Real-Time Performance Prediction and Tuning for Interactive Volume Raycasting

Valentin Bruder*, Steffen Frey and Thomas Ertl
University of Stuttgart

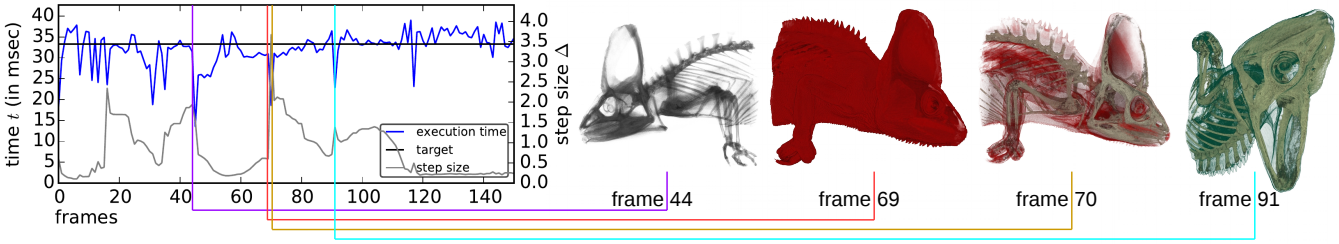


Figure 1: *Based on predictions of the raycasting time (blue) of upcoming frames, the integration step size (gray) is adjusted to meet a target frame rate during interactive exploration of the volume data set. For instance between frame 69 (red) and frame 70 (yellow), a great alteration of the transfer function causes a significant change of the rendering performance that we predict and react in quintupling the step size to avoid a visible lag. We aim for a constant frame rate for different view configurations (e.g., frames 44, 69, 91) and transfer functions (e.g., 44, 69, 70).*

Abstract

We present an integrated approach for the real-time performance prediction and tuning of volume raycasting. The usage of empty space skipping and early ray termination, among others, can induce significant variations in performance when camera configuration and transfer functions are adjusted. For interactive exploration, this can result in various unpleasant effects like abruptly reduced responsiveness or jerky motions. To overcome those effects, we propose an integrated approach to accelerate the rendering and assess performance-relevant data on-the-fly, including a new technique to estimate the impact of early ray termination. On this basis, we introduce a hybrid model, to achieve accurate predictions with only minimal computational footprint. Our hybrid model incorporates both aspects from analytical performance modeling and machine learning, with the goal to combine their respective strengths. Using our model, we dynamically steer the sampling density along rays with our automatic tuning technique. This approach allows to reliably meet performance requirements like a fixed frame rate, even in the case of large sudden changes to the transfer function or the camera. We finally demonstrate the accuracy and utility of our approach by means of a variety of different volume data sets and interaction sequences.

Keywords: volume raycasting, performance prediction

Concepts: •**Human-centered computing** → *Scientific visualization;*

*e-mail: valentin.bruder@visus.uni-stuttgart.de

SA '16 Symposium on Visualization, December 05 - 08, 2016, Macao
ISBN: 978-1-4503-4547-7/16/12
DOI: http://dx.doi.org/10.1145/3002151.3002156

1 Introduction

Volume visualization is a crucial tool for the visual analysis of measured and simulated data in numerous fields such as medicine, engineering, etc. Most importantly, dynamic interaction with the visualization allows for an exploratory approach that enables users to gain insight beyond the original focus. In volume visualization, classic user interactions are changes to the camera configuration (e.g., rotation and zooming) and adjustments to the transfer function. For a satisfying experience during such interactions, low latencies and high frame rates are crucial, while at the same time the achieved image quality needs to be as good as possible. In particular the achievement of low latencies gains even more importance in the context of the recent emergence of virtual reality for scientific applications [Laha et al. 2012]. Here, requirements with respect to responsiveness are very strict to avoid unpleasant side effects for the user. To achieve the required level of performance, hardware capable of massively parallel execution is often used nowadays (in particular GPUs). The achieved performance of volume raycasting not only depends on the used hardware, but there is also a significant impact due to interactively changed parameters (i.e., transfer function and camera configuration). These variations need to be accounted for to accomplish interactivity, even for challenging cases with significant changes between frames. This can be done by adjusting cost-quality trade-off parameters like the sampling density along the rays or the image plane. For an interactive application, however, the basis for this needs to be some kind of assessment how the performance will evolve in the next frame (after potentially big changes) in order to avoid disturbingly long response times or jerky motions. Predicting the performance of such an application on this hardware, however, is a challenging task due to the involved complexity. There is a variety of factors that have a significant performance impact, ranging from the used hardware over specifics of the employed algorithms, to parameter adjustments due to user interactions. While a lot of research on performance prediction has been conducted in recent years, most of it targets scientific applications in high performance computing (HPC) environments. To the best of the authors' knowledge, predicting the performance of an interactive volume rendering application on-the-fly has not been addressed so far.

In this paper, we propose our method to dynamically predict the performance of a volume rendering application using popular acceleration techniques. We then use this as a basis for dynamically adjusting the volume rendering process to reliably achieve inter-

active frame rates. In the following, we first give an overview on related work (Sec. 2) and then discuss what we consider to be the main contributions of our work.

- We present our overall approach for performance prediction and the tuning of sampling densities along rays (Sec. 3). It is based on the following components:
- the assessment of performance-critical numbers for raycasting acceleration techniques, partially including the impact of early ray termination (ERT) (Sec. 4);
- the prediction of the rendering performance for upcoming frames on the fly using a hybrid performance model, (Sec. 5);
- and the corresponding steering of rendering quality in real-time toward a user-specified frame rate (Sec. 6).

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

2 Related Work

Volume Visualization. Salama et al. [2009] discuss basic volume rendering techniques, with a focus on illumination. They describe object-order empty space skipping in detail, an acceleration technique we employ as well. Beyer et al. [2015] give an overview of the current state of the art in GPU techniques for interactive large-scale volume visualization. Kratz et al. [2011] use an error estimator from the field of finite element methods for adaptive screen-space sampling. Qu et al. [2000] and Shen and Johnson [1994], among others, exploit frame coherence by re-utilizing pixel values from the previous frame by warping positions to the current frame and use that to gain stable frame rates. In general, rendering systems typically fix either image quality or frame rate during user interaction [Autodesk, Inc. 2014]. For real-time rendering, Wong and Wang [2014] model the image generation process by an open-loop approach underpinned by constraints and estimations of its constituents with the goal to achieve as constant frame rates as possible. Their heavy-weight approach describes different rendering processes in detail with all their complexities involved to gain a nonlinear model to relate inputs and outputs using neural networks and fuzzy models. In contrast, Woolley et al. [2003] use simple metrics based on image-space distances to steer progressive raytracing. Frey et al. [2014] steer the volume visualization process in three major degrees of freedom: When should the refinement of a frame in the background be terminated and when should a new one be started; when should a frame that is currently being computed be displayed and how many resources should it consume? These decisions are based on the correlation of the errors due to insufficient sampling and response delay.

Performance Prediction. Predicting and modeling application performance for parallel architectures is an active field of research in computer architecture and high-performance computing. Numerous approaches have been proposed for performance modeling, including regression [Barnes et al. 2008], genetic algorithms [Tikir et al. 2007], performance skeletons [Sodhi et al. 2008] and machine learning [Lee et al. 2007]. The data that is being used for performance modeling stems either from empirical measurements (e.g., execution times or performance counters) or from hardware parameters such as floating point operations per second or memory transfer rate. Using former combined with an analytical model results in a so called "semi-empirical" model [Hoefler et al. 2011]. We use an analytical model based on known attributes of the volume raycasting algorithm as well as using observed execution times for fitting. Therefore our approach falls into this category. The approaches mentioned above, primarily target performance prediction in large-scale systems. The main difference between large HPC codes and visual computing applications is interactivity. Offline performance predictions for GPUs has been the target of recent research as well. Baghsorkhi et al. [Baghsorkhi et al. 2010] use an analytical model for predictions and as a basis for an auto-tuning compiler to find bottlenecks in kernel implementations. Hong and Kim [2009] also use an analytical model in combination with CUDA in their work. They predict kernel execution times based on the number of parallel memory requests. Kothapalli et al. [2009] developed a model for CUDA which takes many architectural specifics into account, such as scheduling, memory hierarchy, and pipelining. Through an asymptotic analysis approach, code performance prediction is performed. However, all these models focus on offline prediction and are in parts limited to certain platforms, programming frameworks or architectures.

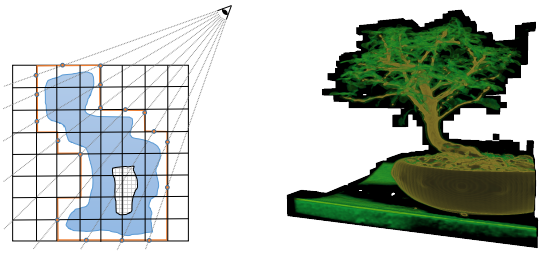
Work on real-time rendering incorporating performance prediction is relatively sparse. What has been proposed mostly focuse either on object-order rendering algorithms [Wimmer and Wonka 2003; Tack et al. 2004] or on creating a performance model for a visualization pipeline [Bowman et al. 2004]. Therefore their rendering tasks are different from ours. Specifically for parallel volume rendering, Rizzi et al. [2014] recently presented an analytical model for offline prediction of scaling behavior on GPU clusters. The model basically sums up timing predictions, made for each part of the overall procedure. Other works have concentrated exclusively on one step of parallel volume rendering, namely the compositing of images from different nodes [Eilemann and Pajarola 2007; Yu et al. 2008]. Among others, a theoretical performance analysis was conducted for this step and compared with the actual results. Our method differs from the approaches mentioned above, in that we focus on real-time performance prediction targeting volume visualization as an image-order algorithm in a workstation environment.

Algorithm 1 Overview over our approach for the dynamic performance prediction and steering of volume raycasting.

```
1:  procedure DynamicPredictionSteeringRaycasting
2:    loop                                              ▷ infinite interaction loop
3:      if δ(V) then                                    ▷ volume V changed
4:        H ← BlockHistogram(V)                         ▷ block histogram H (Sec. 4.1)
5:      if δ(T) ∨ δ(H) then     ▷ transfer function T or histogram H changed
6:        H_α ← T_α(H)          ▷ apply transfer function to histogram (Sec. 4.1)
7:        G ← BoundingFaces(H_α)                        ▷ bounding geometry G (Sec. 4.2)
8:        D ← Depth_ Assessment(G)                      ▷ rasterize G (Sec. 4.2)
9:        D̃_ERT ← Raycasting_ERT(H_α, D, Δ_ERT)         ▷ (Sec. 4.3, Alg. 2)
10:       loop          ▷ iterative adjustment of step size Δ w.r.t. prediction t̃
11:         t̃ ← PerformanceEstimation(D̃_ERT, Δ, M)      ▷ (Sec. 5)
12:         if t̃ ≈ t_target then          ▷ exit when t_target is met by prediction t̃
13:           break
14:         Δ ← StepSizeAdjustment(t̃, t_target)         ▷ adjust Δ (Sec. 6)
15:       (t, D_ERT) ← Raycasting_VOL(T(V), D, Δ)       ▷ (Sec. 4.3, Alg. 2)
16:       M ← UpdateModel(M, t, Δ, D_ERT)               ▷ update model M (Sec. 5)
```

3 Overview

We do volume raycasting, accelerated through early ray termination (ERT) and object-order empty space skipping. We predict the performance of the upcoming frame and adjust the sampling density along the rays based on that. Alg. 1 gives an overview on our approach. Overall, it runs an interaction loop (Lines 2–16), in which the user may interactively change the volume data, transfer function, and camera configuration. Our volume is subdivided into blocks of regular size (we employ a block resolution of 16^3 voxels throughout this paper). Various steps of our approach employ density histograms H of volume blocks. Each histogram in H represents the distribution of scalar values in its respective volume block. If the volume changes (or we are in our first iteration), we need to update H accordingly (Line 4). Then, if the transfer function T was adjusted by the user or if the histogram H changed, we need to update our derived opacity histogram H_α. Like H, it also individually represents different vol-



(a) *empty space skipping* **(b)** *bounding geometry (black)*

Figure 2: *Illustration of object-order empty space leaping (a): A bounding geometry (orange) on a coarse grid defines ray entry and exit points. (b) shows the generated bounding geometry (black) at the example of the Bonsai data set (courtesy S. Roettger).*

ume blocks, but represents opacity instead of density values. H_α is generated by applying the opacity channel T_α of the transfer function to H (Line 6). With this, we then generate bounding geometry G (Line 7) that is subsequently used for object-order empty space skipping (Line 8). This works by simply rasterizing G (via OpenGL) and determining the depth D of the frontmost and the backmost fragment (D_{front} and D_{back}, respectively). As input for our prediction, we then further adjust the depths D_{back} to $\tilde{D}_{\text{ERT}}$ in order to incorporate the estimated effects of ERT (Line 9).

Our performance prediction model provides the basis for dynamically adjusting parameters of applications, e.g. with the goal to achieve high execution efficiency and/or responsiveness. One potential use case could be load balancing, in which a fast prediction of changes in the computational and/or transfer load is required. However, in the context of this paper, we look at a single-node interactive application and define a certain target frame rate t_{target} that we aim to achieve during user exploration. As a parameter, we adjust the sampling step size Δ in ray space to achieve this. For this, we basically follow an iterative optimization approach (Lines 10–14). In each iteration, we first estimate the time $\tilde{t}$ that would be achieved with a step size Δ (on the basis of $\tilde{D}_{\text{ERT}}$ and our model M). If the prediction $\tilde{t}$ is close to our target value t_{target}, we are done with step size adjustment (Lines 12 & 13). Otherwise, we generate a new step size candidate for Δ (Line 14). Next, we actually carry out the raycasting procedure using the obtained value for Δ (Line 15). Finally, we update our performance prediction model M, both with the measured execution time t as well as the determined depth D_{ERT} after ERT (Line 16), assessed during the actual raycasting.

4 Collection of Performance-Relevant Data

In this section, we discuss our approach to assess performance-relevant data w.r.t. volume raycasting acceleration techniques. In the context of this paper, we specifically consider two widely used techniques: object-order empty space skipping and early ray termination (ERT). As the basis for both, we partition the volume into blocks (of 16^3 voxels), and compute a respective histogram (Sec. 4.1). These histograms are then used to determine the depth segment D that is sampled by each ray during raycasting (not considering ERT) (Sec. 4.2). Here, the results are both used directly for the actual raycasting as well as the prediction. Finally, we discuss how ERT is applied during raycasting (it does not rely on any previously computed information), and particularly how we estimate its impact beforehand, on the basis of volume block histograms H_α (Sec. 4.3).

4.1 Histogram of Volume Blocks (H and H_α)

First of all, we logically partition our volume V into blocks of 16^3 voxels. For each of these blocks, we compute a histogram H of the contained scalar values. In our implementation, we employ a histogram H consisting of 64 bins. From H, we then compute the opacity H_α by applying the current (opacity) transfer function $T_\alpha : \mathbb{R} \to \mathbb{R}$. Note that H_α only features 16 bins for the sake of efficiency during ERT estimation (c.f. Sec. 4.3). Generating H_α from H basically works by looping over all bins b in H. Each bin b represents a density range $[v_{\text{min}}, v_{\text{max}}]$. From this density range, we compute a respective opacity value b_α by integrating over the range $[v_{\text{min}}, v_{\text{max}}]$ with the transfer function $T_\alpha(b)$. The resulting opacity value b_α for the bin, is then used to identify the respective histogram bin in H_α, to which we then add the number of of elements that correspond to the original bin $b \in H$.

4.2 Depth Assessment (D)

The extent of empty areas in a volume can vary strongly depending on both, the volume characteristics and the transfer function. There are two different conceptional approaches of skipping these parts: image-order and object-order (also hybrid approaches have been proposed, e.g. [Scharsach et al. 2006]). We use object-order empty space skipping (e.g. [Salama et al. 2009]) based on our opacity-mapped histogram H_α (c.f. Sec. 4.1). The object-order approach uses a pre-processed bounding geometry to determine more accurate entry and exit points of the viewing rays than the – otherwise normally used – bounding box. We employ our opacity-mapped block histogram H_α to determine for each block, whether it is empty or not (i.e. whether not all entries of H_α are in the bin for fully transparent voxels). Generating a surface of quads around the nonempty, outer blocks of the coarse volume representation, yields a bounding geometry that is then rasterized. Fig. 2 shows (a) the schematic approach as well as (b), a rendered bounding geometry. We use blending with a minimum function to write the minimum as well as the negative maximum depth values into a texture (therefore only a single render pass is required to obtain both values). This pass has to be done when the camera configuration changes, while re-building the bounding geometry is only needed after a change of the transfer function. When using this technique, empty space inside the volume (and, consequently, the bounding geometry) cannot be skipped. Note that more complex approaches exist to handle multiple input and output points for each ray, however, these typically also exhibit a higher cost overhead. With minor adjustments, such techniques could be used as a drop-in replacement for our current solution.

4.3 Early Ray Termination (D_{ERT} & $\tilde{D}_{\text{ERT}}$)

Depending on data set and transfer function, early ray termination (ERT) can lead to huge gains in performance. While applying ERT during raycasting is very simple, the a-priori estimation is comparably difficult as this cannot be solved locally (e.g., for a voxel or block on its own) in general, but actually requires the consideration of the accumulated opacities so far. To achieve this in an efficient manner, we implement our estimation of the impact of ERT on the sampled depth segment D as a variant of the basic volume raycasting procedure. Alg. 2 outlines this procedure as well as our general raycasting algorithm using two colors as encoding. The orange colored parts indicate an execution for the ERT estimation only (Alg. 1, Line 11) while the blue colored ones are only carried out during the actual volume rendering (Alg. 1, Line 15).

In the setup phase, we initialize the accumulated opacity (Line 2) and color (only required regular volume rendering, as denoted via blue color) (Line 3). For our ERT estimation, we basically take the thread number as a seed for our pseudo-random number generator (PRNG) (Line 4). In the following, we basically sample along our respective ray R in a front-to-back manner using step size Δ from D_{front} to D_{back} as determined by our depth assessment (Line 5, c.f.

Algorithm 2 Front-to-back raycasting along a ray with sample distance Δ. Steps in orange are executed for the estimation of ERT only, steps in blue are only carried out for the actual rendering.

```
1:  function RAYCASTING_ERT/VOL(H_α/V_rgbα, D, Δ)
2:    χ_α ← 0                                          ▷ initialize opacity
3:    χ_rgb ← (0,0,0)                                  ▷ initialize color
4:    τ ← RAYID                                        ▷ seed for PRNG
5:    for all d ∈ {D_front ... D_back, Δ} do           ▷ sample ray w/ step size Δ
6:      rgb, α ← V_rgbα(R(d))                          ▷ color and opacity from mapped volume
7:      τ ← HYBRIDTAUSWORTHE(τ)                        ▷ τ from PRNG [Nguyen 2007]
8:      α ← SAMPLE(H_α(R(d)), τ)                       ▷ τ for weighted histogram sampling
9:      α ← 1 − (1 − α)^Δ                              ▷ adjust opacity contribution w.r.t. step size
10:     χ_α ← χ_α + α(1 − χ_α)                         ▷ blend opacity
11:     χ_rgb ← χ_rgb + α · rgb(1 − χ_α)               ▷ blend color
12:     if χ_α > 0.98 then                             ▷ early ray termination
13:       break
14:   return(d, χ)                                     ▷ return depth (and color) at ray termination
```

Sec. 4.2). For standard volume rendering, we then simply fetch color and opacity by accessing a scalar value from the volume, and then applying the transfer function to it (Line 6). In contrast, our approach to estimate ERT employs the opacity block histogram H_α (that is also used for depth assessment earlier). For this, we first generate a new pseudo-random number τ by employing a hybrid Tausworthe RNG (cf. [Nguyen 2007]) (Line 7). With respect the position along the ray $R(d)$, we then determine the block we are in, and consider the respective opacity histogram $H_\alpha(R(d))$. From this histogram, we draw a random sample for opacity α using τ, weighting each histogram bin with its respective size (Line 8). As volume rendering is typically memory-bound, the core idea of using H_α is to achieve a faithful estimation of ERT behavior at a drastically reduced cost, particularly for I/O.

In our implementation, we represent a 16^3 voxel block by a 16 byte histogram (one byte per bin), that can be obtained via a single fetch operations on GPUs (which is done very efficiently even across different rays due to texture caching). The random sampling of the opacity histogram H_α basically aims to statistically reproduce the actual raycasting for the full volume V_α. Apart from the significantly reduced cost for texture fetches, also the step size along the ray is chosen much more coarsely than for the volume rendering variant. For both ERT estimation and volume rendering, we adjust the opacity with respect to the employed step size Δ (Line 9). This is crucial for several reasons: First of all, as mentioned above, we typically traverse our histogram structure for ERT estimation with a much coarser step size (in ray space) than the one used for volume rendering. Therefore the adjustment is necessary to make the obtained opacities directly correspondent to each other. Secondly, as discussed in Sec. 6, step sizes may be dynamically adjusted, and the correction is required to basically yield the same result for different steps sizes (apart from undersampling effects). As a last step, we check for early ray termination by comparing the accumulated opacity χ_α against a certain threshold, and exit the raycasting loop if it is exceeded (Lines 12 & 13). Finally, we output the depth d_{back} along the ray at which it was terminated. For standard volume rendering, we naturally also output the resulting (pixel) color (Line 14).

5 Hybrid Performance Model

Our hybrid model estimates the performance of the upcoming volume raycasting, considering multiple factors including the step size Δ (Alg. 1, Line 11). It consists of two major components. First, we use machine learning on the basis of execution time measurements to obtain the average cost σ per sample during raycasting (Sec. 5.1). Then, we use σ along with the estimated depth $\tilde{D}_{\text{ERT}}$ (from Sec. 4) to predict the total cost $\tilde{t}$ of the upcoming volume rendering (Sec. 5.2).

Our hybrid model can be categorized as "semi-empirical" because we use known attributes of our raycasting algorithm as well as empirical measurements of the execution time [Hoefler et al. 2011].

5.1 Machine Learning: Prediction of Sample Cost σ

We use kernel recursive least squares (KRLS) as machine learning technique [Engel et al. 2004]. KRLS is an online regression algorithm that is kernel-based, i.e. it is able to perform non linear regression using Mercer kernels. We use the implementation provided by the Dlib machine learning library [King 2009]. KRLS can be categorized as a nonlinear adaptive filter that minimizes an error or loss function, characterizing the distance from ideal behavior. Most importantly for our needs, it provides real time capabilities. Recursive least squares (RLS) is used as the underlying regression algorithm. In its most simple form, RLS accepts training points (x_i, y_i) incrementally and, at each frame, maintains the solution to the following optimization problem:

$$\min_w \left(\sum_i F^i (y_i - x_i^T \times w)^2 \right). \tag{1}$$

In training pairs (x_i, y_i), x_i is our features vector and y_i is a target scalar value. By using kernel methods, KRLS implements a nonlinear transfer function while at the same time keeping the low computational footprint for adding training data on the fly and evaluating the model. We use radial basis functions (RBF) as kernel functions due to their flexibility. In our case, the target scalar vector is the sample cost σ that we aim to predict. Our feature vector x_i consists of the following components:

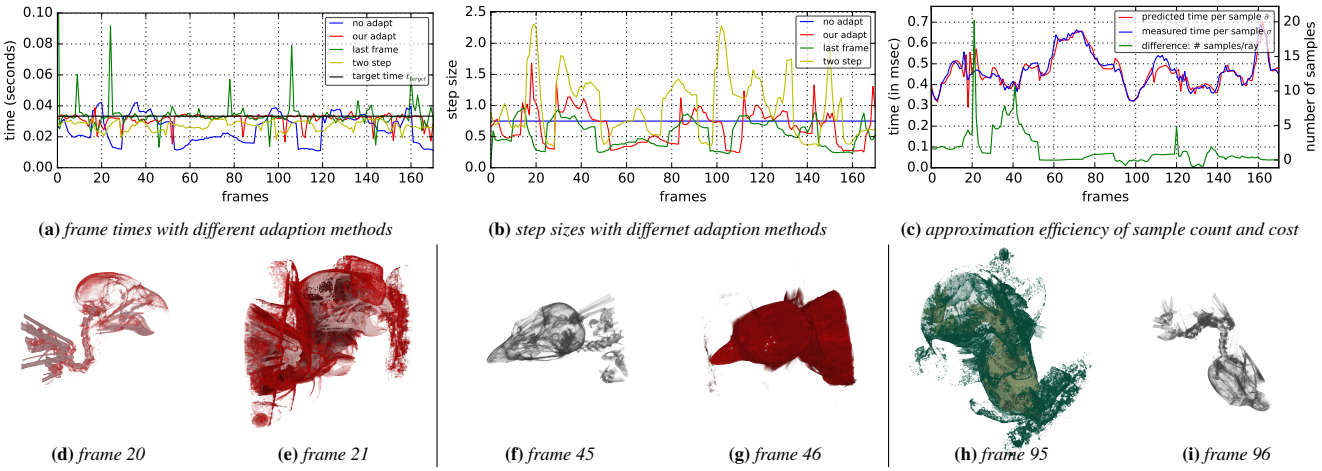
- **Viewing angles** that are directly derived from our arcball-style camera rotation.
- **Size of a splatted voxel** which is important for the performance, because potentially, it has a significant impact on the caching and varies with view distance and resolution.
- **Step size** Δ as the respective feature defined in ray space.
- **Execution time** of our ERT approximation routine.

Among other reasons, these featues are chosen to reflect the performance influencing characteristics due to different texture access patterns (e.g. [Bethel and Howison 2012]). Furthermore, all features are already available values or can be calculated with minimal computational footprint.

The Dlib implementation of KRLS provides us with the ability to adjust some parameters: The maximum number of dictionary vectors that are used to represent the regression function as well as a tolerance value. Furthermore, a γ-parameter can be chosen for the used RBF kernel functions. We determined a set of good working parameters using a two step auto tuning approach. Therefore we first sampled the parameter space in coarse steps for all possible configurations, using different data sets and sequences for testing. Afterwards, we made a second run with fine grained steps around the best result from the previous run that resulted in the following parameters: $\gamma = 0.00025$, a tolerance of 0.006 and a dictionary limit of 10 million entries.

5.2 Analytic Model: Prediction of Total Cost

The entry and exit depths of the ray are written to a 2D texture during our data collection described in Section 4. These values are used for empty space leaping and additionally give us the possibility to directly assess the average of the depth values $\bar{d}_{\text{front}}$ and $\bar{d}_{\text{back}}$ via the top level of a full mipmap stack of this texture. Using the distance of the two average values as well as the estimated cost per



(a) *frame times with different adaption methods* **(b)** *step sizes with differnet adaption methods* **(c)** *approximation efficiency of sample count and cost*

(d) *frame 20* **(e)** *frame 21* **(f)** *frame 45* **(g)** *frame 46* **(h)** *frame 95* **(i)** *frame 96*

Figure 3: *Analysis of a single sequence at the example of the Parakeet data set. Frame times of our approach (red) are shown in (a), in comparison to methods based on last frame adaption (green), two pass adaption (yellow) and no adaption (blue). The according step sizes can be found in (b). Plot (c) shows the differences between our estimation and measurements for average sample count per ray and sample cost σ. Example pairs of consecutive frame renderings from the sequence can be found in (d)-(i).*

sample σ , we now estimate the total cost $\tilde{t}$ via analytic modeling:

$$\tilde{t} = \frac{\overline{d}_{\text{back}} - \overline{d}_{\text{front}}}{\Delta} \cdot \sigma. \tag{2}$$

Simply put, we first compute the average length of all rays $\overline{l} = \overline{d}_{\text{back}} - \overline{d}_{\text{front}}$ (with d_{front} being the entry depth and d_{back} denoting its estimated termination depth in ray space). Dividing $\overline{l}$ by the step size Δ then gives the estimated number of samples, that we multiply with σ to yield out estimation of total cost $\tilde{t}$.

6 Prediction-based Tuning

Our performance prediction model provides the basis for dynamically adjusting parameters of applications with the goal to achieve high execution efficiency and/or responsiveness. We look at a single-node interactive application and define a certain target frame rate t_{target} that we aim to constantly achieve during user exploration. Other possible use cases for our approach, such as load balancing in distributed ray casting are thinkable but outside the scope of this work. As a parameter, we consider adjusting our step size Δ to achieve the target frame rate. Here, we basically follow an iterative optimization approach (Alg. 1(Lines 10–14)). In each iteration, we employ an approach based on linear extrapolation and bisection:

$$\Delta = \begin{cases} \Delta_{\text{upper}} \cdot \frac{t_{\text{target}}}{\tilde{t}_{\text{upper}}} & \text{if } \tilde{t}_{\text{upper}} < t_{\text{target}} \\ \Delta_{\text{lower}} \cdot \frac{t_{\text{target}}}{\tilde{t}_{\text{lower}}} & \text{if } \tilde{t}_{\text{lower}} > t_{\text{target}} \\ \Delta_{\text{lower}} + (\Delta_{\text{target}} - \Delta_{\text{lower}}) \frac{t_{\text{target}} - \tilde{t}_{\text{lower}}}{\tilde{t}_{\text{upper}} - \tilde{t}_{\text{lower}}} & \text{else} \end{cases} \tag{3}$$

$\tilde{t}_{\text{upper}}$ denotes the estimated timing that, so far, is the smallest one larger than t_{target} (Δ_{upper} stands for the respective step size). Conversely, $\tilde{t}_{\text{lower}}$ and Δ_{lower} denote the timing and respective step size that is the largest one computed so far that is smaller than t_{target}.

For extrapolation, the top two conditions in Eq. 3 (i.e. only smaller or only larger timings than t_{target} have been estimated so far) employ an implicit general assumption about the approximately linear impact of the step size on the performance. The third condition basically employs linear interpolation to approximate a new candidate step size Δ. To avoid an overcompensation, we set a fixed maximum adaption factor of 0.8 for reducing Δ. Although this can result in a lower quality for a few frames, we can make sure to avoid any lags that could be the result of a too optimistic step size adaption. This is not a problem for increasing the step size due to the fact that it has no negative impact on the performance. Therefore we only apply this limit for lowering the step size.

7 Results

In this section, we evaluate, compare and discuss the results achieved with our approach. Our measurements were conducted on a workstation PC running Linux with an Intel Core i7-6700 CPU, 16 GB of RAM and two graphics cards. An NVIDIA GeForce GTX 680 with 4 GB of VRAM was used as an OpenCL computing device for the actual volume ray cast with a 1024×1024 viewport. For the OpenGL preprocessing steps and computations, an NVIDIA GeForce GTX 960 graphics card with 4 GB of VRAM was employed. We used two dedicated GPUs to avoid distortions in our measurements that could be caused by other applications running on the GPU such as the X-Server. Note that we only do this for the sake of evaluation, our implementation is actually designed to run on a single GPU.

A short overview of the properties of the data sets used for measurement is presented in the second column of Table 1. Representative renderings of those can be found in the first column. Computing the histogram (BLOCKHISTOGRAM), which is done only when loading a new volume, costs 38.7 ms for a data set with 80.6 million voxels (Supernova) and up to one 1 s for the largest data set tested (Chameleon, 1.7 billion voxels). The individual steps of our pipeline for interactive exploration were designed for high efficiency to leave only a minor computational overhead: The most expensive part is the computation after a change of the transfer function. Measurements show a required maximum time of 8.952 ms, with an average of 3.573 ms. Thereby, generating the bounding geometry G takes most of the time (3.553 ms), while the opacity histograms H_α are created in about 0.020 ms on average. However, these computations have to be done only when the user changes the transfer function. The depth assessment D, that has to be calculated when the camera configuration changes, has been measured to cost a maximum of about 2.682 ms, while the ERT approximation step needed no more than 0.017 ms for completion. After training more than a thousand samples, the average cost of one prediction was about 0.001 ms while the training of one additional sample resulted in a cost of

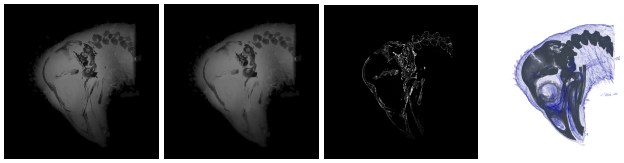
about 0.015 ms. The typical overhead time measured, accumulated to less than 3 ms for changes of the camera configuration and less than 4 ms for transfer function changes. All measurements were conducted with the Chameleon data set after a running time of more than a minute. For each of the data sets tested, a 30 s long sequence of user interactions has been recorded and rendered with different modes (see below) for comparison. Those sequences feature camera rotations around the volume in an arcball-style, zooming and changes of the transfer function (Fig. 3d-i show example renderings of different configurations during a sequence). The learning model is reset after each run. For comparison, we investigated four different modes, with the adaptive ones being measured using a target of 30 frames per second (FPS).

- **No adapt.** A conservative, fixed step size of $0.75 \times l_{\text{voxel}}$ is used (i.e., relative to voxel length l_{voxel}). The execution time of each frame is predicted with our method, but no step size adjustment is conducted.
- **Our adapt.** The methods described above are employed to predict execution times and steer the step size Δ accordingly.
- **Last frame.** The execution time of the last frame is used to adapt the step size (popular choice in load balancing).
- **Two pass.** The frame is rendered in a first pass with a quarter of the current step size. If the resulting execution time is lower than half of the target time, the step size is linearly extrapolated and the image rendered a second time with the adapted step size (this bears some similarity to progressive methods).

7.1 Analysis and Comparison of a Single Sequence

Figure 3 shows execution times of the raycasting kernel for the Parakeet data set, along with step size factors for the respective frames. Our machine learning model was not trained in advance. After a few samples, our adaption approach predicts the execution time fairly accurately, even for larger changes of the transfer function (e.g. frames 32, 57, 98). Those changes can be identified via significant drops in the execution time (respective renderings are shown in Figure 3d-i). Manipulations of the camera configuration have less abrupt impact on the performance in this example (e.g. frames 57-80). Among others, this is due to the fact that changes to the camera are comparably smooth during typical user interactions. Small transfer function changes in contrast, can lead to huge changes in the appearance, if greater parts of the volume become completely transparent or opaque for instance (c.f. Fig. 3). The discrepancies between execution and prediction mainly correlate with the approximation of the number of samples. In these cases our approach overestimates the early ray termination and predicts a longer ray sampling distance. Other differences are caused by deficits in the sample cost estimation σ.

The results of our adaption algorithm for a 30 FPS target are shown in Figure 3a. Generally, the execution time stays around the given target. However, there are also some outliers which are only in the direction of a faster execution time. Those are caused by our conservative approach for shortening the step size Δ (cf. Sec. 6). The other spikes towards a faster execution time are mostly caused by a underestimation of ERT (cf. Sec. 7.3). This is caused by the fact that we do not change our ERT approximation step when adjusting the step size. Most importantly there are no big outliers indicating a greatly incresed execution time relative to the target. In comparison, an adaption using the last frame time, produces huge spikes that indicate a much longer execution time for some frames. Those are the critical frames that cause a visible lag during interactive exploration. The obvious reason for these outliers is that the technique cannot predict any changes by design and solely uses recent history to adapt respectively. The two pass approach, that is also depicted in Figure 3a, has the advantage that the frame target is never exceeded.



(a) *real* **(b)** *estimation* **(c)** *difference* $\times 4$ **(d)** *rendering*

Figure 4: *Comparison of the ray termination depths (a), and our estimation (b) (both mapped to gray values), for the Hoatzin data set (courtesy UTCT). (c) is the difference between (a) and (b) (scaled by factor* 4*). (d) shows a rendering, using the same transfer function.*

However, as can be seen in Figure 3b, the average step size Δ is significantly higher than the one of the other two techniques, resulting in a lower overall quality. This is obviously caused by the pre-render pass that is needed. The available rendering time is generally not exploited to the highest efficiency as earlier results need to be discarded in the second pass which in turn has a shortened possible rendering time. The difference between the real ray termination depth values and our estimation is illustrated in Figure 4. As can be seen, our approximation is very similar to the real ray termination depth. However, there are some differences, mainly at the borders, that are caused by the lower resolution that comes from our block partitioning used in the estimation process as well as the stochastic methods (cf. Sec. 4.3). For the test sequence, these discrepancies are also shown in Figure 3c in form of a difference plot (green). The same plot also shows the measured cost per sample (blue) in comparison to our estimation σ (red). It illustrates the efficiency of the adaptive filter with our configuration (see Sec. 5.1).

7.2 Measurement Series

In the following, we discuss our prediction results for multiple different data sets (Table 1). Typical interaction sequences have been recorded for each data set to capture different scenarios. They last 30 s each and the same sequence was used to test the different modes. As a basic indicator of rendering quality, the average step size factor Δ is used, that is relative to the respective voxel size of the measured data set. In general, a smaller value for Δ indicates a higher rendering quality. The root-mean-square error (*RMSE*) is used as a measure for the differences between predictions ($\hat{y}_t$) and measurements (y_t) across a sequence run of n frames:

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

For better insight, two additional *RMSE*s of subsets are calculated: one that only takes into account execution times that lie above the frame target ("*up*") and one calculated from the remaining values ("*low*"). The most important goal for our technique is a fluid user interaction without lags, so we consider $RMSE_{up}$ to be more significant. The same holds true for the maximum errors listed in the Table 1. One is showing the maximum deviation that is higher than the target of 30 FPS ($error_{up}$), while the other shows the maximum that is lower ($error_{low}$). Both relative as well as absolute errors are listed, Equation 5 shows the calculation formulas, with t being the frame where the maximum error occurs.

$$error_{abs} = |\max(y_t - t_{target})| \qquad error_{rel} = \frac{error_{abs}}{y_t}. \tag{5}$$

Our approach keeps the frame times around a target value during the sequences with only small deviations. In addition, as can be seen in Table 1, our secondary goal to maximize the sampling rate

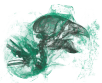
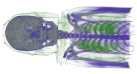
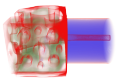
representative rendering	**data set** resolution / bps / courtesy	mode	avg. step size Δ	$RMSE_{up}$	$RMSE_{low}$	$RMSE$	absolute $error_{up}$	absolute $error_{low}$	relative $error_{up}$	relative $error_{low}$
	**Chameleon** 1024 × 1024 × 1024 16 UTCT	last frame no adapt our adapt two pass	0.38377 0.75000 0.42096 0.64208	0.02077 0.00583 0.00422 0.00975	0.00402 0.01909 0.00685 0.00512	0.01356 0.00276 0.00567 0.00515	0.18874 0.01187 0.01783 0.00975	0.02607 0.02762 0.02514 0.01752	0.84990 0.26267 0.34854 0.22624	3.58959 4.83096 3.07031 1.10808
	**Foraminifera** 1024 × 1024 × 219 16 UTCT	last frame no adapt our adapt two pass	0.41585 0.75000 0.47134 0.54485	0.00400 0.00099 0.00157 0.00320	0.00208 0.01660 0.00472 0.00194	0.00290 0.00268 0.00401 0.00195	0.02282 0.00153 0.00379 0.00320	0.01072 0.02556 0.01521 0.00388	0.40640 0.04384 0.10220 0.08759	0.47395 3.28719 0.83918 0.13164
	**Parakeet** 1024 × 1024 × 340 16 UTCT	last frame no adapt our adapt two pass	0.52785 0.75000 0.59595 0.96444	0.01014 0.00565 0.00197 0.00000	0.00443 0.01529 0.00485 0.00675	0.00742 0.00231 0.00422 0.00675	0.05941 0.01246 0.00709 0.00083	0.02042 0.02268 0.01782 0.01512	0.64059 0.27212 0.17544 0.02565	1.58062 2.12939 1.14804 0.82977
	**Supernova** 432 × 432 × 432 8 John M. Blondin	last frame no adapt our adapt two pass	0.33685 0.75000 0.36342 0.52813	0.00574 0.00000 0.00204 0.00000	0.00253 0.01883 0.00384 0.00469	0.00398 0.00128 0.00352 0.00469	0.04413 0.00371 0.00553 0.00007	0.02113 0.02709 0.01994 0.01373	0.56971 0.12538 0.14230 0.00200	1.73206 4.34284 1.48944 0.70007
	**VisFemale** 512 × 512 × 512 8 U.S. Nat. Lib. of Medicine	last frame no adapt our adapt two pass	0.55918 0.75000 0.60369 1.02540	0.02672 0.01638 0.00265 0.01651	0.00397 0.01956 0.00429 0.00603	0.01562 0.00218 0.00378 0.00614	0.21792 0.02708 0.01010 0.01651	0.02785 0.02662 0.02755 0.01634	0.86733 0.44828 0.23260 0.33130	5.08358 3.96197 4.76140 0.96160
	**Zeiss** 680 × 680 × 680 8 Daimler AG	last frame no adapt our adapt two pass	0.47995 0.75000 0.49640 0.83315	0.01085 0.01440 0.00419 0.00137	0.00534 0.02172 0.00556 0.00521	0.00868 0.00197 0.00523 0.00520	0.10152 0.02736 0.02061 0.00137	0.02707 0.03145 0.02569 0.01321	0.75282 0.45081 0.38209 0.03936	4.31777 16.67390 3.35975 0.65619

Table 1: *Results of sequence measurements for different volumes. A smaller average step size factor indicates better quality, while a lower RMSE indicated a better performance (closer to the target). Maximum errors and relative maximum errors are listed as well. The "up"-tag indicates that only measured values above the target of 30 FPS are taken into account, the "low"-tag that only those below are considered.*

by adapting it according to the predicted frame times could be achieved as well. The $RMSE_{up}$ of the values above the target frame time is considerably lower for our technique than that of the last frame approach in most cases (e.g., Parakeet, Chameleon). At the same time, the average step size factor is generally not much higher. Employing the two-pass-technique, usually results in lower *RMSE*s but also in noticeably greater step size factors, compared to our method. One of our main goals was to avoid lags that are caused by greatly differing configurations, as a result of user interactions. The maximum error values indicate that our method can handle such cases far better than a last-frame-approach (e.g. Supernova).

7.3 Discussion of Limitations and Extensibility

While we demonstrated that our technique works well for the tested data sets and we can stay within a defined frame time limit while at the same time keeping a high sampling rate, there are also a few limitations. In some cases, the underestimation of the impact of ERT, especially with higher step sizes, causes the *RMSE* for the execution times below the target to be higher than that of the last-frame-approach. However, faster frames naturally do not cause any lags in interactivity. Additionally, there are some cases where our method predicts inaccurate execution times per sample. This can occur when our machine learning model has not yet learned a case from which it can derive a proper approximation of the sample cost. Further refinement of our model or the introduction of uncertainty handling could reduce those cases. Possible uncertainty handling could include a test, whether a prediction is made with a configuration that is distant from all samples learned so far. Appropriate reactions could then be conducted, e.g. taking a large step size Δ that basically guarantees a quick response. Due to the nature of the used KRLS algorithm, the learned weights cannot be transferred between data sets. As can be seen however, the adaptive filter yields decent results after training only a small set of samples. A different adaptive machine learning algorithm could potentially be better suited for transferring learned decision functions between different data sets.

8 Conclusion

We presented an integrated approach for real-time performance prediction and tuning of volume raycasting. For interactive exploration, it can significantly reduce unpleasant effects such as jerky motions and abruptly reduced responsiveness. To overcome those effects, we proposed techniques to explicitly assess acceleration information for the rendering and thereby calculate performance-relevant data. This also includes a new technique to estimate the impact of early ray termination. Using this data, we introduced a hybrid prediction model that can give accurate performance predictions in real-time. It is a combination of an analytical model and machine learning that, among others, takes the sampling density along each ray into account. This sampling density is adjusted on-the-fly by our automatic tuning technique to reliably meet performance requirements. By applying this, we were able to keep frame execution times at a defined target for various volume data sets and sequences of user interactions, even in the case of large sudden changes to the transfer function or the camera configuration. At the same time, we could keep the rendering quality at a high level by adjusting the step size according to the predicted performance requirements.

For future work, we aim to further improve our prediction model by integrating uncertainty handling to recognize cases in which no adequate prediction is possible. Also, the consideration of additional acceleration techniques (e.g., using octrees), data representations, as well as different illumination techniques could further improve the generality of our approach. Besides that, we plan to tune not only in object space (via the step size along rays), but also in image space (via adaptive sampling). Finally, we want to look into further use cases: for optimized scheduling in in-situ scenarios as well as parallel rendering with dynamically resized data.

Acknowledgments

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

References

AUTODESK, INC., 2014. Optimize hardware rendering for frame rate or quality.

BAGHSORKHI, S. S., DELAHAYE, M., PATEL, S. J., GROPP, W. D., AND HWU, W.-M. W. 2010. An adaptive performance modeling tool for gpu architectures. In *ACM Sigplan Notices*, vol. 45, ACM, 105–114.

BARNES, B. J., ROUNTREE, B., LOWENTHAL, D. K., REEVES, J., DE SUPINSKI, B., AND SCHULZ, M. 2008. A regression-based approach to scalability prediction. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, ACM, New York, NY, USA, 368–377.

BETHEL, E. W., AND HOWISON, M. 2012. Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning. *International Journal on High Performance Computing Appllications 26*, 4 (Nov.), 399–412.

BEYER, J., HADWIGER, M., AND PFISTER, H. 2015. State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum* (2015).

BOWMAN, I., SHALF, J., MA, K.-L., AND BETHEL, W. 2004. Performance modeling for 3d visualization in a heterogeneous computing environment. *Lawrence Berkeley National Laboratory*.

EILEMANN, S., AND PAJAROLA, R. 2007. 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, 29–36.

ENGEL, Y., MANNOR, S., AND MEIR, R. 2004. The kernel recursive least-squares algorithm. *IEEE Transactions on signal processing 52*, 8, 2275–2285.

FREY, S., SADLO, F., MA, K.-L., AND ERTL, T. 2014. Interactive progressive visualization with space-time error control. *IEEE Transactions on Visualization and Computer Graphics*. DOI:10.1109/TVCG.2014.2346319.

HOEFLER, T., GROPP, W., KRAMER, W., AND SNIR, M. 2011. Performance modeling for systematic performance tuning. In *State of the Practice Reports*, ACM, New York, NY, USA, SC '11, 6:1–6:12.

HONG, S., AND KIM, H. 2009. An analytical model for a gpu architecture with memory-level and thread-level parallelism awareness. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ACM, NY, USA, ISCA '09, 152–163.

KING, D. E. 2009. Dlib-ml: A machine learning toolkit. *Journal of Machine Learning Research 10*, 1755–1758.

KOTHAPALLI, K., MUKHERJEE, R., REHMAN, M. S., PATIDAR, S., NARAYANAN, P., AND SRINATHAN, K. 2009. A performance prediction model for the cuda gpgpu platform. In *High Performance Computing (HiPC), 2009 International Conference on*, IEEE, 463–472.

KRATZ, A., REININGHAUS, J., HADWIGER, M., AND HOTZ, I. 2011. Adaptive screen-space sampling for volume ray-casting. Tech. Rep. 11-04, Zuse Institute Berlin, Takustr.7, 14195 Berlin.

LAHA, B., SENSHARMA, K., SCHIFFBAUER, J. D., AND BOWMAN, D. A. 2012. Effects of immersion on visual analysis of volume data. *IEEE Transactions on Visualization and Computer Graphics 18*, 4 (April), 597–606.

LEE, B. C., BROOKS, D. M., DE SUPINSKI, B. R., SCHULZ, M., SINGH, K., AND MCKEE, S. A. 2007. 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, 249–258.

NGUYEN, H. 2007. *Gpu Gems 3*, first ed. Addison-Wesley Professional.

QU, H., WAN, M., QIN, J., AND KAUFMAN, A. 2000. Image based rendering with stable frame rates. In *Proceedings of the IEEE Conference on Visualization '00*, 251–258.

RIZZI, S., HERELD, M., INSLEY, J. A., PAPKA, M. E., URAM, T. D., AND VISHWANATH, V. 2014. Performance modeling of vl3 volume rendering on gpu-based clusters. In *Proceedings of the Eurographics Symposium on Parallel Graphics and Visualization*, 65–72.

SALAMA, C., HADWIGER, M., ROPINSKI, T., AND LJUNG, P. 2009. Advanced Illumination Techniques for GPU Volume Raycasting. In *ACM SIGGRAPH Courses Program*.

SCHARSACH, H., HADWIGER, M., NEUBAUER, A., WOLFSBERGER, S., AND BÜHLER, K. 2006. Perspective isosurface and direct volume rendering for virtual endoscopy applications. In *Proceedings of the Eighth Joint Eurographics/IEEE VGTC conference on Visualization*, Eurographics Association, 315–322.

SHEN, H.-W., AND JOHNSON, C. R. 1994. Differential volume rendering: a fast volume visualization technique for flow animation. In *Proceedings of the conference on Visualization '94*, 180–187.

SODHI, S., SUBHLOK, J., AND XU, Q. 2008. Performance prediction with skeletons. *Cluster Computing 11*, 2, 151–165.

TACK, N., MORÁN, F., LAFRUIT, G., AND LAUWEREINS, R. 2004. 3d graphics rendering time modeling and control for mobile terminals. In *Proceedings of the ninth international conference on 3D Web technology*, ACM, 109–117.

TIKIR, M. M., CARRINGTON, L., STROHMAIER, E., AND SNAVELY, A. 2007. 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, 47:1–47:12.

WIMMER, M., AND WONKA, P. 2003. Rendering time estimation for real-time rendering. In *Proceedings of the 14th Eurographics workshop on Rendering*, Eurographics Association, 118–129.

WONG, G., AND WANG, J. 2014. *Real-time rendering: Computer graphics with control engineering*. CRC Press, Boca Raton, FL.

WOOLLEY, C., LUEBKE, D., WATSON, B., AND DAYAL, A. 2003. Interruptible rendering. In *Proceedings of the 2003 Symposium on Interactive 3D Graphics*, 143–151.

YU, H., WANG, C., AND MA, K.-L. 2008. Massively parallel volume rendering using 2-3 swap image compositing. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, IEEE Press, Piscataway, NJ, USA, 48:1–48:11.