the view rays [31], which can be later reused for efficient transfer function exploration. Tikhonova et al. [32] convert a small number of volume renderings to a multi-layered image representation, enabling interactive exploration in transfer function space. In another work, Tikhonova et al. [33] use an intermediate volume data representation based on Ray Attenuation Functions, which encode the distribution of samples along each ray. Shareef et al. [34] use image-based modeling to allow for efficient GPU-based rendering of unstructured grids based on parallel sampling rays and 2D texture slicing. In contrast to VDIs, LDIs and related techniques are targeted toward surfaces (with specific depth values and “vacuum” in

between), and not volumetric representations.

III. Fundamentals

In this section, we derive VDIs starting from the volume rendering integral (Sec. III-A), and subsequently discuss the representation in detail (Sec. III-B).

A. Divisibility

The volume rendering integral [35] defines the color resulting from compositing volume data to be

$$I(D) = \int_0^D c(s) e^{-\int_s^D \tau(t)\,\mathrm{d}t}\,\mathrm{d}s,$$

with $c(s) := C(s)\tau(s)$ representing the premultiplied color where $C(s)$ depicts the radiance or color and $\tau(s)$ stands for the attenuation of the sample s along a ray, with s ranging from 0 (back) to D (front). Discretization with Riemann sums by dividing $[0, D]$ into N segments yields

$$I(D) \approx J(D) := \sum_{i=1}^{N} c(i) \prod_{j=i+1}^{N} T(j), \tag{1}$$

where $c(i) := C(i)\omega(\alpha(i), \delta(i))$ represents the discretized premultiplied color of segment i, with $\omega(\alpha(i), \delta(i))$ representing the corrected opacity $\alpha(i)$, accounting for the potentially varying segment (or step) size $\delta(i)$, defined as follows [36]:

$$\omega(\alpha(i), \delta(i)) := 1 - \big(1 - \alpha(i)\big)^{\delta(i)}, \tag{2}$$

with $T(j) := 1 - \omega(\alpha(j), \delta(j))$ standing for the corrected transmittance of segment j along the view direction, and $\prod_{j=i+1}^{N} T(j)$ is the attenuation due to all segments in front of segment i. Eq. 1 can be grouped into P so-called supersegments, with $s_b(p)$ and $s_f(p)$ representing the index of the back and front segment of supersegment p, respectively (see Fig. 2(a)):

$$J(D) = \sum_{p=1}^{P} \left(\left(\sum_{i=s_b(p)}^{s_f(p)} c(i) \prod_{j=i+1}^{s_f(p)} T(j) \right) \prod_{k=s_f(p)+1}^{N} T(k) \right). \tag{3}$$

Supersegment 1 is located at the back while supersegment P is the frontmost one. Each supersegment features composited premultiplied color c_b^f and transmittance T_b^f:

$$c_b^f := \sum_{i=b}^{f} c(i) \prod_{j=i+1}^{f} T(j),$$

$$T_b^f := \prod_{j=b}^{f} T(j).$$

Eq. 3 can accordingly be rewritten as follows:

$$J(D) = \sum_{p=1}^{P} c_{s_b(p)}^{s_f(p)} \prod_{q=p+1}^{P} T_{s_b(q)}^{s_f(q)}. \tag{4}$$

Eq. 4 provides the basis for VDIs as it states that samplings can be arbitrarily grouped and composited into continuous segments of varying length without any loss in quality (cf. Eq. 1). Varying step size is accounted for by $\omega(\cdot, \cdot)$, which will be revisited for reusing supersegments for preview rendering with different viewing parameters (Sec. V).

B. Volumetric Depth Images

Our Volumetric Depth Image representation saves a 2D array of so-called supersegment lists (one per pixel, Fig. 2b). Each supersegment stores color, opacity, and a pair of depth values. A VDI further contains the modelview and projection matrices used during their generation. Theoretically, if the volume would be densely covered along a ray, only one depth value per supersegment would be sufficient, as the depth of the successor along the ray could be used to terminate a supersegment. However, similar to empty space skipping in raycasting [37], we aim not to generate supersegments for regions with negligible impact. This saves memory and render time, particularly because for many transfer functions and datasets, transitions between dense and transparent regions are common. A VDI can be seen as a generalization of an LDI, as LDIs represent the special case where the bounding pair of depth values are the same: $s_f = s_b$. Trivially, it can also represent a standard image, e.g., from volume rendering, by using one supersegment per pixel, ignoring the depth values.

IV. Volumetric Depth Images from Raycasting

In this section, the generation of VDIs using a modified version of a regular front-to-back raycaster (IV-A) is described. Subsequently, the criterion for merging segments into supersegments is discussed (IV-B).

A. Supersegment Generation with Raycasting

Algorithm 1 Generation of supersegments using raycasting.

```
1:  function SupersegmentGeneration
2:      c ← (0,0,0), T ← 1, p ← −1
3:      for i = 1 → N + 1 do                          ▷ step along ray
4:          g ← Γ(γ, C, α, C(i), α(i))   ▷ new supersegment? (Eq. 5)
5:          e ← (α(i) = 0 ∧ α(i − 1) ≠ 0)
6:          if p ≠ −1 ∧ (g ∨ e ∨ i = N + 1) then ▷ close old supersegment
7:              s_b(p) ← min(i, N)
8:              α_{s_b(p)}^{s_f(p)} ← 1 − T
9:              C_{s_b(p)}^{s_f(p)} ← c/α_{s_b(p)}^{s_f(p)}
10:         if (g ∨ α(i − 1) = 0) ∧ i ≠ N + 1 ∧ α(i) > 0 then
                                              ▷ start new supersegment
11:             p ← p + 1, s_f(p) ← i
12:             c ← (0,0,0), T ← 1
13:         c ← T · α(i)c(i), T ← T · (1 − α(i))
```

A modified front-to-back raycasting procedure for creating supersegments is depicted in Alg. 1. The main difference to standard raycasting is that color and opacity are not composited over the whole ray but only within supersegments. The segmentation criterion Γ (Line 4, see Sec. IV-B (Eq. 5) for details) returns true if the old supersegment should be terminated (Lines 11–12) and a new one should be started (Lines 7–9). Additionally, the current supersegment is finalized

when entering a transparent segment (Line 5), and a new supersegment is started when leaving a transparent segment (Line 10). Finally, a supersegment is closed before terminating the ray (the last iteration $i = N + 1$ solely has this purpose).

When a new supersegment is started, the color and opacity values are reset (Line 12). As c represents a premultiplied color value after compositing, it is divided by the opacity of the supersegment to make it non-premultiplied (Line 9) as a preparation for the subsequent over-operator type blending in the rendering stage (Sec. V).

B. Supersegment Merging Criterion

Segments are created and composited in a front-to-back procedure into supersegments during raycasting. Mostly homogeneous supersegments are desirable to achieve good results, particularly for large view parameter changes in VDI rendering. There is a trade-off between quality and both storage requirements and rendering speed: the more supersegments, the higher the quality but also the higher the memory usage. Our criterion Γ is based on premultiplied color values and the correction of opacity with respect to integration lengths:

$$\Gamma : \gamma > |(c_{s_f}^{i-1}, \alpha_{s_f}^{i-1}) - (\hat{\alpha}(s_f, i)C(i), \hat{\alpha}(s_f, i))| \quad (5)$$

with γ being the sensitivity parameter, s_f is the starting index of the supersegment, $(c_{s_f}^{i-1}, \alpha_{s_f}^{i-1})$ represent color and opacity of the supersegment respectively, and i depicts the current segment. The opacity is length-corrected

$$\hat{\alpha}(s_f, i) = \omega(\alpha(i), \mu(s_f, i-1)),$$

with μ being the length of the supersegment:

$$\mu(f, b) = \sum_{j=f}^{b} \delta(j). \quad (6)$$

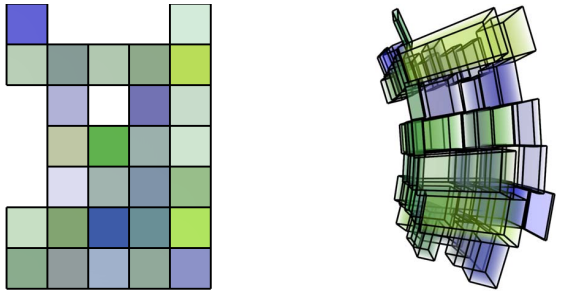
This means that a raycaster segment is merged into the supersegment if the adjusted color and opacity difference is below the user-provided sensitivity parameter γ. Otherwise, a new supersegment is started. This greedy criterion is fast and simple to compute, can easily be integrated with existing raycasting codes, and proved to deliver good results during our experiments. Many other (more complex) schemes are possible and could easily be used with our flexible approach to meet different demands of a specific application at hand.

V. Rendering Volumetric Depth Images

A VDI is rendered by using quadrilateral frustums as a proxy to represent supersegments (Fig. 3, Sec. V-A). These frustums are composited using the over-operator, and thus require depth sorting (Sec. V-B). Finally, during rendering, the correct opacity contribution of every frustum needs to be computed (Sec. V-C).

A. Frustum Geometry

Each supersegment is represented by a frustum for rendering. For every supersegment list of a VDI, four rays are casted from the initial camera position through the corners of the respective pixel. For this purpose, the original modelview and projection matrices are used. The "bottom left" ray defines a perpendicular front and a back plane by using its direction vector as well as the two depth values marking the beginning and end of the respective supersegment. The front and back planes are then intersected by the other three rays, thus defining a quadrilateral frustum belonging to each supersegment.



(a) VDI generation view (b) Rotated and zoomed

Fig. 3: The engine data set (see Sec. VII for details) with very low resolution settings shows the geometry of frustums (Sec. V-A), and the results of both depth sorting (Sec. V-B) and opacity determination (Sec. V-C).

B. Depth Sorting

For rendering our frustums, we employ alpha compositing with the over operator-according to Eq. 7: a over b (i.e., a is "in front of" b) computes the resulting color C as follows [35]:

$$C = C_a\alpha_a + C_b(1 - \alpha_b) \quad (7)$$

where C_a and C_b are the colors, and α_a and α_b are the opacities belonging to a and b, respectively. This requires depth ordering for correct results. We use a back-to-front ordering using Painter's algorithm [38] due to its simplicity and suitability to our problem. All polygons in a scene are ordered by their depth and painted in this order, farthest to closest. The algorithm can fail in some cases, including cyclic overlap or piercing polygons. This, however, cannot happen in our case, as there are only convex frustums that cannot exhibit cyclic overlaps. Numerous other ordering algorithms would be applicable. For instance, Shade et al. [1] use McMillans [17] ordering algorithm in their original LDI paper.

However, due to the way our frustums are constructed, we can do sorting in $O(L\log(L))$ in contrast to $O(S\log(S))$ that would be required when sorting every polygon or frustum on its own (L being the number of supersegment lists while S is the number of supersegments). For this, we exploit that frustums belonging to the same supersegment list are already implicitly sorted. This significantly decreases the sorting cost and makes it invariant with respect to γ.

Each frustum list is represented according to the view ray that was used originally to create the respective supersegment list. They are then reversely ordered with respect to their Euclidean distance to the new camera position (Fig. 2d). Note that the ordering within a frustum list needs to be inverted if a

Algorithm 2 Opacity correction of supersegments in a fragment shader, with o_w denoting the camera position in world space and P_w being the list of the six planes defining the frustum, respectively. FC stands for fragment coordinates, VP for viewport, and DR for depth range.

1: **function** OPACITYCONTRIBUTION
2: $f_c \leftarrow (((2 \cdot \text{FC.xy}) - (2 \cdot \text{VP.xy}))/(\text{VP.zw}) - 1, (2 \cdot \text{FC.z} - \text{DR.near} - \text{DR.far})/(\text{DR.far} - \text{DR.near}), 1)/\text{FC.w}$
3: $f_e \leftarrow \texttt{gl\_ProjectionMatrixInverse} \cdot f_c$
4: $f_w \leftarrow \texttt{gl\_ModelViewMatrixInverse} \cdot f_e$ ▷ Convert fragment position world space f_w
5: $d_w \leftarrow (f_w - o_w)/|(f_w - o_w)|$ ▷ Construct view ray
6: $l \leftarrow \infty$
7: **for all** $p_w \in P_w$ **do** ▷ Determine length l in frustum
8: $v \leftarrow p_w.xyz \cdot d_w$
9: **if** $v > 0$ **then** ▷ Only consider back planes
10: $l \leftarrow \min(l, -(p_w.xyz \cdot o_w) + p_w.w)/v)$
11: $l \leftarrow l - |(f_w - o_w)|$ ▷ view ray length in frustum
12: **return** $\kappa(s_b, s_f, l)$ ▷ Compute opacity (Eq. 8)

supersegment list is viewed in opposite direction with respect to its generating view ray.

C. Opacity Determination

The length l of a new view ray in a frustum (Fig. 2d) is used to adjust the opacity contribution of the respective part of a supersegment. In the same way as for segments (to Eq. 4, Eq. 2), the opacity contribution of supersegments can be adjusted with respect to the step size δ. This can be computed for an arbitrary length l in relation to the original length $\mu(s_b, s_f)$ of the supersegment (Eq. 6) as follows:

$$\kappa(s_f, s_b, l) = \omega(\alpha_{s_f}^{s_b}, l/\mu(s_f, s_b)), \tag{8}$$

where l is determined by intersecting the view ray belonging to the current pixel with the frustum belonging to the supersegment. This is discussed in more detail in the next section by means of Alg. 2.

VI. IMPLEMENTATION

Our prototypical system uses a CUDA-based front-to-back volume raycaster for VDI generation employing one GPU thread per ray. We reserve a fixed-size supersegment list per pixel. If this size limit is reached, the last supersegment has to contain all remaining segments until ray termination (see Alg. 1), potentially leading to a "smearing" artifact. Storing 32 supersegments per pixel proved to be a good compromise between memory usage and rendering quality during the course of our experiments, and this problem was only encountered rarely for very low values of γ.

A VDI is rendered using rasterization as directly supported by commodity graphics hardware using OpenGL and GLSL. Concerning the implementation, there is a significant trade-off between graphics memory usage and the amount of work that needs to be done on-the-fly by the GPU. For the most memory-saving approach, supersegments (depth bounds and color/opacity) could be rendered directly using a geometry shader or a raycasting approach computing the intersections with an (implicit) frustum in a GPU shader. While this is efficient memory-wise, it generates a lot of work per frame that, technically speaking, only needs to be computed once directly after VDI-generation. On the opposite, the whole frustum geometry could be generated and uploaded to the GPU enriched with precomputed information to minimize the runtime cost. Amongst others, this includes uploading the six planes defining a frustum for computing the length that a view ray spends inside the frustum, for opacity correction in the shader. Doing that in a straightforward manner with indexed vertex buffer objects (VBOs) would lead to a massive memory footprint, as for every frustum there are eight vertices, each of which needs to store color and plane information (amongst others) redundantly. This would be fast as it minimizes the work that needs to be done on-the-fly, but occupies a large amount of GPU memory. In our prototype implementation, we strive for a trade-off and generate the frustum geometry (Sec. V-A) used for rasterization once on the CPU and upload it to GPU using VBOs, but compute the planes required for opacity correction (Sec. V-C) on-the-fly in the vertex shader. Backface culling ensures that only one face of the frustum geometry is hit such that the frustum only contributes once.

A fragment shader determines the opacity contribution of a frustum for a specific fragment or view ray (Alg. 2). First, the fragment coordinates are converted to world space (Lines 2–4) to determine the first point of intersection of the view ray with the frustum. Then the view ray d_w is reconstructed (Line 5) to compute the ray length inside the frustum (Lines 6–11) and finally calculate the corrected opacity (Line 12). The depth ordering of series of frustums is done on the CPU using a fast multi-core sorting algorithm. Our implementation only requires OpenGL 2 (ES) capability and thus could efficiently be implemented on almost all modern mobile devices.

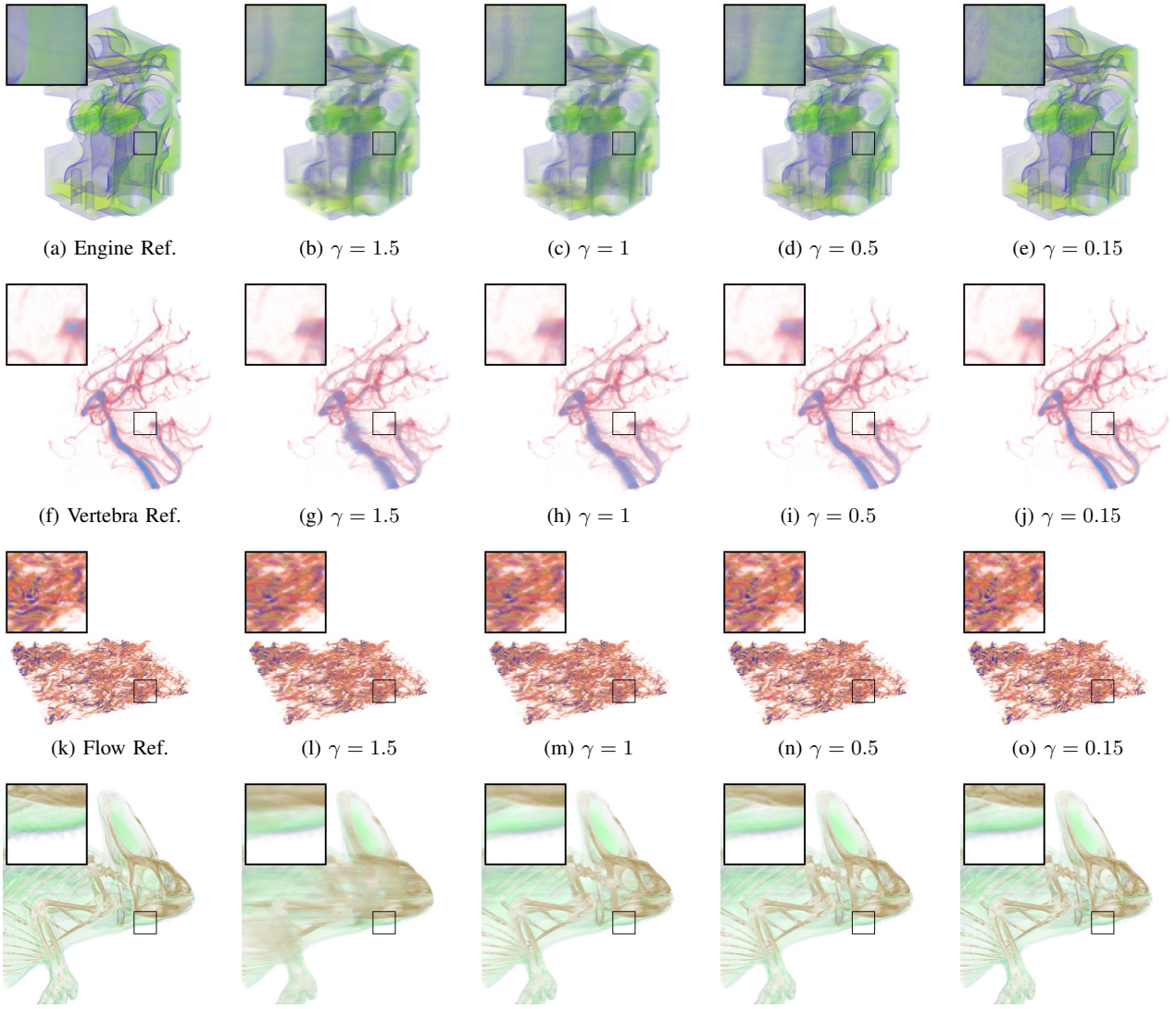
Data set	Resolution	Size	Raycast.	Gen.
Engine	$256 \times 256 \times 256$	32 MB	23ms	11ms
Chameleon	$1024 \times 1024 \times 1080$	2160 MB	560ms	54ms
Vertebra	$512 \times 512 \times 512$	256 MB	48ms	18ms
Flow	$2018 \times 220 \times 1085$	919 MB	37ms	5ms

TABLE I: Data sets and respective raycasting time for results in Fig. 4 for $\gamma = 1$. All data sets are given in 16-bit accuracy.

VII. RESULTS

Our measurements were carried out on an Intel Core i7-2600k and a NVIDIA GTX 580 with 3GB of video memory with a default image resolution of 512×512 unless otherwise noted. The list of data sets we used in our experiments, including their standard raycasting timings, is given in Table I. It also shows the timing overhead for generating a VDI during raycasting, which is relatively low, although varying depending on various factors, most notably data set and transfer function complexity.

Fig. 4 shows VDIs rotated by $50°$ with respect to their direction of generation, for different settings of γ. In general, as



(a) Engine Ref. (b) $\gamma = 1.5$ (c) $\gamma = 1$ (d) $\gamma = 0.5$ (e) $\gamma = 0.15$

(f) Vertebra Ref. (g) $\gamma = 1.5$ (h) $\gamma = 1$ (i) $\gamma = 0.5$ (j) $\gamma = 0.15$

(k) Flow Ref. (l) $\gamma = 1.5$ (m) $\gamma = 1$ (n) $\gamma = 0.5$ (o) $\gamma = 0.15$

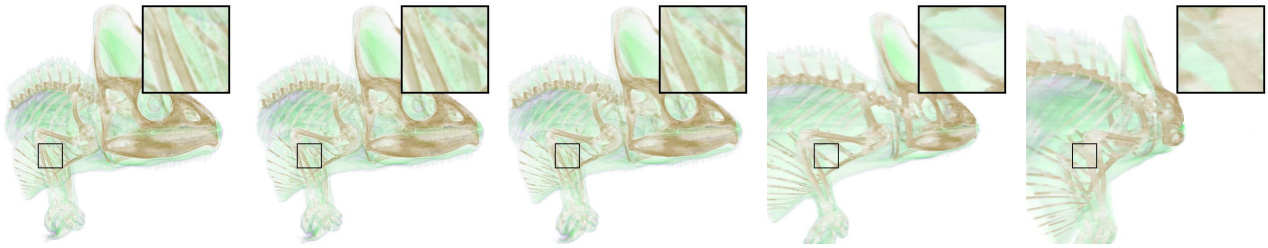
(p) Chameleon Ref. (q) $\gamma = 1.5$ (r) $\gamma = 1$ (s) $\gamma = 0.5$ (t) $\gamma = 0.15$

Fig. 4: Comparison of ground-truth raycasting (leftmost) and the VDI renderings with decreasing γ from left to right and $50°$ rotation with respect to the original camera configuration. Closeups are given for the black rectangle. Fig. 5 provides respective render times and further details.

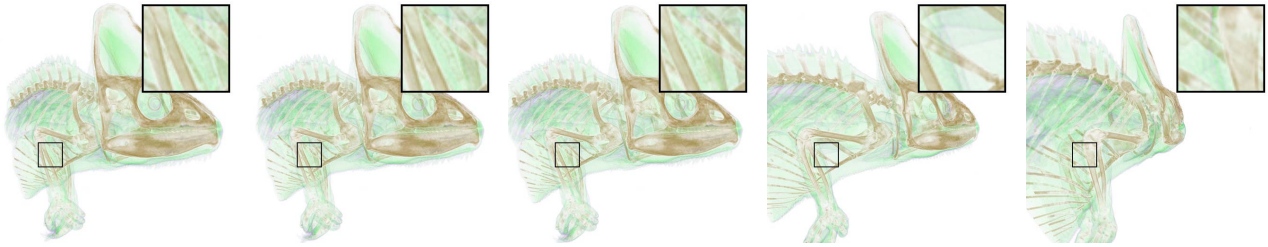
expected, the rendering gets crisper and closer to the raycasting reference the lower the value for γ is. This is due to the fact that the supersegments exhibit a lower deviation in color and opacity from the original underlying data, which leads to less blur during rendering. The loss of detail due to too large segments can clearly be seen by means of the almost opaque bone structure of the chameleon data set (Fig. 4q). A similar effect can be observed by the example of the vertebra in Fig. 4g. Figs. 4j, 4o, and 4t show that particularly for low values of γ the VDIs provide a good approximation of both the structure and value with respect to the reference volume raycasting.

Fig. 5 depicts the corresponding timings and shows that the influence of γ on the number of supersegments varies in detail with each data set due to their differing complexity, but all functions share an exponential decay behavior. A deeper investigation of this behavior remains for future work. The render time increases with an increasing number of supersegments due to lower values for γ, yet at a much lower rate. We attribute this to the circumstance that, rather than the number of primitives, the number of generated fragments is crucial for the performance, due to the comparably compute-intense fragment shader for α-correction (Alg. 2). However, all render times are below 50ms and thus highly interactive.

Table II shows that the number of supersegments increases approximately linearly with the number of supersegment lists across different resolutions, and so does the time for



(a) VDI, 0°, 42ms (b) VDI, 5°, 42ms (c) VDI, 10°, 40ms (d) VDI, 45°, 39ms (e) VDI, 90°, 39ms



(f) Ref., 0°, 560ms (g) Ref., 5°, 592ms (h) Ref., 10°, 604ms (i) Ref., 45°, 558ms (j) Ref., 90°, 542ms

Fig. 6: Comparison of ground-truth renderings with the raycaster and VDI-renderings with $\gamma = 0.8$ and 450k supersegments. The geometry was generated at 0°. Timings for VDI both include sorting and rendering.

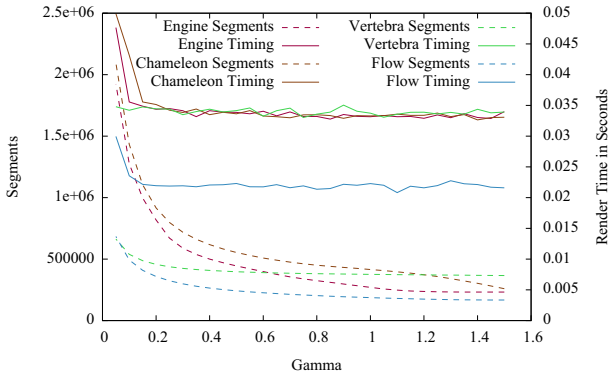


Fig. 5: Number of segments and render timings for different data sets and different settings for γ.

Res.	Lists	SupSegs	Geometry	Sorting	Rendering
Engine					
256^2	29k	67k	36ms	5ms	11ms
512^2	118k	269k	142ms	6ms	33ms
768^2	267k	606k	323ms	21ms	71ms
Chameleon					
256^2	29k	103k	48ms	5ms	10ms
512^2	119k	415k	189ms	6ms	33ms
768^2	268k	934k	428ms	20ms	67ms
Vertebra					
256^2	29k	94k	45ms	5ms	10ms
512^2	118k	375k	175ms	8ms	33ms
768^2	267k	846k	398ms	22ms	68ms
Flow					
256^2	15k	46k	28ms	4ms	11ms
512^2	63k	186k	111ms	6ms	22ms
768^2	113k	419k	250ms	8ms	40ms

TABLE II: Supersegment numbers and timings for $\gamma = 1$. Respective raycasting times are given in Table I.

geometry creation and rendering. Sorting times are consistently below rendering times and only depend on the number of supersegment lists as expected. In our implementation, the geometry generation step only needs to be carried out once as a preprocessing step and is only executed on a single CPU core with large potential for improvement.

Figure 6 shows the influence of the rotation angle away from the original camera view on quality, as determined by means of the images from VDI rendering and respective reference images from raycasting. Except for unperceivable numerical deviations, the rendered VDI match the raycasting image initially (Figs. 6a and 6f). For small rotation angles, the visual difference is minor, only becoming noticeably larger for large angles (e.g., Figs. 6d and 6e). Timings for VDI rendering are about one order of magnitude faster in comparison to raycasting, leading to a significant gain from less than 2 fps to over 20 fps. This goes along with a much lower graphics memory usage, enabling the rendering on more limited hardware. The reduction in memory usage heavily depends on the implementation as discussed in Sec. VI. With our prototype application, it results in approximately 82MB: 450k supersegments times eight vertices per frustum times 32 bit per channel RGBA and 64 bit for the supersegment bounds. This is more than one order of magnitude lower than that of the original data set (2160MB), and there is still substantial room for improvement if required by a specific use case. For instance, by using 1 Byte instead of 4 Bytes per color channel, by generating the geometry on-the-fly on the GPU, or using lower 2 Byte precision to store the supersegment bounds, the memory usage could be reduced to $450\text{k}\cdot(4+4) \approx 3\text{MB}$. Evaluating these possibilities in more detail remains, however, for future work.

VIII. Conclusion

We introduced Volumetric Depth Images as a view-dependent volume representation that allows for arbitrary camera view changes and can both be generated and rendered at interactive rates. It only requires a single view and can easily and quickly be generated using slightly modified raycasters. Instead of only saving depth and color values for surfaces as in the LDI approach, supersegments covering a certain depth range with composited color and opacity values are determined from an initial view point and stored for later interaction. We showed that even for large changes in the camera view good quality images are generated from a VDI. There are numerous possible applications, ranging from preview rendering in a local or remote rendering setting, to offline scenarios with VDIs being generated by a supercomputer and visualized at a later stage on a workstation. Besides investigating these scenarios more closely, we also plan to look into further degrees of freedom for interactive exploration as proposed in other works, e.g., adjustment of the transfer function. Attempting to combine several VDIs for rendering, or merging similar supersegments across supersegment lists (pixels) to exploit local color coherence, also remains for future work. We would further like to investigate more elaborate supersegment partitioning criteria. Finally, we also plan to experimentally compare our technique to other image-based volume rendering techniques.

References

[1] J. Shade, S. Gortler, L.-w. He, and R. Szeliski, "Layered depth images," in *25th annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '98, 1998, pp. 231–242.

[2] Z. Zheng, N. Ahmed, and K. Mueller, "iview: A feature clustering framework for suggesting informative views in volume visualization," *IEEE Trans. on Visualization and Computer Graphics*, vol. 17, no. 12, pp. 1959–1968, 2011.

[3] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl, "A simple and flexible volume rendering framework for graphics-hardware-based raycasting," in *Volume Graphics*, 2005, pp. 187–195.

[4] C. Rezk-Salama, M. Hadwiger, T. Ropinski, and P. Ljung, "Advanced illumination techniques for gpu volume raycasting," in *ACM SIGGRAPH Courses Program*, 2009.

[5] P. Ljung, C. Winskog, A. Persson, C. Lundstrom, and A. Ynnerman, "Full body virtual autopsies using a state-of-the-art volume rendering pipeline," *IEEE Trans. on Visualization and Computer Graphics*, vol. 12, no. 5, pp. 869–876, 2006.

[6] M. Ament, D. Weiskopf, and H. Carr, "Direct interval volume visualization," *IEEE Trans. on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 1505–1514, 2010.

[7] I. Viola, A. Kanitsar, and M. E. Gröller, "Importance-driven volume rendering," in *IEEE Visualization*, 2004, pp. 139–146.

[8] J. Krüger and R. Westermann, "Acceleration techniques for gpu-based volume rendering," in *IEEE Visualization*, 2003, pp. 287–292.

[9] L. A. Westover, "Splatting: a parallel, feed-forward volume rendering algorithm," Ph.D. dissertation, 1991, uMI Order No. GAX92-08005.

[10] K. Mueller, N. Shareef, J. Huang, and R. Crawfis, "High-quality splatting on rectilinear grids with efficient culling of occluded voxels," *IEEE Trans. on Visualization and Computer Graphics*, vol. 5, no. 2, pp. 116–134, 1999.

[11] F. Vega-Higuera, P. Hastreiter, R. Fahlbusch, and G. Greiner, "High performance volume splatting for visualization of neurovascular data," in *IEEE Visualization*, 2005, pp. 271–278.

[12] P. Shirley and A. Tuchman, "A polygonal approximation to direct scalar volume rendering," in *Computer Graphics*, 1990, pp. 63–70.

[13] S. Röttger, M. Kraus, and T. Ertl, "Hardware-accelerated volume and isosurface rendering based on cell-projection," in *IEEE Visualization*, 2000, pp. 109–116.

[14] H.-Y. Shum and S. B. Kang, "A survey of image-based rendering techniques," in *Videometrics, SPIE*, 1999, pp. 2–16.

[15] P. E. Debevec, C. J. Taylor, and J. Malik, "Modeling and rendering architecture from photographs: a hybrid geometry- and image-based approach," in *23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96, 1996, pp. 11–20.

[16] W. R. Mark, L. McMillan, and G. Bishop, "Post-rendering 3d warping," in *Symposium on Interactive 3D graphics*, 1997, pp. 7–16.

[17] L. McMillan and G. Bishop, "Plenoptic modeling: an image-based rendering system," in *22nd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '95, 1995, pp. 39–46.

[18] M. Levoy and P. Hanrahan, "Light field rendering," in *23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96, 1996, pp. 31–42.

[19] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen, "The lumigraph," in *23rd annual conference on Computer graphics and interactive techniques*, ser. SIGGRAPH '96, 1996, pp. 43–54.

[20] M. Meyer, H. Pfister, C. Hansen, C. Johnson, M. Meyer, H. Pfister, C. Hansen, and C. Johnson, "Image-based volume rendering with opacity light fields," no. UUSCI-2005-002. Tech Report, 2005.

[21] C. Rezk-Salama, S. Todt, and A. Kolb, "Raycasting of light field galleries from volumetric data," in *Eurographics conference on Visualization*, ser. EuroVis'08. Eurographics Association, 2008, pp. 839–846.

[22] J.-J. Choi and Y.-G. Shin, "Efficient image-based rendering of volume data," in *Computer Graphics and Applications, 1998. Pacific Graphics '98. Sixth Pacific Conference on*, 1998, pp. 70–78, 226.

[23] B. Chen, A. Kaufman, and Q. Tang, "Image-based rendering of surfaces from volume data," in *Eurographics conference on Volume Graphics*, ser. VG'01. Eurographics Association, 2001, pp. 281–300.

[24] T. He, L. Hong, A. Kaufman, and H. Pfister, "Generation of transfer functions with stochastic search techniques," in *IEEE Visualization*, 1996, pp. 227–234.

[25] Y. Wu and H. Qu, "Interactive transfer function design based on editing direct volume rendered images," *IEEE Trans. on Visualization and Computer Graphics*, vol. 13, no. 5, pp. 1027–1040, 2007.

[26] C. Donner and H. W. Jensen, "Light diffusion in multi-layered translucent materials," *ACM Trans. Graph.*, vol. 24, no. 3, pp. 1032–1039, 2005.

[27] T. Ropinski, J. Prassni, F. Steinicke, and K. Hinrichs, "Stroke-based transfer function design," in *Fifth Eurographic, IEEE VGTC conference on Point-Based Graphics*, ser. SPBG'08. Eurographics Association, 2008, pp. 41–48.

[28] P. Rautek, S. Bruckner, and E. Gröller, "Semantic layers for illustrative volume rendering," *IEEE Trans. on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1336–1343, 2007.

[29] E. J. Luke and C. D. Hansen, "Semotus visum: a flexible remote visualization framework," in *IEEE Visualization*, ser. VIS '02, 2002, pp. 61–68.

[30] E. LaMar and V. Pascucci, "A multi-layered image cache for scientific visualization," in *2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, ser. PVG '03, 2003, pp. 61–68.

[31] K.-L. Ma, M. F. Cohen, and J. S. Painter, "Volume seeds: A volume exploration technique," *The Journal of Visualization and Computer Animation*, vol. 2, no. 4, pp. 135–140, 1991.

[32] A. Tikhonova, C. Correa, and K.-L. Ma, "Explorable images for visualizing volume data," in *IEEE Pacific Visualization Symposium*, 2010, pp. 177–184.

[33] A. Tikhonova, C. D. Correa, and K.-L. Ma, "An exploratory technique for coherent visualization of time-varying volume data," in *Eurographics conference on Visualization*. Eurographics Association, 2010, pp. 783–792.

[34] N. Shareef, T.-Y. Lee, H.-W. Shen, and K. Mueller, "An image-based modelling approach to gpu-based rendering of unstructured grids," in *Volume Graphics 2006*, 2006, pp. 31–38.

[35] N. Max, "Optical models for direct volume rendering," *IEEE Trans. on Visualization and Computer Graphics*, vol. 1, no. 2, pp. 99–108, 1995.

[36] K. Engel, M. Hadwiger, J. M. Kniss, A. E. Lefohn, C. R. Salama, and D. Weiskopf, "Real-time volume graphics," in *ACM SIGGRAPH 2004 Course Notes*, 2004.

[37] D. Cohen and Z. Sheffer, "Proximity clouds - an acceleration technique for 3d grid traversal," *The Visual Computer*, vol. 11, pp. 27–38, 1994.

[38] J. Ahrens and J. Painter, "Efficient sort-last rendering using compression-based image compositing," in *Eurographics Workshop on Parallel Graphics and Visualization*, 1998, pp. 145–151.