Prediction-Based Load Balancing and Resolution Tuning for Interactive Volume Raycasting

Valentin Bruder, Steffen Frey, and Thomas Ertl

University of Stuttgart

Abstract

We present an integrated approach for real-time performance prediction of volume raycasting that we employ for load balancing and sampling resolution tuning. In volume rendering, the usage of acceleration techniques such as empty space skipping and early ray termination, among others, can cause significant variations in rendering performance when users adjust the camera configuration or transfer function. These variations in rendering times may result in unpleasant effects such as jerky motions or abruptly reduced responsiveness during interactive exploration. To avoid those effects, we propose an integrated approach to adapt rendering parameters according to performance needs. We assess performance-relevant data on-the-fly, for which we propose a novel technique to estimate the impact of early ray termination. On the basis of this data, we introduce a hybrid model, to achieve accurate predictions with minimal computational footprint. Our hybrid model incorporates aspects from analytical performance modeling and machine learning, with the goal to combine their respective strengths. We show the applicability of our prediction model for two different use cases: (1) to dynamically steer the sampling density in object and/or image space and (2) to dynamically distribute the workload among several different parallel computing devices. Our approach allows the renderer to reliably meet performance requirements such as a user-defined frame rate, even in the case of sudden large changes to the transfer function or camera orientation.

Keywords:

Volume raycasting, performance prediction, load balancing

1. Introduction

Volume visualization is a widely used tool for visualization of measured and simulated data in numerous different areas such as physics, engineering, biology and many more. By enabling users of visualization applications to dynamically interact with the volume data, additional insight beyond the initial focus may be gained. Thereby, classic user interactions are adjustments to the transfer function (which maps density values to color) as well as changes to the camera configuration (e.g., rotation and zooming). There are typically two main factors that contribute to a satisfying user experience during interactive exploration of volume data sets: low response times and a high rendering quality. While the latter can be achieved by employing a high sampling of the data set, low latencies and high frame rates are crucial for response times. In the context of the recent revive of virtual reality for scientific applications [1], maintaining high and stable frame rates as well as low latencies gains even more importance. In those applications, variable frame rates often tend to cause unpleasant side effects, such as cybersickness, for many users.

To be able to gain interactive frame rates for volume visualizations on workstations, GPUs are often used to accelerate the computation and rendering. Besides the hardware used for computation, interactively changed parameters (i.e., transfer function and camera configuration) have a significant impact on rendering performance. In order to accomplish constant interactivity, those variations in performance need to be accounted for, especially in challenging cases with significant changes between frames (e.g., switching to a different transfer function). One way of absorbing such effects is to adapt the sampling density in object or image space. However, in the case of an interactive application, the basis for this adaption has to be some kind of assessment of how the performance will evolve in upcoming frames (after potentially big changes) in order to avoid unpleasantly long response times or jerky motions.

Predicting performance of volume rendering on parallel hardware is a challenging task because of the involved complexity. Numerous factors have a significant, non-obvious impact on performance. For instance, this includes the hardware employed for parallel computation, as well as the specific algorithm and parameter configuration that

may be changed during runtime.

We propose our method to dynamically predict performance of a volume raycasting application that uses popular acceleration techniques. To show the usability of our technique, we present two use cases that are based on our frame time prediction. In the first one, we use the predictions to dynamically adjust the sampling rate of the volume rendering process to reliably meet a user-defined frame target (i.e., interactive frame rates). Thereby, we can adjust the sampling rate in ray space (integration step size along rays) as well as in image space (image resolution, i.e. the number of rays). As a second use case, dynamically distribute the computational load among multiple heterogeneous GPUs and balance this load according to our predicted frame execution times.

In the following section, we give an overview on related work (Sec. 2), afterwards we discuss what we consider to be the main contributions of our work.

- We present our general approach for performance prediction and the tuning of sampling rate in image and ray space (Sec. 3). It is based on the following components:
- assessing performance-critical numbers of raycasting acceleration techniques, including the impact of early ray termination (ERT) and empty space skipping (Sec. 4);
- on the fly prediction of the execution time of upcoming frames using a hybrid performance model, (Sec. 5);
- and balancing of the computational load among multiple devices in real-time as well as steering rendering quality towards a user-defined frame rate (Sec. 6).

To the best of our knowledge, on-line prediction of volume rendering performance has not been published before our conference paper [2]. This work is an extended version of that paper. In detail, the extensions compared to our conference paper are:

- load balancing between different GPUs as an additional use case,
- resolution adjustment in image space, also combined with tuning in ray space,
- and minor improvements and additions, such as local illumination.

We present and discuss results in Sec. 7 and conclude our work in Sec. 8.

2. Related work

Volume visualization and frame rate adaption. Volume visualization has been a core subject in scientific visualization research for several decades. In recent times, raycasting has turned out to be one of the mostly used techniques, with its parallel nature supporting GPU and distributed implementations [3]. Salama et al. [4] give an overview on basic volume rendering techniques, thereby focusing on illumination and acceleration techniques that we use as well.

Many works have focused on distributed volume rendering, due to the computational requirements posed by high resolution data sets. The current state of the art in GPU techniques for interactive large-scale volume visualization is discussed by Beyer et al. [5]. Especially for distributed rendering, load balancing plays an important role [6, 7]. In this context, Fogal et al. [8] discuss and investigate different algorithms for load balancing in their work, while Müller et al. [9] demonstrate that zooming on parts of volume data sets critically impairs load balance during distributed rendering. To counter this effect, they dynamically reorganize the data distribution in their cluster. The decision on when to move data to another node is based on a simple cost function and the actual load of the previous frame. While such cost functions as basis for load balancing typically work well in the case of gradual changes, sudden changes (e.g. due to a rapidly adjusted transfer function) cannot be handled adequately, inducing significant load-imbalance and performance drops.

Rendering systems typically fix either image quality or frame rate during user interaction. There is some work on techniques designed to keep stable frame rates for image-based rendering, which we do as well as one application of our prediction model. Shen and Johnson [10], Qu et al. [11] and others re-use pixel values from previous frames and use that to achieve stable frame rates. Wong and Wang [12] have the same goal for real-time rendering applications but use an open-loop approach of the image generation process underpinned by estimations of its constituents. Using artificial neural networks and fuzzy models, as well as detailed descriptions of distinct rendering processes, they relate inputs and outputs in a non-linear model. In contrast, Woolley et al. [13] take a more simple approach by using metrics, based on image space

distances to steer progressive raytracing. Frey et al. [14] use a progressive approach to steer the volume visualization process, thereby focusing on resource management, response times and sampling errors. Compared to our approach however, none of those techniques adapt the frame-rate-based on an on-the-fly prediction of the execution time.

Performance prediction. There is a large amount of research in the area of application performance prediction and modeling for parallel architectures. However, it is mostly limited to the fields of system architecture and high-performance computing, whereas the research in (interactive) visual computing, which has its own characteristics and challenges, is comparably sparse. Various different approaches have been proposed for performance modeling, including performance skeletons [15], regression [16], genetic algorithms [17], and machine learning [18]. Those approaches primarily target performance prediction in large-scale (HPC) systems. However, visual computing applications have different characteristics than those systems in that they typically rely heavily on interaction. The data that is being used for performance modeling typically stems from either specific hardware characteristics, such as (parallel) computational operations per second and memory bandwidth; or from empirical measurements, such as frame execution times and performance counters. Using the latter combined with an analytical model has been defined as a "semi-empirical" model [19]. In our approach, we employ a machine learning model to learn from execution time measurements and combine this with an analytical model, based on known properties of the volume raycasting algorithm. Therefor, we consider it to be such a semi-empirical model.

There exist various off-line performance modeling tools for GPGPU, which has many similarities to GPU volume rendering. An overview of the landscape is given by Madougou et al. [20]. Amarís et al. [21] compare different machine learning models, namely linear regression, support vector machines and random forests with a BSP-based analytical model for the task of GPU execution time prediction.

In contrast, work on real-time rendering or scientific visualization incorporating real-time performance prediction is comparably sparse. The proposed techniques mainly focus either on performance models for the visualization pipeline [22] or on object-order rendering algorithms [23, 24]. Ganestam and Doggett [25] perform auto tuning for

interactive ray tracing, thereby using an analytical GPU architecture model as a basis. Compared to our work, their approach mainly focuses on ray-tracing and while their model incorporated caching effects to some degree, other hardware-level algorithms, such as swizzling, are not covered. In contrast, our model covers those effects implicitly via the machine learning approach.

Specifically for volume rendering on clusters, Rizzi et al. [26] presented an analytical model for off-line prediction of scaling behavior. They predict timings for each part of the overall procedure and sum them up for the overall prediction. Other works have concentrated exclusively on the compositing of images from different nodes [27, 28] in distributed volume rendering and analyzed performance theoretically. Our method mainly differs from the above mentioned approaches, in that we focus on on-line performance prediction. Furthermore, we target volume raycasting in a workstation environment.

3. Overview

We perform volume raycasting with front-to-back compositing. For acceleration, we employ early ray termination (ERT) as well as object-order empty space skipping as widely used techniques. We also implement gradient-based local illumination, with a globally defined light direction. Those gradients are evaluated on the fly, using central differences during our raycast, to limit the memory requirements. At the core of our technique is a hybrid model that is able to predict the execution time of the upcoming frame and adjust sampling density in object as well as image space, based on this prediction. Fig. 1 gives an overview of our approach.

Central to all processing steps are user interactions (cf. top row in Fig. 1): loading a new volume data set, changing the transfer function, and rotating or zooming the camera. When loading a data set, the volume is divided into coarse blocks (we use a resolution of 16^3 voxels per block throughout this paper). We calculate a density histogram H for each of those volume blocks, representing the distribution of scalar density values in its respective block. The histograms H have to be updated only if the volume data set changes. In a next step, we use the user-selected opacity channel T_α

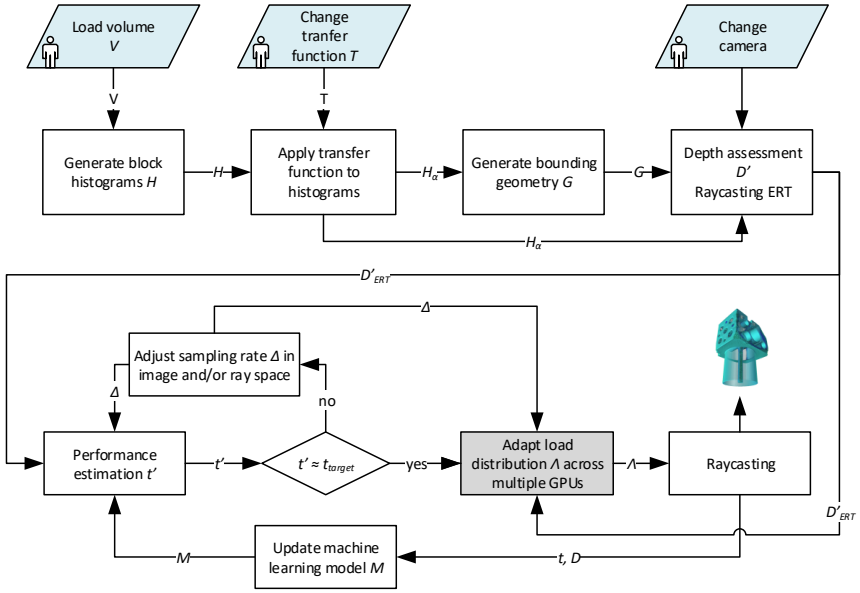


Figure 1: Overview of our adaptive volume rendering process. The top row depicts possible user interactions that trigger data generation and assessment methods (second row). The lower part shows our prediction and parameter tuning approach. Adaption of load distribution (gray) is only used for multi GPU setups.

of the transfer function to derive opacity histograms H_α from the density distribution histograms H. Again, there is one opacity histogram H_α per block, but in this case representing opacity distributions instead of density distributions. This step has to be performed whenever the user changes the transfer function. By directly evaluating the opacity histograms, we determine which blocks of the low-resolution volume are empty, and use this information to generate a bounding geometry G that is used for empty space skipping. Therefor, we use OpenGL to rasterize G and determine the depth of the foremost (D_{front}), as well as the backmost (D_{back}) fragment of the bounding geometry G in a single render pass. Those depth values are used as ray entry and exit points. In order to incorporate estimated effects of ERT in our prediction model, we further adjust the depth values D_{back} to D'_{ERT}. In the context of this paper, we focus on achieving interactive, stable frame rates for a single-node volume rendering application. Therefor, the user may select a target frame rate t_{target} that we aim to achieve and keep at all times during user exploration. As parameters, we can adjust the sampling rate Δ along each

ray and/or the number of rays, to basically trade rendering quality for performance. For this, we follow an iterative optimization approach, by looping over the following operations until we approximately predict the target frame rate:

- We estimate, on basis of D'_{ERT} and our model M, the time t' that would be achieved with the current step size and/or resolution Δ.
- If the prediction t' is close to t_{target}, we stop the adaption.
- Otherwise, we calculate a new step size and/or resolution candidate for Δ.

In the case of having multiple computing devices available for rendering, we support using our prediction for load balancing. Therefor, we adapt the load distribution Λ between available devices, based on the adapted sampling rate respective image resolution Δ, and the depth estimation D'_{ERT}. Next, we actually raycast the volume, by using the obtained value for Δ (ray and/or image space) and the load distribution Λ. Finally, we update our prediction model M by adding the measured values for the execution time t and the actual depth D_{ERT} after ERT, assessed during the actual raycast.

4. Collection of performance-relevant data

Object-order empty space skipping and early ray termination (ERT) are two widely used acceleration techniques for volume raycasting. Therefor, we focus on those in particular, for our performance assessment. In this section, we describe our approach for collecting data that is relevant with respect to those acceleration techniques. We base our assessment, as well as the actual empty space skipping on a coarse volume representation. Therefor, we partition the volume into blocks of 16^3 voxels each and compute a density histogram for each one of them (cf. Sec. 4.1). The histogram data is used to determine the ray entry and exit points, that define depth D without considering ERT (cf. Sec. 4.2). We use those values for the prediction as well as the actual raycasting acceleration. In Section 4.3, we discuss how we incorporate an estimation of ERT effects in our model, that is based on per block opacity histograms H_α.

4.1. Histograms of volume blocks (H and H_α)

When loading a volume data set V, we logically partition the volume into coarse blocks of 16^3 voxels. Using all scalar values contained in a respective block, we generate

density histograms H. Those histograms have a size of 64 bins in our implementation, because we use data sets with 8 and 16 bit precision (more bins could be necessary for volumes with higher precision of scalar values). After applying the transfer function, transparency is a crucial factor for the performance of volume raycasting when using empty space skipping and early ray termination. We compute opacity histograms H_α from every density histogram by applying the opacity transfer function $T_\alpha : \mathbb{R} \rightarrow \mathbb{R}$. Thereby, we distribute the computed values into 16 bins, mainly because it is more efficient during our ERT approximation step, without having much impact on the estimation performance (cf. Sec. 4.3). Each bin b in the original density histogram H represents a density range $[v_{\min}, v_{\max}]$. We generate H_α from H by basically looping over those bins b. Thereby, we integrate over the range $[v_{\min}, v_{\max}]$ with the user-defined (opacity) transfer function $T_\alpha(b)$, resulting in opacity values b_α:

$$\forall b \in H : b_\alpha = \int_{v_{min}}^{v_{max}} T_\alpha(b).$$

Those opacity values b_α are then used to select the respective opacity histogram bin b_α of H_α to which we add the number of corresponding elements from the original bin b of the density histogram H.

We generate one opacity histogram H_α per volume block. Due to the fact that we use the density values as well as the transfer function, this procedure has to be performed whenever the user either loads a new volume data set or changes the transfer function.

4.2. Depth assessment (D_{front} and D_{back})

The amount of empty space depends on the volume characteristics as well as the selected transfer function. We employ our opacity-mapped histogram H_α (cf. Sec. 4.1) to implement object-order empty space skipping. For this, we pre-process a bounding geometry of the volume to determine entry (D_{front}) and exit points (D_{back}) that are more closely to the visible data than an otherwise commonly used bounding cuboid.

To decide whether a block of our proxy geometry is visible or not, we use our opacity-mapped block histograms H_α, by simply evaluating if there are values in bins for non-transparent voxels. We generate quads for the surfaces of the outermost voxels, thereby creating a polygon mesh of the volume hull. We rasterize this geometry using

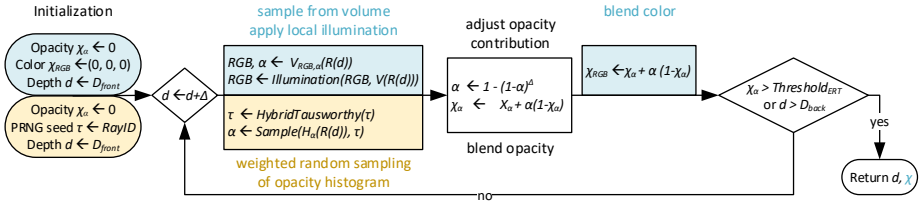


Figure 2: Front-to-back raycasting along a ray using sampling distance Δ. Steps in orange are executed only for ERT estimation (as discussed in Sec. 4.3), the blue ones only for the actual rendering.

a minimum blend equation. This allows us to write the minimum depth as well as the negated maximum depth values into the frame buffer in a single render pass. Using this approach, we cannot skip empty space inside a volume (i.e., our bounding hull) conceptually. This limitation could be circumvented by using a dual depth peeling approach with multiple rendering passes, at the cost of a higher overhead time (this remains for future work). The depth assessment step (i.e., the rasterization) has to be performed whenever the user changes camera parameters, while the generation of the bounding geometry only has to be performed whenever the transfer function or data set changes.

4.3. Early ray termination (D_{ERT} & D'_{ERT})

Early ray termination (ERT) is a simple method that can possibly result in huge performance gains for volume raycasting. The actual speedup mostly depends on the used data set and the transfer function. However, compared to the simplicity of the approach, the a-priori estimation of the actual performance gain is non trivial. This stems from the fact that a possible estimation cannot be solved locally (e.g., on a per-block basis), in contrast to the depth estimation. That means we have to consider the full accumulated opacities along the rays. To nevertheless achieve such an estimation of the ERT impact on a depth segment D in reasonable time, we implement a modified version of our standard raycasting procedure.

Fig. 2 outlines the estimation process as well as our general raycasting algorithm by using two colors for distinction. Here, the orange colored blocks indicate an execution for the ERT estimation run only (Fig. 2, lower half) while the blue colored ones are only processed during the actual volume rendering (Fig. 2, upper half). First, we initialize

opacity and the ray starting position. For the actual raycasting, we also initialize the color value, while for the ERT estimation, we use the thread-id to create a seed for our pseudo-random number generator (PRNG). After the setup phase, we process our raycasting loop, in which we sample at depth d along the respective ray R in front-to-back order by using step size Δ. The sampling starts at D_{front}, the entry point determined by our depth assessment (cf. Sec. 4.2), and we sample until we reach D_{back} or the opacity surpasses an ERT threshold value.

For actual volume rendering, we fetch the respective scalar value from the data set and apply the transfer function, resulting in color and opacity values. For the ERT estimation pre-run, we use the opacity block histograms H_α (that we also use for depth assessment), instead of sampling the volume data. In more detail, we start with generating a pseudo-random number τ, using a hybrid Tausworthe RNG (compare [29]). Next, we determine the block we are currently in with respect to the sampling position on the ray $R(d)$. Using the opacity histogram $H_\alpha(R(d))$ of this block and τ, we randomly draw an opacity value α. Thereby, we weight each histogram bin according to its size, i.e. the sampling is proportional to the number of elements in each bin.

The core idea behind using opacity histograms H_α is to estimate the ERT behavior in a realistic manner at a drastically reduced cost compared to the actual rendering. The cost savings especially stem from a largely reduced I/O cost, that is particularly high due to the typically memory bound nature of volume rendering. We use 16 byte histograms with one byte per bin for each block (a block aggregates 16^3 voxels) in our implementation (cf. Sec. 4.1). This has the advantage that the whole histogram can be obtained using only a single fetch operation on GPUs. This is also comparably fast across multiple rays due to texture caching. By using random sampling of the opacity histogram values, we account for the statistical distribution of the actual opacity values and thereby aim to more closely reproduce the actual raycast. We also sample much more coarsely along the rays, which also contributes significantly in reducing the computational cost compared to the full volume rendering. In both raycasting passes, we adjust the opacity according to the used step size Δ. This has two reasons: we sample the opacity histograms for ERT estimation with a much lower frequency than the actual raycast, that is why we have to adjust opacities, to make them directly correspondent

to one another. Secondly, as step sizes may be dynamically adjusted (see Sec. 6), the correction is crucial for producing similar results when using different step sizes (apart from under-sampling effects).

The raycasting loop terminates if the accumulated opacity χ_α exceeds a defined threshold (early ray termination) or the sampling along the ray exits the proxy geometry. In either case, we use the depth value, as ERT estimation value or training data. In our actual rendering, we naturally present the pixel color value.

5. Hybrid performance model

We use a hybrid performance model to perform an on-line estimation of the execution time of the upcoming frame. Our model may be categorized as a "semi-empirical" performance model (cf. [19]), because we use empirical measurements of previous execution times as well as known attributes of our volume raycasting algorithm. To learn hardware-specific characteristics, such as caching or swizzling algorithms, we employ a machine learning model on the basis of execution time measurements. This part of our model effectively learns and estimates the average cost σ per sample during raycasting (Sec. 5.1). Combining this approximated sample cost with an estimated depth per ray D'_{ERT} (Sec. 4), we predict the total cost t' of rendering the upcoming frame (Sec. 5.2).

5.1. Machine learning: prediction of sample cost σ

We based our decision of the machine learning technique to use mainly on two specific requirements. First, the learning algorithm has to be fast enough to work in realtime, i.e. training as well as evaluation has to be significantly faster than a single frame execution. Second, the technique should be able to perform non-linear regression. We decided to employ kernel recursive least squares (KRLS) as technique, because it fulfills our two requirements, is comparably simple, but nevertheless shows convincing prediction results [30]. The Dlib machine learning library [31] provides an implementation of KLRS that we use in our model. One separate machine learning model is used for each device in the case of employing multiple devices for rendering. Due to the nature of the KRLS algorithm, weights cannot be transferred directly between different runs, i.e. we have to build a new model for every data set.

KRLS is a kernel-based regression algorithm that is able to dynamically include measurement samples for training during runtime, meaning that the model is dynamically trained during runtime and does not need any prior training sequence (see Sec. 7.2 for a discussion of the approximation accuracy and learning speed). Through the use of the recursive least squares (RLS) algorithm, with the addition of Mercer kernels, non-linear regression is implemented. The core of the RLS algorithm is an optimization problem (whose solution is maintained every frame) to find weights w by minimization as follows:

$$\min_{w} \left(\sum_{i} \lambda^{n-i} (y_i - x_i^T \times w)^2 \right) \tag{1}$$

Here, (x_i, y_i) is a pair of training points, where x_i denotes a feature vector and y_i is a target scalar value. The "forgetting factor" λ may be used to give exponentially less weight to older samples.

We use linear radial basis functions as kernel function because of their flexibility. The target scalar value we want to predict is the sample cost σ, while our feature vector consists of several properties that we judge to have a significance for the value of σ:

- **Viewing angles** that we derive directly from the rotation of our arcball camera. Among others, they impact performance because of different texture respective memory access patterns caused by the perspective.
- **Size of a splatted voxel** potentially has a significant impact on texture caching and also varies with view distance and resolution. It is one of our tuning parameters.
- **Step size along rays** has similar properties as the size of a splatted voxel, but in ray space. It is also one of our tuning parameters that defines the number of overall samples. Changed caching patterns may impact performance here as well.
- **Execution time** of our ERT pre-rendering approximation step, which is actually a rough estimate of the rendering time.
- **Maximum ray depth** as a possible indicator for maximum warp/wavefront processing time. All threads (usually 32 or 64) in a single warp/wavefront run in lockstep on current GPUs, meaning that they must all stall until the last one in the warp/wavefront has finished.

Overall, those features reflect performance influencing characteristics on hardware

level, such as different texture access patterns [32]. Note that all feature are values that are already available or can be computed with minimal computational footprint, therefore being well suited for on-line prediction of our volume rendering application.

The implementation of KRLS provided by Dlib provides us with the possibility to change the parameters for the maximum number of dictionary entries (used to represent the regression function), a tolerance value, and a γ-parameter for our RBF kernel functions. We determined the following set of well working parameters by using a grid search auto tuning approach: $\gamma = 0.00025$, a tolerance of 0.006 and a dictionary limit of 10 million entries.

5.2. Analytic model: prediction of frame execution time

Combining our proxy geometry that is used for empty space skipping (see Sec. 4) with step size and image resolution, we can calculate the number of samplings we are going to make during raycasting of an upcoming frame. Therefor, we use the 2D texture that is generated during the rendering pass of our proxy geometry and generate a full mipmap stack of this texture. The topmost layer of this stack effectively contains is the average minimum and maximum depth values $\overline{d}_{\text{front}}$ and $\overline{d}_{\text{back}}$. Combined with our estimated cost per sample σ (see Sec. 5.1), we can calculate an estimate of the total frame execution time t':

$$t' = \frac{7 \cdot (\overline{d}_{\text{back}} - \overline{d}_{\text{front}})}{\Delta} \cdot \sigma.$$

Simply put, we compute the average ray length $\overline{l} = \overline{d}_{\text{back}} - \overline{d}_{\text{front}}$ (with d_{front} denoting the ray entry point and d_{back} being the estimated termination depth in ray space). Dividing the value $\overline{l}$ by our step size Δ gives us the average samplings per ray that we multiply by factor 7 (one RGBA value, plus six for gradient estimation with central differences), as well as the cost per sample σ to finally gain the estimate of the total rendering time t'.

6. Prediction-based parameter tuning

Our realtime performance prediction model provides us with the basis for various use cases. In this paper, we present two distinct scenarios for our interactive volume rendering application. First, we use our model to dynamically steer the sampling

resolution of or volume raycasting application in ray space as well as in image space, with the goal to achieve constant frame rates and thereby high responsiveness and/or execution efficiency. Second, we use our on-line predictions to dynamically distribute and balance computational load among multiple different GPUs.

6.1. Tuning of sampling resolution

Central to our sampling resolution adaption routine is the definition of a target frame rate t_{target}. By dynamically adjusting the sampling rate in ray space and/or image space, our algorithm tries to consistently achieve the target frame rate during user exploration. We use Δ as tuning parameter, that denotes the sampling resolution along a ray in the case of ray space adaption, or the image resolution in x and y direction in the case of image space adaption. We also support a hybrid approach that adapts both parameters at the same time. Basically, we follow an iterative optimization approach, using linear extrapolation and bisection during each iteration:

$$\Delta = \begin{cases} \Delta_{\text{upper}} \cdot \frac{t_{\text{target}}}{t'_{\text{upper}}} & \text{if } \tilde{t}_{\text{upper}} < t_{\text{target}} \\ \Delta_{\text{lower}} \cdot \frac{t_{\text{target}}}{t'_{\text{lower}}} & \text{if } \tilde{t}_{\text{lower}} > t_{\text{target}} \\ \Delta_{\text{lower}} + (\Delta_{\text{target}} - \Delta_{\text{lower}}) \frac{t_{\text{target}} - t'_{\text{lower}}}{t'_{\text{upper}} - t'_{\text{lower}}} & \text{else} \end{cases} \tag{2}$$

In Eq. 2, t'_{upper} and t'_{lower} denote the smallest (respective largest) estimated timing below (respective above) t_{target}. Analogously, Δ_{upper} and Δ_{lower} stand for the respective sampling resolution. We use the same approach for adjustment of the sampling resolution in image as well as in ray space. Note, that in the case of image space adaption, we normalize relative to the size of a splatted voxel, while also taking into account the quadratic image resolution adaption. For tuning of the sampling distance along the rays, we also factor in the additional samples used to evaluate gradients, which we need for local illumination. Furthermore, we employ the general assumption (as can be seen in the top two conditions in Eq. 2) that the sampling resolution has an approximately linear impact on performance. A new candidate resolution Δ in ray and/or image space is generated via a linear interpolation, as denoted in the else branch of Equation 2.

Additionally, we introduce a fixed maximum adaption of $\Delta_{max} = 0.8 \cdot \Delta$ per frame, to avoid overcompensation. This form of damping further helps to avoid lags that may

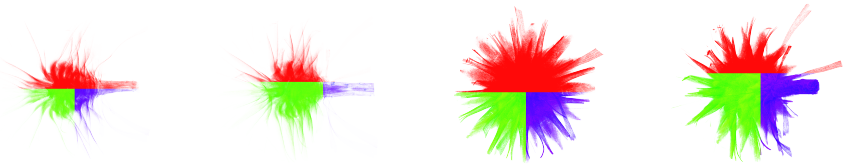


Figure 3: Load balancing distribution among three GPUs for different configurations during an interaction sequence of the Flower data set (cf. Tab. 1). Color coding by GPU: red for Titan X (Pascal), green for GTX 980 and blue for GTX 960 (cf. Tab. 2). The load distribution is adapted dynamically.

result from overestimated sampling resolution adjustments. We only use this limit while increasing resolution, because overly decreasing it does not have any negative impact on performance, only on quality. The reasoning behind this is that we consider it important for the system to be always responsive. However, if the system significantly underestimates the performance impact, it may happen that the application becomes unresponsive and therefore cannot adapt to changes until the kernel run finishes. On the other hand, if the system overestimates the performance, it can quickly re-adapt for better quality.

6.2. Load balancing

As a second use case, we employ our prediction model for load balancing in a multi-GPU setup. Conceptually, we use a separate machine learning model M_i (cf. Sec. 5.1), generating a distinct sample cost estimation σ_i for every available computing device i. We calculate the load distribution Λ_i for n computing devices using Equation 3:

$$\Lambda_i = \frac{(1-\zeta)}{n} + \zeta \cdot \frac{\prod_{j=1, j\neq i}^{n} \sigma_j}{\sum_{k=1}^{n} \sigma_k} \tag{3}$$

Basically, we multiply the sampling costs σ_i of all devices except the one thats being calculated, and divide the resulting product by the sum of all sampling costs. ζ denotes a damping factor that we use to avoid oscillation effects that are otherwise present during load balancing. By using a grid sampling approach, we determined a damping factor of $\zeta = 0.5$ to yield the best results for the tested data sets.

Volume	Resolution [voxels]	Precision [bits]	Courtesy
Chameleon	$1024 \times 1024 \times 1024$	16	UTCT
Hoatzin	$1024 \times 1024 \times 729$	16	UTCT
Kingfisher	$1024 \times 1024 \times 885$	16	UTCT
Field mouse	$1024 \times 1024 \times 975$	16	UTCT
Parakeet	$1024 \times 1024 \times 340$	16	UTCT
Zeiss	$640 \times 640 \times 640$	8	Daimler AG
Flower	$1024 \times 1024 \times 1024$	8	UZH

Table 1: Names, resolutions, and scalar precision of all volume data sets used for testing. Representative renderings are shown in Figure 4.

We partition our image space into 2D tiles with a size of 8×8 pixels each, to avoid warp/wavefront divergence (typically, warps on NVIDIA GPUs have a size of 32 threads, wavefronts on AMD GPUs 64 threads). We then use a k-d tree to distribute the tiles among the available devices, based on the average depth per tile as well as the determined load distribution Λ_i per device. For this, we use the depth values from our rendered proxy geometry texture (see Sec. 4.2), more precisely the third mipmap layer, which corresponds to our tile size. Figure 3 shows four renderings of the Flower data set during a user interaction sequence, where the image space partitioning among three distinct GPUs has been encoded via the color channel. Note that the impact of ERT significantly impacts load balancing that is dynamically adjusted during runtime.

7. Results

We evaluate our approach using multiple volume data sets presented in Table 1 (see Fig. 4 for representative renderings of the data sets) and compare the results against a volume raycaster without any parameter adjustments as well as two other adaption approaches:

- **No adapt.** A fixed step size relative to the length of a voxel as well as one pixel per ray are used for sampling. We predict the execution time of each frame using our method, but do not adjust any parameters.

Figure 4: Representative renderings of the data sets used for evaluation, ordered as in Table 1.

- **Our adapt.** We use our method to predict execution times of upcoming frames and steer the step size and/or image resolution accordingly.
- **Last frame.** Here, we adjust the sampling of the volume, based on the execution time of the last rendered frame.
- **Two pass.** Two rendering passes are conducted, as a very simple form of progressive rendering. In the first pass, a quarter of the sampling parameters from the last frame is used for rendering. In case of the execution time being lower than half of the target frame time, we linearly extrapolate the sampling parameters according to the leftover rendering budget, and render a second time.

We evaluate our approach with two different setups:

- **Single GPU system (A)** to test general performance characteristics at the example of single data sets. Evaluation includes the analysis of a frame time diagram (Sec. 7.1), the overall accuracy of our approximations and prediction (Sec. 7.2), and the computational overhead of our prediction model (Sec. 7.3).
- **Multi-GPU system (B)**, featuring three distinct GPUs to evaluate our approach including load balancing. We compare our technique against others for multiple volume data sets (Sec. 7.4), give a detailed analysis of our load balancing (Sec. 7.5), and compare adaption in image space, ray space and hybrid (Sec. 7.6).

For all adaption modes, the frame target was set to be 30 FPS (A) respective 40 FPS (B), which is generally considered to be interactive. We recorded a 30 s long sequence of user interactions using the four modes mentioned above for comparison. The sequences contain changes of the transfer function as well as rotation and zooming of the camera in an arcball-style. Rendering samples of different configuration in such a sequence are shown in Figure 5c-f. The machine learning model used for prediction is trained anew after the execution of each sequence.

We conducted all measurements on a workstation with an Intel Core i7-6700 CPU,

GPU (NVIDIA)	Cores	Clock [MHz]	Mem. [GB]	Bandw. [GB/sec]	
GTX 680	1536	1006-1058	4	192	A
Titan X (Pascal)	3584	1417-1531	12	548	B
GTX 980	2048	1126-1216	4	224	B
GTX 960	1024	1127-1178	4	112	B

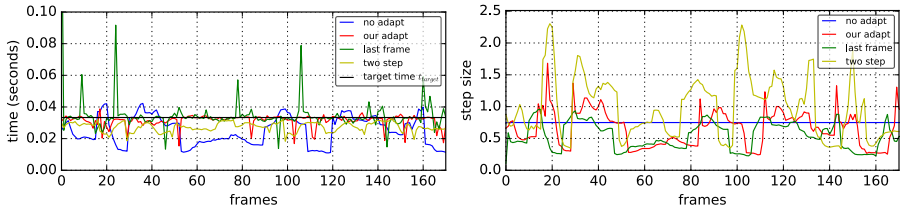
Table 2: Core specifications of the GPUs used for testing.

16 GB of RAM and either one (A) or three graphics cards (B), running Linux. Table 2 list the three GPUs we used for rendering, including specifications that give a hint on computing capabilities of the devices. The GTX 680 was used in the single GPU case, while the others were used to evaluate load balance.

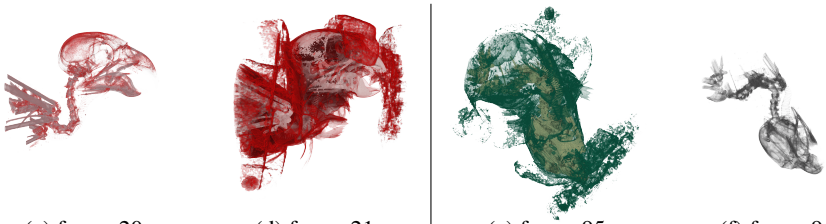
7.1. Exemplary analysis and comparison of one sequence using a single GPU

Figure 5a shows a frame time diagram for a sequence of rendering the Parakeet data set (cf. Tab. 1) in four different modes on a single GPU (scenario A). The black marker line depicts the frame target of 30 FPS. For comparison, Figure 5b shows the corresponding step size factors (relative to the voxel length) for the respective frames. We use a fixed step size of $0.75\times$ length of a voxel as step size for the mode without adaption (cf. blue line Fig. 5). As can be seen in the graph, using no adaption leads to significant deviations from the target frame rate. This is especially the case for changes of the transfer function, e.g. frame 20 and 95 (cf. renderings Fig. 5c-Fig. 5f). Even small changes to the transfer function may have significant performance impacts. This is the case, when large portions of the volume become (completely) transparent or opaque. Smaller deviations are usually caused by changes to the camera configuration that are comparably smooth during typical user interactions.

In comparison, when using adaption based on our model, the frame time stays around the target, even in the case of larger changes to the transfer function. At the same time, an overall higher sampling rate can be achieved. Note that our machine learning model is trained on the fly. However, the frame times for our approach also show a few outliers with shorter execution times. Those are mainly caused by our conservative



(a) frame times with different adaption modes (b) step sizes with different adaption modes



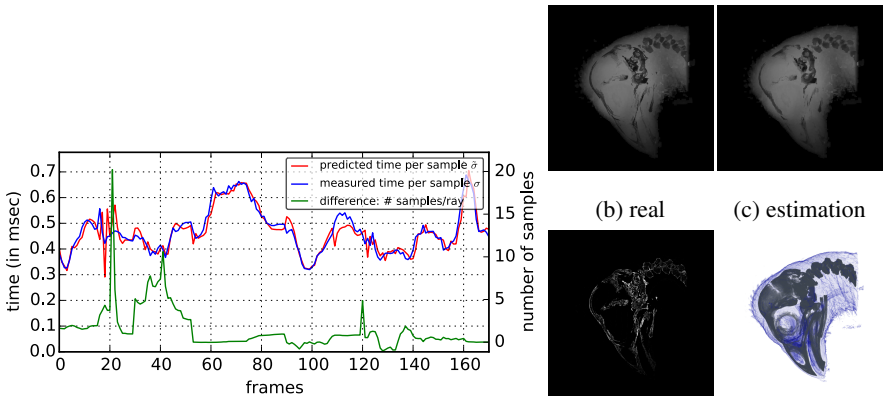
(c) frame 20 (d) frame 21 (e) frame 95 (f) frame 96

Figure 5: Plot (a) shows a single sequence at the example of the Parakeet data set. Frame times of our approach (red) are shown in comparison to methods based on last frame adaption (green), two pass adaption (yellow) and without adaption (blue). Corresponding step sizes can be found in (b), lower step size means higher quality. Example pairs of consecutive frame renderings from the sequence are depicted in (c)-(f).

adaption for higher sampling resolution (cf. Sec. 6.1). Some of the outliers can also be traced back to under- or overestimating the impact of ERT (see Sec. 7.2). Smaller deviations are probably caused by our machine-learning-based sample cost estimation σ'. Overall, there are no significant outliers with longer frame times, meaning that during the sequence, interactivity is granted, enabling a high responsiveness for the user.

In comparison, the adaption mode based on the last frame (green curve) shows huge frame time spikes that may cause poor responsiveness and jerky motions during user exploration. Those outliers are mainly caused by changes to the transfer function, which the last frame approach cannot handle by design (basically, it has one frame delay). For those cases, the sampling density is naturally higher for the last-frame-mode, while otherwise being on a similar level compared to our prediction technique.

We also compare our approach against a two-pass-mode (yellow curve). This mode has the advantage, that the frame target is hardly ever exceeded. The major drawback of this technique however, is the much lower sampling rate that eventually leads to a lower overall rendering quality. This is caused by the lower rendering budget for each frame,



(a) approximation efficiency of sample count and cost (b) real (c) estimation (d) difference $\times 4$ (e) rendering

Figure 6: Plot (a) depicts the estimation accuracy for average samples per ray (green) and sample cost σ (blue/red) at the example of the Parakeet sequence (cf. Fig. 5). Comparison of measured ray termination depths (b), and our estimation (c) (both mapped to gray values), for the Hoatzin data set (cf. Tab. 1) is shown. Subfigure (d) shows the difference of (b) and (c) with $4\times$ intensity, while (e) shows the respective rendering.

because of the time requirements of the pre-rendering pass. That means, in general for this two step approach, the available rendering time cannot be exploited fully, because the results of the first pass do not contribute to the rendering result.

7.2. Accuracy of approximations and predictions

Figure 6a shows the difference of our prediction (red) compared to the measurement (blue) of the sample cost σ of the same rendering sequence as in Section 7.1 to show the general efficiency of our approach. As can be seen, our machine learning model is able to make fairly accurate predictions of the sample cost after learning only a few samples (30 frames). The differences are also reflected in the overall prediction (cf. Fig. 5a).

Figure 6a also shows the difference between the estimated number of samples per ray and the measured number (green). Here, the discrepancies are caused by under- or overestimating the impact of early ray termination on the number of samples. This is probably because our probabilistic estimation approach may yield improper results for some difficult cases.

For investigating the efficiency of our depth estimation including ERT, Figure 6 shows the measurement (b), our estimation (c) and the difference (d) of a rendering of

Action	Required when	Maximum time [ms]
Generate histograms	Loading data set	1000
Generate bounding geometry	Transfer function changes	3.551
Generate opacity histograms	Transfer function changes	0.053
Depth assessment	Camera changes	2.680
ERT approximation	Camera changes	0.028
Training	Frame was processed	0.015
Single prediction	Adjusting resolution	0.001

Table 3: Detailed maximum runtime measurements of the steps needed for our prediction on an NVIDIA GTX 960 after a running time of more than a minute and over 1000 learned samples.

the Hoatzin data set (e). The intensity of (d) has been scaled by factor 4 to emphasize the differences. As can be seen, the overall depth estimation is fairly accurate although there are some discrepancies, mainly at the edges. Those are caused by the difference between the proxy geometry (used for our pre-rendering step) and the original high resolution data. Other differences can be caused by our stochastic methods (cf. Sec. 4.3).

7.3. Computational overhead of our prediction model

One important aspect of our prediction approach is the realtime capability. That means that the computational overhead is significantly lower than actually rendering the volume data. We explicitly designed our pipeline to provide interactive exploration capabilities, e.g. training, which is done on the CPU, and pre-processing, run interleaved. Furthermore, it is conceptually possible to do the pre-processing steps (i.e. depth assessment and ERT approximation) on a different device than the actual frame computation. For instance, one could use a GPU that is integrated in the CPU for the pre-processing steps and one or several dedicated graphics cards for rendering. Table 3 gives an overview of the processing times upper bounds of the various steps needed to perform our prediction for the chameleon data set (cf. Tab. 1) with 1024^3 voxels on the NVIDIA GTX 960 (the slowest tested GPU). Figure 1 gives a structured overview on the various steps.

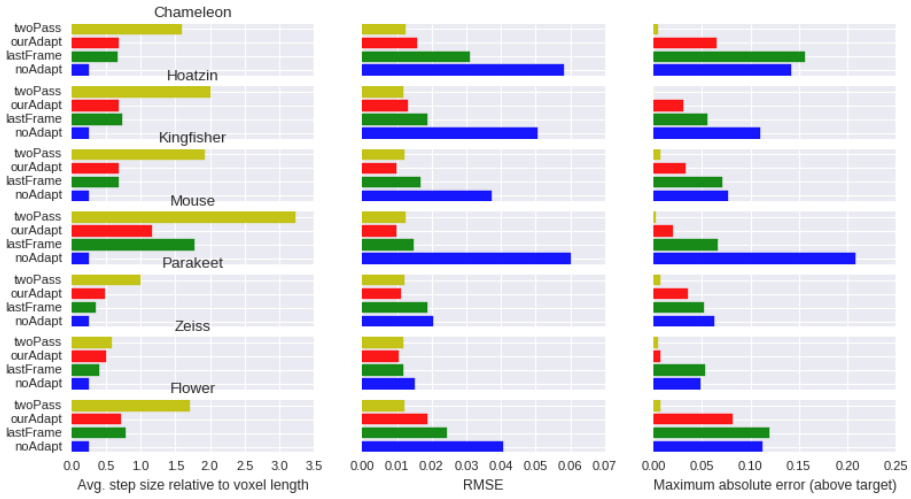


Figure 7: Results of sequence measurements for different volumes. A smaller average step size factor indicates better quality, while a lower RMSE indicated a better performance (i.e., closer to the FPS target). Maximum absolute errors indicate worst case performance.

As listed in Table 3, even for a single midrange GPU, the computational overhead for our prediction model is comparably low. For the tested volume with a resolution of 1024^3 voxels, generating the bounding geometry (only needed when changing the transfer function) and performing the depth assessment has a combined execution time of about 6 ms, a fraction of an interactive frame time. Training and prediction are even significantly faster. Overall, the measurements show an acceptable computational overhead, that implies a smoothly on-line usage of our prediction method for average workstations.

7.4. Evaluation of multiple rendering sequences

We conducted detailed measurements of interaction sequences for seven different volume data sets listed in Table 1. Individual sequences were created for each data set separately, to capture different scenarios and last 30 s each. In Section 7.1, we give an exemplary, detailed analysis of one of these sequences. In contrast to the sequence presented in Section 7.1 however, we conducted the measurements discussed in this Section using system setup (B) with three GPUs (cf. Tab. 2) and our load balancing enabled. All following measurements were conducted on this system. Integration step

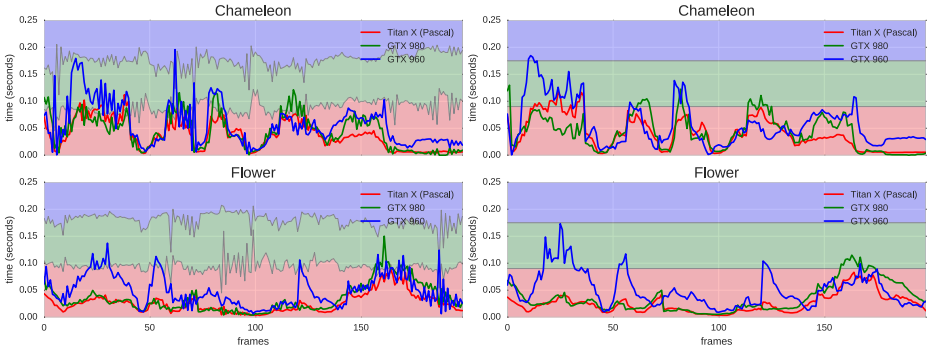
size was set to 0.25× voxel length, for the mode without adaption and the target frame rate was set to be 40 FPS (0.025 ms frame time). The idea behind those changes is to increase rendering complexity to stress the three GPUs sufficiently. The viewport is the same with 1024^2 pixels and we adjust the ray sampling size for the three adaption modes.

Figure 7 gives an overview of the results obtained during our measurement series. As an indicator of the rendering quality we show the average step size along rays during the whole sequence (smaller is better). To judge the prediction accuracy, we use the root-mean-square error (*RMSE*), as a measure for the difference of predictions ($\hat{y}_t$) and measurements (y_t) across a sequence with n frames:

$$RMSE = \sqrt{\frac{\sum_{t=1}^{n} (\hat{y}_t - y_t)^2}{n}}. \tag{4}$$

As a third quantity, we show the maximum absolute error above the frame target, meaning the biggest absolute discrepancy between the target execution time and the measured time. This is an indicator of the worst case performance, as a high error usually results in lags or jerky motions during user exploration. One of the main goals of our technique is to avoid such high frame times.

As can be seen in Figure 7, our approach (red) has a comparably low *RMSE* (meaning a low deviation from the target frame rate) with only the two-pass-technique (yellow) being better in some cases. At the same time, the step size (rendering quality) is significantly better than the two-pass-mode in all cases and only slightly worse than the last-frame-mode (green) in some cases. The mode without any step size adaptations (blue) performs best in terms of rendering quality, but has a significant deviation from the frame target for all sequences. The worst case performance of the two-pass-mode (cf. Sec 7.1) can be observed in the maximum absolute error, while our approach performs fairly well in this regard. Naturally, the two step mode is better than ours. The results reflect the ones observed and discussed in Section 7.1 and the reasonings are basically the same. Overall, our measurements show, that our technique keeps a good balance between rendering quality and speed while guaranteeing responsiveness during interactive exploration of the volume data set.



(a) Our load balancing (b) Static balancing (empirical)

Figure 8: Comparison of frame times (solid lines) and distribution (stacked areas) among three different GPUs: one shows measurements with dynamic load balancing based our prediction (a); the other one depicts a static load distribution (b), based on empirically determined average sample costs $\bar{\sigma}_i$ per device.

7.5. Load balancing

We compare our load balancing against a static distribution, based on all our empirical measurements, i.e. the average sample cost over all measured data sets respective sequences. Figure 8 shows the frame diagrams of the Chameleon and the Flower data sets (cf. Tab. 1) for those two modes. In the plot, the solid lines depict the frame times for the three tested GPUs (cf. Tab. 2), while the stacked semi-transparent plots show the (dynamic) relative load distribution among the GPUs.

Although the overall trend is the similar, as can be seen in Figure 3, our load balancing generally outperforms the static approach. However, the dynamic balancing is not without flaws. Immediately noticeable is the oscillation pattern, which is common to load balancing methods. We try to counter this effect with our damping factor. We found a factor of 0.5 to work best across the tested volumes, while selecting a higher value resulted in converging of the load balancing and static versions, while lower values worsen the oscillation. In addition, the problems of our model (discussed in Sec. 7.1) also show in the load balancing. Yet throughout all tested volumes, the achieved load distribution efficiency (i.e., how well the timings of all GPUs match on average) is about 18% better for our load balancing approach, compared to the static distribution. We could show that our prediction approach is well-suited for load balancing. However,

Adaption type	MSE	SSIM	PSNR
Image space	34.256	0.981	37.215
Ray space	20.563	0.988	40.237
Hybrid	20.170	0.992	40.747

Table 4: Image metrics for adaption in image space, ray space and hybrid (both), relative to a reference rendering without adaption of the Zeiss data set. All values are averaged over the whole sequence.

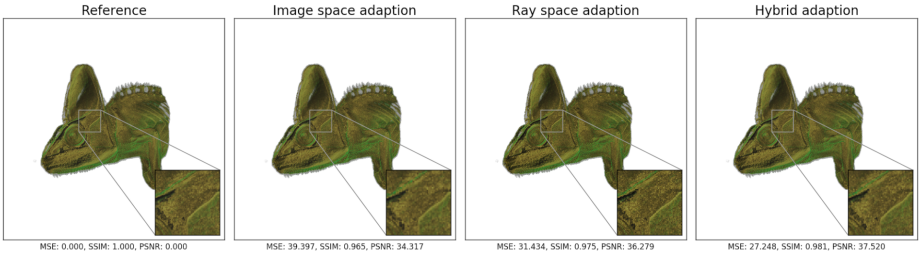


Figure 9: Comparison of adaption in image space vs. ray space vs. a hybrid for one frame of the Chameleon data set rendering sequence.

improvements w.r.t. damping and prediction accuracy could further improve the results.

7.6. Image vs. ray space adaption

Using our prediction approach, we can dynamically steer the rendering quality by adapting the sampling resolution in ray space and/or in image space. In this section, we present our evaluation of the image quality when adapting the resolution only in ray space vs. only in image space as well as a hybrid approach where we change both parameters at the same time. We use three measures to evaluate the image rendering quality: the mean square error (MSE), which is basically the same as the RMSE (see Eq. 7.4) but without the square root, the structural similarity (SSIM) [33], and the peak signal-to-noise ratio (PSNR). Table 4 gives average numbers for the three measures during the whole sequence of the Zeiss data set (cf. Tab. 1).

Figure 9 shows a direct comparison of a single frame for the three methods against a reference rendering. The reference is rendered with a image resolution of 1200^2 pixels and an integration step size of $0.1\times$ voxel length, the adapted sampling resolution is at most 1024^2 rays respective a step size of $0.25\times$ voxel length. As can be seen in Figure 9

and Table 4, the ray space adaption generally outperforms the image space one, in all metrics. The hybrid approach yields similar results or slightly outperforms ray space adaption. The exemplary renderings in Figure 9 show that a low integration step size in ray space may result in sampling artifacts, that can be partly avoided when using the hybrid adaption and therefor lead to higher image quality. This is especially the case for data sets containing sharp edges and thin surfaces where under-sampling may occur more easily, as is the case for the Zeiss data set (cf. Tab. 4/ Fig. 4).

7.7. Discussion of limitations and extensibility

We demonstrated that our technique works well for the tested datasets and we are able to stay within the range of a user-defined frame target even for difficult to predict cases as well as use the prediction for load balancing between different GPUs. However, our approach has also some limitations. For some cases, especially for high step sizes, our model produces a slightly worse predictions than the two-pass or last-frame-modes. This is mainly because of an underestimation of ERT impact and usually results in a faster frame time. Those naturally do not cause any lags in interactivity but may result in a slightly lower rendering quality. Furthermore, our machine learning model sometimes produces inaccurate predictions for the sample cost, that we assume to be the result of not yet learned configurations. Those inaccuracies also influence the load balancing performance.

Possible measures to meet those issues in the future could be to include uncertainty handling, e.g. in the form of an additional feature that evaluates how different ("far away") a new configuration is from all previously learned samples. Another possibility would be to perform a short learning run after loading a data set that covers several important configurations. However, it is hard to determine which configurations are interesting, because it highly depends on the data set. A third option to improve the quality of the prediction would be to transfer learned features between data sets, which is not possible for the currently used KRLS machine learning algorithm. As shown in Section 7.1, even after a short period of time, the algorithm already shows a high prediction accuracy.

The load balancing use case shows promising results. Raising the general prediction

accuracy (as mentioned above) could allow for a lower damping factor and thereby improve the balancing. A general drawback of this approach is the mandatory availability of the data on all devices to avoid costly transfers via the PCI-e bus, so sufficient memory on all devices is required. Adaption of the sampling resolution in image space seems to result in inferior quality opposed to resolution adjustment in ray space. However, a hybrid adaption can improve the perceived quality, especially for data sets with sharp edges and thin structures, where an under-sampling in ray space may lead to artifacts. A more refined sampling technique in image space (e.g., with non-regular patterns) could possibly improve the results for this adaption method.

8. Conclusion

We presented an integrated approach for predicting rendering performance of a volume raycaster on-the-fly and use this to perform load balancing as well as dynamic tuning of the sampling resolution. Using our technique, we can adapt the image sampling and distribute computational load among different devices, to significantly reduce lags and jerky motions during interactive exploration. To overcome those unpleasant effects, we proposed methods to explicitly assess the impact of acceleration techniques on the raycasting performance. Thereby we also employ a novel technique to estimate the effect of early ray termination. We introduced a hybrid performance prediction model that is capable of predicting accurate frame execution times on-the-fly. The model consists of two parts: in an analytical fashion, we uses the assessed acceleration data together with general information on the data set and sampling density for a depth estimation, and combine this with an estimate of the cost per sample by using a machine learning technique. We demonstrate the usability of our technique by means of two use cases. For the first one, we adjust the sampling density in ray space and/or image space to reliably meet user-defined performance requirements. This resulted in stable frame execution times, even for sudden large changes to the transfer function, while at the same time keeping rendering quality at a high level by adjusting sampling resolutions according to predicted performance requirements. The second one is prediction-based load balancing among multiple GPUs, with the goal to consistently maximize hardware

usage and thereby improve rendering speed and quality.

In future work, we want to further improve the generality of our approach by integrating different data representations, interactive illumination design, and additional acceleration techniques. Additionally, we aim to integrate a form of uncertainty handling to further improve our predictions. As a next step, we also think of evaluating the transferability of our model to other rendering techniques.

Acknowledgments

We would like to thank the German Research Foundation (DFG) for supporting the project within project A02 of SFB/Transregio 161.

References

[1] B. Laha, K. Sensharma, J. D. Schiffbauer, D. A. Bowman, Effects of immersion on visual analysis of volume data, IEEE Transactions on Visualization and Computer Graphics 18 (4) (2012) 597–606. doi:10.1109/TVCG.2012.42.

[2] V. Bruder, S. Frey, T. Ertl, Real-time performance prediction and tuning for interactive volume raycasting, in: SIGGRAPH ASIA 2016 Symposium on Visualization, ACM, 2016, p. 7.

[3] K. Engel, M. Hadwiger, J. Kniss, C. Rezk-Salama, D. Weiskopf, Real-time volume graphics, CRC Press, 2006.

[4] M. Hadwiger, P. Ljung, C. R. Salama, T. Ropinski, Advanced illumination techniques for gpu-based volume raycasting, in: ACM SIGGRAPH 2009 Courses, SIGGRAPH '09, ACM, New York, NY, USA, 2009, pp. 2:1–2:166. doi:10.1145/1667239.1667241.
URL http://doi.acm.org/10.1145/1667239.1667241

[5] J. Beyer, M. Hadwiger, H. Pfister, State-of-the-art in gpu-based large-scale volume visualization, in: Computer Graphics Forum, Vol. 34, Wiley Online Library, 2015, pp. 13–37.

[6] K.-L. Ma, J. S. Painter, C. D. Hansen, M. F. Krogh, Parallel volume rendering using binary-swap compositing, IEEE Computer Graphics and Applications 14 (4) (1994) 59–68.

[7] S. Marchesin, C. Mongenet, J.-M. Dischler, et al., Dynamic load balancing for parallel volume rendering., in: EGPGV, 2006, pp. 43–50.

[8] T. Fogal, H. Childs, S. Shankar, J. Krüger, R. D. Bergeron, P. Hatcher, Large data visualization on distributed memory multi-gpu clusters, in: Proceedings of the Conference on High Performance Graphics, HPG '10, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 2010, pp. 57–66.
URL http://dl.acm.org/citation.cfm?id=1921479.1921489

[9] C. Müller, M. Strengert, T. Ertl, Optimized Volume Raycasting for Graphics-Hardware-based Cluster Systems, in: Eurographics Symposium on Parallel Graphics and Visualization, Eurographics Assoc., 2006, pp. 59–66.

[10] H.-W. Shen, C. R. Johnson, Differential volume rendering: a fast volume visualization technique for flow animation, in: Proceedings of the conference on Visualization '94, 1994, pp. 180–187.

[11] H. Qu, M. Wan, J. Qin, A. Kaufman, Image based rendering with stable frame rates, in: Proceedings of the IEEE Conference on Visualization '00, 2000, pp. 251–258.

[12] G. Wong, J. Wang, Real-time rendering: Computer graphics with control engineering, CRC Press, Boca Raton, FL, 2014.

[13] C. Woolley, D. Luebke, B. Watson, A. Dayal, Interruptible rendering, in: Proceedings of the 2003 Symposium on Interactive 3D Graphics, 2003, pp. 143–151. doi:10.1145/641480.641509.
URL http://doi.acm.org/10.1145/641480.641509

[14] S. Frey, F. Sadlo, K.-L. Ma, T. Ertl, Interactive progressive visualization with space-time error control, IEEE transactions on visualization and computer graphics 20 (12) (2014) 2397–2406.

[15] S. Sodhi, J. Subhlok, Q. Xu, Performance prediction with skeletons, Cluster Computing 11 (2) (2008) 151–165. doi:10.1007/s10586-007-0039-2.
URL http://dx.doi.org/10.1007/s10586-007-0039-2

[16] B. J. Barnes, B. Rountree, D. K. Lowenthal, J. Reeves, B. de Supinski, M. Schulz, A regression-based approach to scalability prediction, in: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2008, pp. 368–377. doi:10.1145/1375527.1375580.
URL http://doi.acm.org/10.1145/1375527.1375580

[17] M. M. Tikir, L. Carrington, E. Strohmaier, A. Snavely, A genetic algorithms approach to modeling the performance of memory-bound computations, in: Proceedings of the ACM/IEEE Conference on Supercomputing, ACM, New York, NY, USA, 2007, pp. 47:1–47:12. doi:10.1145/1362622.1362686.
URL http://doi.acm.org/10.1145/1362622.1362686

[18] B. C. Lee, D. M. Brooks, B. R. de Supinski, M. Schulz, K. Singh, S. A. McKee, Methods of inference and learning for performance modeling of parallel applications, in: Proceedings of the SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM, NY, USA, 2007, pp. 249–258. doi:10.1145/1229428.1229479.
URL http://doi.acm.org/10.1145/1229428.1229479

[19] T. Hoefler, W. Gropp, W. Kramer, M. Snir, Performance modeling for systematic performance tuning, in: State of the Practice Reports, SC '11, ACM, New York, NY, USA, 2011, pp. 6:1–6:12. doi:10.1145/2063348.2063356.
URL http://doi.acm.org/10.1145/2063348.2063356

[20] S. Madougou, A. Varbanescu, C. de Laat, R. van Nieuwpoort, The landscape of gpgpu performance modeling tools, Parallel Computing 56 (2016) 18 – 33. doi:http://dx.doi.org/10.1016/j.parco.2016.04.002.

[21] M. Amarís, R. Y. de Camargo, M. Dyab, A. Goldman, D. Trystram, A comparison of gpu execution time prediction using machine learning and analytical modeling,

in: Network Computing and Applications (NCA), 2016 IEEE 15th International Symposium on, IEEE, 2016, pp. 326–333.

[22] I. Bowman, J. Shalf, K.-L. Ma, W. Bethel, Performance modeling for 3d visualization in a heterogeneous computing environment, Lawrence Berkeley National Laboratory.

[23] M. Wimmer, P. Wonka, Rendering time estimation for real-time rendering, in: Proceedings of the 14th Eurographics workshop on Rendering, Eurographics Association, 2003, pp. 118–129.

[24] N. Tack, F. Morán, G. Lafruit, R. Lauwereins, 3d graphics rendering time modeling and control for mobile terminals, in: Proceedings of the ninth international conference on 3D Web technology, ACM, 2004, pp. 109–117.

[25] P. Ganestam, M. Doggett, Auto-tuning interactive ray tracing using an analytical gpu architecture model, in: Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units, ACM, 2012, pp. 94–100.

[26] S. Rizzi, M. Hereld, J. A. Insley, M. E. Papka, T. D. Uram, V. Vishwanath, Performance modeling of vl3 volume rendering on gpu-based clusters, in: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization, 2014, pp. 65–72. doi:10.2312/pgv.20141086.
URL http://dx.doi.org/10.2312/pgv.20141086

[27] S. Eilemann, R. Pajarola, Direct send compositing for parallel sort-last rendering, in: Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization, Eurographics Assoc., Aire-la-Ville, Switzerland, 2007, pp. 29–36. doi:10.2312/EGPGV/EGPGV07/029-036.
URL http://dx.doi.org/10.2312/EGPGV/EGPGV07/029-036

[28] H. Yu, C. Wang, K.-L. Ma, Massively parallel volume rendering using 2-3 swap image compositing, in: Proceedings of the ACM/IEEE Conference on Supercomputing, IEEE Press, Piscataway, NJ, USA, 2008, pp. 48:1–48:11.
URL http://dl.acm.org/citation.cfm?id=1413370.1413419

[29] H. Nguyen, Gpu Gems 3, 1st Edition, Addison-Wesley Professional, 2007.

[30] Y. Engel, S. Mannor, R. Meir, The kernel recursive least-squares algorithm, IEEE Transactions on signal processing 52 (8) (2004) 2275–2285.

[31] D. E. King, Dlib-ml: A machine learning toolkit, Journal of Machine Learning Research 10 (2009) 1755–1758.

[32] E. W. Bethel, M. Howison, Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning, International Journal on High Performance Computing Appllications 26 (4) (2012) 399–412. doi:10.1177/1094342012440466.
URL http://dx.doi.org/10.1177/1094342012440466

[33] Z. Wang, A. C. Bovik, H. R. Sheikh, E. P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE transactions on image processing 13 (4) (2004) 600–612.