Published in final edited form as:

IEEE Trans Vis Comput Graph. 2014 May ; 20(5): 767–780. doi:10.1109/TVCG.2014.2303092.

Hybrid Rendering with Scheduling under Uncertainty

Georg Tamm and

DFKI, Saarbrücken, Germany

Jens Krüger [Member IEEE]

Intel VCI, SCI, and the University Duisburg-Essen, Essen, Germany

Georg Tamm: georg.tamm@dfki.de; Jens Krüger: jens.krueger@dfki.de

Abstract

As scientific data of increasing size is generated by today's simulations and measurements, utilizing dedicated server resources to process the visualization pipeline becomes necessary. In a purely server-based approach, requirements on the client-side are minimal as the client only displays results received from the server. However, the client may have a considerable amount of hardware available, which is left idle. Further, the visualization is put at the whim of possibly unreliable server and network conditions. Server load, bandwidth and latency may substantially affect the response time on the client. In this paper, we describe a hybrid method, where visualization workload is assigned to server and client. A capable client can produce images independently. The goal is to determine a workload schedule that enables a synergy between the two sides to provide rendering results to the user as fast as possible. The schedule is determined based on processing and transfer timings obtained at runtime. Our probabilistic scheduler adapts to changing conditions by shifting workload between server and client, and accounts for the performance variability in the dynamic system.

Index Terms

Remote/hybrid rendering; client/server systems; uncertainty; probabilistic scheduling

1 Introduction

Today, simulations and measurements regularly generate large scientific data sets. Often these data sets can only be understood by means of visual analysis. However, to interactively visualize high-resolution data sets, significant computing and storage resources are required. These resources may not be available on all devices where the data shall be displayed. Mobile devices may especially lack the capabilities to *fully* store and process large or even medium-sized data sets. Another more extreme case is a future exascale scenario [1], where simulations run on highly parallel, dedicated supercomputing architectures. As it is predicted processing power will increase more rapidly than storage capacity and I/O bandwidth in such systems [2], it may not even be feasible to permanently store exascale simulation results. Instead, data sets may be generated and visualized in-situ.

A solution to provide visualization to devices with limited capability is to outsource processing. A render server, or cluster of servers, processes part of or the whole

visualization pipeline, and then transmits the results to the client for further processing and/or display. Requirements on the client are reduced, and minimal if the client is used for display only. Bethel et al. [5] and Luke and Hansen [23] classify how the visualization pipeline can be distributed across server and client workstations. However, network bandwidth, latency and reliability become issues. These considerations are especially relevant for best-effort networks like the Internet or wireless connections. Further, scalability on the server becomes an issue if multiple clients request rendering tasks simultaneously. The user experience suffers if an overloaded server can not complete requests in a reasonable time. To increase server scalability, considerable investments in hardware may be necessary.

In this paper, a client is connected to a single server. We refer to remote rendering when the client is used for display only.

While the server component is necessary to provide the resources for large-scale visualization, a purely server-based approach may leave a considerable amount of client hardware idle. Nowadays, even thin mobile devices may have capabilities to support interactive visualization to a certain degree; and using them for such tasks becomes increasingly popular [8].

Therefore, hybrid rendering techniques utilize both server- and client-side resources. When we assume the data is initially stored on the server, the three core aspects are: Splitting the visualization process into units of workload, assigning workload to client and server, and transferring the data required for client-side processing. Performing work on the client reduces server load. Network load may sometimes be reduced compared to remote rendering (e.g., there may be no need to constantly transfer image data; see Section 2 for references). Also, server and client working together can result in faster image generation. The client hardware and increased scalability is available without investment from the server provider.

We present a hybrid rendering method that adaptively adjusts what rendering workload needs to be done and where it is done based on server, client and network conditions. Workload is identified in terms of quality levels (QL) of a data set. The view is progressively refined by displaying QLs of increasing number. Any renderer adhering to the concept can be plugged in. QLs are scheduled for rendering on server and client. The goal is to provide the client with the next QL as soon as possible. Depending on the capabilities of the client a subset of the QLs is supported.

Our approach uses a probabilistic scheduling model. Probability distributions for processing and transfer timings are acquired and updated at runtime for each QL to determine what QLs are to be rendered on each side. The method allows us to account for the uncertain conditions affecting the performance.

A client with arbitrary capabilities may connect via a network link with arbitrary characteristics. Load on server and network may vary depending on how many clients are active and what outside traffic is on the link. We make no assumptions about these conditions, which are subject to change. Other factors, like background tasks performed by

the operating system, hardware characteristics or render system events (e.g., allocating memory) may result in fluctuating timing observations.

To account for the dynamic variables, timing information is updated and compared in terms of a probability distribution to adapt the schedule.

2 Related Work

2.1 Hybrid Rendering

We divide hybrid rendering methods into three categories: First, every image is cooperatively rendered by server and client. Second, a number of images are independently rendered by the client after an initial input from the server. From time to time, additional information and/or tasks have to be provided by the server to keep up rendering on the client. Third, each image is independently produced by server or client.

Falling into the first category, Aranha et al. [3] distribute ray tracing workload. A cost function is used to decide on the number of pixels to be rendered on each side.

Several techniques have been developed for volume rendering. To progressively render unstructured, tetrahedral grids, Callahan et al. [9] utilize the client as the rendering unit while the server stores the data and preprocesses the geometry. Prohaska et al. [25] use a hierarchical volume renderer for CT-scan exploration on the client. Data blocks are accessed remotely to progressively refine the view. The system supports a user-defined region-of-interest (ROI) for which the highest resolution is chosen.

A server-centered approach is described by Diepstraten et al. [10] for line rendering on mobile devices. Lines are extracted and projected to image space by the server. A package of 2D lines is then rendered by the client. Okamoto et al. [24] propose a system, where the server stores geometric data and a repository of pre-rendered images. When a client requests a view, a selection of images most closely matching the view is sent along with a coarse version of the mesh. The view is then reconstructed on the client.

Falling into the second category, Engel et al. [14] describe a method to distribute isosurface reconstruction. The approach employs a hierarchical level-of-detail (LOD) concept. The user can define a ROI, which will be reconstructed with highest quality. After reconstruction has been completed on the client, local interaction is possible until the isosurface changes. Similar approaches are outlined by Engel et al. [13] for the visualization of multidimensional chemical data sets.

Li et al. [21] render an image on the server, and extract a mesh from the depth buffer. The mesh is textured with the color buffer and sent to the client for rendering. Should the error due to view changes get too high, a new mesh is requested from the server. A similar approach is outlined by Luke and Hansen [23]. Visapult [6] produces images using a parallel volume renderer on the server, and then interactively constructs new views from the images on the client. Client-side rendering is decoupled from receiving image updates.

The techniques presented above require an application-specific algorithm to distribute the rendering workload. Further, a persistent connection to the server is required. High latency reduces the interactivity. Connection loss terminates the visualization on the client immediately or when new input from the server is required.

Our approach falls into the third category: generic hybrid rendering. A medical application is given by Engel et al. [12]. Two different volume renderers are used. Low-quality rendering is performed by the client. Higher quality can be requested from the server. Dyken et al. [11] describe a web application, which supports a browser-based client. An illustrative, coarse representation of a geometric data set is drawn interactively on the client with WebGL. The detailed original model is processed on the server, and rendering results are transferred to the client as images. The generic method allows us to maintain interactivity on a capable client should the connection suffer from latency or the server be occupied or unavailable. In addition, it allows us to support applications independent of the rendering algorithm.

2.2 Scheduling under Uncertainty

In various scheduling problems, uncertainty is a significant factor. The time required to complete a task may be difficult to estimate before actual observations. Resources may be unreliable, occupied or even become unavailable. Probabilistic approaches have been employed to deal with such scenarios. Instead of absolute values, probability distributions represent the state of the system. Related methods have been used in the areas of project management [17], maintenance/production [28], and process planning [19].

Ierapetritou and Li [22] [19] distinguish two approaches, and give several references for each. First, in a preventive scheduling system, the behavior of uncertain factors can be modeled based on historical data and statistics gathered previously. Thus, the characteristics of the probability density function (PDF) are either known or can be estimated to determine the schedule in a pre-process. Examples are described by Janak et al. [20] and by Balasubramanian and Grossmann [4].

In contrast, in a reactive scheduling system, not enough information about the uncertain factors is available before the tasks are performed. Therefore, the schedule is constantly adapted to respond to changing conditions.

Our method can be classified as reactive. The timings for QL processing and transfer are subject to fluctuation due to several dynamic factors (see Section 4.2), which can not be known a priori for any client-server connection. Therefore, timing distributions are obtained at runtime to determine the schedule. However, our approach adds additional complexity in that we consider not scheduling certain QLs to keep load from the system, and allow the rendering of QLs to be aborted to maintain interactivity. Consequently, timings are not guaranteed to be obtained in frequent intervals.

Our approach is also related to resource-aware scheduling on an abstract level. Resource-aware scheduling partitions data-parallel problems into pieces of load, which are then to be scheduled for distributed processing on possibly heterogeneous computing resources.

Similar to our hybrid rendering, knowledge about the availability, capability and performance of the resources must be considered to balance the load.

Viswanathan et al. [29] implement a system based on the divisible load theory [7]. Load is scheduled in a cluster, whose nodes are connected in a LAN. Source nodes generate load at runtime, which is to be processed by sink nodes. A control node coordinates the distribution of load based on the sinks' estimated memory and processing capabilities, and the sources' load size and deadline requirements, which are not real-time. The goal is a maximum utilization and throughput, and a maximum acceptance rate of new load. The algorithm runs iteratively until all load has been admitted and processed. Unlike our system, network overhead is regarded negligible, and a substantial amount of control information is passed between the nodes.

Teresco et al. [26] schedule load for large simulations in a heterogeneous computing environment. This includes possibly non-dedicated nodes processing additional tasks apart from the target application, which is a scenario similar to a loaded server in our case, which processes rendering tasks for multiple clients independently. The problem domain is discretized for cooperative processing using a mesh partitioning algorithm. The system obtains performance characteristics of computing and network components to guide the partitioning. It supports runtime monitoring to allow adapting the partitioning. Developers have to write their applications within a specific architecture to plug them into the scheduling system.

3 Scheduling Fundamentals

First, we describe the fundamentals required for the hybrid rendering method. The goal of the scheduler is to provide the client with a new QL for display as fast as possible. We therefore start with the definition of quality levels. Next, we describe the requirements on the render system, which includes addressing limitations of the QL concept. Last, we outline the transfer of data required to render QLs from server to client to enable client-side rendering in the first place.

3.1 Quality Levels

Our method builds on the concept of *quality levels*. A QL is a representation of a data set to be displayed, which can be independently rendered on either the client, the server, or both, to produce an image. We require the QLs to be totally ordered, with detail increasing with the level number. QLs are requested consecutively when rendering a frame to progressively refine the view. We often refer to either server or client instance of a QL when using the term QL.

The classification of a data set into a set of totally ordered QLs is provided by the underlying render system, and thus ultimately up to the application developer. The format of the QL data (including possible compression), and the rendering algorithm to produce images from the data, must not be known to our hybrid scheduling layer on top of the render system.

We assume that all QLs are available to the server. QLs of higher resolution may not be supported by resource limited clients.

3.2 Underlying Render System

Our approach supports any renderer that can map its data to QLs, e.g., using a multi-resolution hierarchy, different sampling rates or some other unforeseen representation. It is therefore applicable to any visualization system that already uses a multi-resolution data set representation. We acknowledge the mapping to QLs may not be straight forward. This especially applies to view-dependent rendering systems, where the LOD may vary over regions of the data set. Also, approaches offering a continuous refinement can not be directly mapped. It is required to discretize the continuous representation to be able to distribute QLs for scheduling. Here it is feasible to create a QL hierarchy not too fine considering images for QLs rendered server-side must be sent over the network.

We expect the renderer to be interactive. It should provide at least one QL per data set that can be completed at interactive frame rates. It should also be interruptible to maintain responsiveness.

We allow the renderer to reject QLs above a certain level. For example, the screen space size of the visualization may be used to limit the rendering of higher QLs to avoid aliasing of small details, and to save resources.

We have integrated two different rendering systems. The first is the *Tuvok* volume rendering system [16]. *Tuvok* provides a hierarchical renderer, which divides a data set into LODs. These LODs can be independently stored and rendered. They can be directly mapped to our QLs (Fig. 1). Moreover, we extend this QL mapping to underline the flexibility of the concept. An LOD can be divided into multiple QLs to enable a more fine-grained view refinement (Section 5.3). The second system uses a streaming geometry renderer, based on progressive meshes [18].

3.3 Interleaved QL Data Transfer

We make no assumptions as to the availability of QL data on the client. Initially, all rendering is performed by the server. The server encodes image data as JPEG and sends to the client via a TCP connection.

To enable the ability to switch where QLs are rendered, the client needs to obtain the QL data for local rendering. This process is outlined in Fig. 2. First, the system determines the QLs the client supports. This is performed in a handshake phase. The system queries the server-side renderer for QL specifications of a data set. This meta-data contains application-specific information, such as the QL data size and required renderer features (like 3D textures). The server sends the meta-data to the client. The client-side render system then determines for each QL if it can be supported.

The client may not support a QL due to limited memory and processing power. Required renderer features may not be present. Disapproving a QL can be done to save memory and network bandwidth. It may not be feasible to transfer large QL data (e.g., to a mobile device

over a wireless link). Further, a small display resolution could make it infeasible to even consider a high-resolution QL on either side. Finally, the QL data may be confidential.

After the handshake, the server sends the supported QL data to the client using the *Deflate* compression algorithm. Time division multiplexing is used to interleave image data with QL data. There are two suitable scenarios to insert data chunks. First, if user interaction stops, the server renders QLs of increasing number to progressively refine the view. Rendering a high QL can take time, during which the client waits for image data. Second, after the client received the final QL, both sides are idle waiting for a new interaction event.

4 Scheduling Quality Levels

Once a QL is available at the client, it is included in the scheduling process. Scheduling is performed only on the client. For simplicity of the explanation, we assume in this section that the client has all supported QLs available. Fig. 3 gives an architecture overview to set the stage for the following detailed description: First, we describe the timing distributions the client holds to base the scheduling decision on, and the fluctuation expected to occur in the timings (Sections 4.1 and 4.2). Next, we present the details of the scheduling algorithm, and how it connects to the rendering process (Section 4.3). Finally, we describe how the distributions are initialized and updated with timings at runtime (Sections 4.4 and 4.5), and how the timings are obtained (Sections 4.6 and 4.7).

4.1 Timing Distributions

For each QL on server and client, the scheduler maintains a continuous processing and transfer time *normal distribution* (ND) with millisecond accuracy. A discussion on why we choose the ND as the distribution type follows in Section 6. The scheduling algorithm uses the NDs to decide which QLs are to be rendered on which side (Section 4.3). The NDs are built by adding timings (Section 4.5). Processing time includes the render time, and in case of the server, the image encoding time. Transfer time only applies to server QLs. It includes the time to request a QL for rendering, and to send the rendered image to the client in return.

In this paper, we simply use the term random sampling to describe generating random variates distributed according to a ND. We simply refer to a variate as a sample. The system enforces a lower bound for time NDs by falling back to the mean should a sample ≤ 0 occur.

The scheduler bases its decisions on probability distributions due to the uncertain conditions that affect our timings.

4.2 Timing Fluctuation

Fluctuation in the timings can occur due to several uncertain conditions, which we classify in the following:

First, for server QLs, processing timings are affected by server load, which is arbitrary. Transfer timings are affected by network load. Since we make no assumptions about the link

between client and server, bandwidth, latency and outside traffic are arbitrary. Transfer and processing timings move along with changes in these conditions.

Second, processing timings on both server and client are affected by render system parameters. Most importantly, there are view changes, but also other parameters, like the display resolution, or a volume rendering transfer function.

Third, processing time fluctuation can occur due to background factors such as operating system and hardware characteristics, even when server load and render system parameters are stable. Task scheduling and memory performance variation are persistent examples. There may also be temporary factors like a copy or update procedure or a throttling of the GPU performance due to overheat. The range of background factors can not be modeled by our generic method. Server and client timings are affected independently.

Finally, fluctuation may occur due to outliers. An outlier is a timing, which is not representative for subsequent timings under the same conditions. Outliers are difficult to predict, but we can make assumptions about when they are likely to occur for the processing time. Such outliers can be caused by render system specific initialization effects, where resources need to be allocated. Also, hardware-specific warm-up effects are possible, especially for a GPU-heavy renderer (e.g., setting up GPU state).

To mitigate the effect of these outliers our approach utilizes a weighting system when adding timings to a ND (Section 4.5). In Section 4.4, we describe parameters which allow us to account for initialization and warm-up related outliers.

4.3 Schedule Algorithm and Rendering Process

Here, we present our probabilistic scheduling algorithm, which allows the scheduler to adapt to changing conditions. We assume there are a number of QLs on server and client, and the corresponding NDs are available. In the following sections, we describe how the NDs and timings are obtained at runtime.

Given a frame to render, the goal of the scheduling algorithm is: The view should be updated with a newly completed QL as fast as possible on the way to reach the highest QL. Completing a QL means its rendering result is available at the client for display.

We describe two versions of the algorithm. The pre-sampling strategy reflects the variance in the NDs perfectly, but undesirable slow downs can occur as a side effect (Section 4.3.1). Therefore, we provide the alternative distribution-comparison strategy to mitigate this effect (Section 4.3.2).

4.3.1 Pre-Sampling Strategy—To set the stage for the following detailed algorithm description, we give a summary here. The scheduler executes once every frame, and determines the list of QLs which are to be requested for rendering in the frame (as illustrated in Fig. 3). It uses the timing information contained in the NDs for its decision. This information tells the scheduler when a QL is expected to complete. The scheduler takes into account that client and server can operate in parallel, but each side individually works sequentially. Thus, scheduling a QL on one side affects the expected completion time of

other QLs on that side. The goal is to minimize the time until a new QL is displayed. The scheduler will thus always choose the QL which is expected to complete earliest next. QLs lower or equal to this QL are skipped, as their rendering and network transfer is expensive, and may delay the completion of subsequent QLs. Though, no timings can be obtained for skipped QLs.

Let list A be a list containing all QLs and their processing and transfer time NDs. First, the scheduler takes a random sample from the processing and the transfer time ND for each QL in A . The accumulated sample time is defined as the sum of processing and transfer time sample. The scheduler stores the processing and accumulated sample of each QL in a new list B . B is sorted by the accumulated time in ascending order. Should two QLs match in number and time, the client QL is preferred to keep load off server and network.

The main execution loop of the algorithm now processes B . No further samples are generated from the NDs within the loop, which is why the strategy proceeds deterministically from here on, and is called pre-sampling. The first QL q in B is added to a list C . C is the output of the algorithm, and QLs placed in it are to be rendered in the frame. QLs smaller or equal q are then removed from B . If B is not empty, the processing time sample of q is added to the accumulated sample of every QL on the same render side as q . The next QL on that side can not be rendered before q has been finished. The transfer time does not delay the rendering of the next QL, and is thus not added. B is sorted by accumulated time again and iteration proceeds. If B is empty, the algorithm concludes.

The first QL in C is the interactive QL, as it is expected to complete first and thus should provide interactive performance. The remaining QLs in C are the non-interactive QLs. The scheduler finally splits C into client and server QLs, which the system then requests for rendering. Server and client can process their list of QLs independently, and thus render in parallel (illustrated in Fig. 3).

Our approach uses a small time interval between completing the interactive QL and requesting the non-interactive QLs. This request timer prevents requesting noninteractive QLs prematurely, which would result in permanent rendering aborts during user interaction phases. The overhead can be noticeable, which is dependent on how quickly the renderer can exit after an abort command. In contrast, if a new interaction event arrives within the interval, the next frame can be immediately started.

Each time the algorithm is run, different random samples could be chosen to create list B , causing a different schedule outcome in list C . The order of QLs in B is uncertain, and reflects the variance in the NDs. The PDF of a ND approaches zero, but never reaches it. A sample can therefore take on any value, and any order of QLs is possible. The probability of a specific ordering is arbitrary, and may only exist theoretically; in practice, zero probabilities are possible due to limited computer precision.

The probabilistic approach allows the system to adapt to the uncertain and changing conditions. Every QL will eventually get scheduled for rendering. Gathering timings for completed QLs allows the system to notice change. However, the adaption process is not guaranteed to be immediate. First, how fast a ND shifts in response to new conditions is

dependent on our weighting system when adding timings (Section 4.5). Second, the scheduling frequency for a QL may be irregular. The probability of a QL to be scheduled for rendering in a frame is the sum of probabilities of the schedule outcomes containing the QL, and may be very small. If the system rarely schedules a QL (e.g., on an occupied server), it can be slow to notice change in the QL's performance behavior.

The pre-sampling strategy produces a schedule that takes into account the variance in the observed timings. However, there is an undesirable side effect as any QL is potentially chosen as the interactive QL. If a QL is chosen, which then does not fulfill the interactivity requirement, a slow down may be noticeable. To maintain responsiveness, the system allows rendering the interactive QL to be aborted if it does not complete within the requirement. This is optional, since it causes another problem. If no QL can meet the target FPS reliably, the system suffers from constant aborting and gaps in updating the display during interaction.

4.3.2 Distribution-Comparison Strategy—To mitigate the side effect of the pre-sampling strategy, the system provides an alternative strategy: distribution-comparison. The basic structure of the algorithm stays the same. For each QL, an accumulated time ND, which is defined as the sum of the independent processing and transfer time ND, is stored in list B . B is then sorted by the mean of the accumulated ND.

The loop is then entered to produce C . For the first two QLs in B , the accumulated NDs are compared: a random sample is taken from each ND, and the QL q with the smaller sample is added to C . All other QLs lower or equal to q are removed from B . If only one QL is left, it is chosen without competition. If B is not empty, the scheduler adds the processing time ND of q to the accumulated ND of all QLs on the same render side, sorts B again, and proceeds with the next iteration.

We can see the first scheduled QL is either the first or second QL in B . QLs with higher completion time, and thus more likely to disrupt an interactive frame rate, are placed further up in B . They can not occasionally end up as the interactive QL anymore.

To conclude, the distribution-comparison strategy does not account for all the possible schedule outcomes like the pre-sampling strategy. Still, it can provide a smoother user experience. Both strategies are viable, and a selection can be done by personal preference. The system does not automatically switch between the two. Though, we consider to investigate runtime detection of the pre-sampling strategy side effect to automatically decide whether to fall back to distribution-comparison.

4.4 Distribution Initialization and Auto-Schedule

When a render session is established, our system has not gathered any timings yet. Before a QL can be considered for the scheduling algorithm, the system has to acquire a minimum amount of timings to initialize mean and variance of the QL's NDs. Until this state is reached, a QL is automatically scheduled to facilitate the timing acquisition.

To account for initialization and warm-up related outliers, the scheduler can be set up to ignore the first N processing timings. We assume an initial rendering of a QL to be a likely outlier. The $N + 1$ th timing initializes a ND, and is set as the initial mean.

In addition, the initial variance needs to be determined. There is limited information on-hand with the $N + 1$ th timing, which could still be an outlier. Therefore, the system artificially adds variance to a ND newly initialized from a single timing. Starting with uncertainty in the schedule benefits the acquisition of timings across the QLs, gradually narrowing down the NDs and the schedule according to the actual conditions.

Consequently, we set the initial variance of a ND to the maximum valid variance. Let us define an interval $[0, 2 \cdot \text{mean}]$. We want to ensure random samples taken from the ND fall into this interval with a probability of one. However, the cumulative distribution function (CDF) never becomes zero. We therefore use $1 - \varepsilon$ as the probability in our calculation for a very small ε . The variance is determined by solving the CDF for deviation:

$$0.5 \cdot \left(1 + \operatorname{erf} \left(\frac{h}{d \cdot \sqrt{2}} \right) \right) = p + \frac{1-p}{2}.$$

Solving for the variance v results in:

$$v = \left(\frac{0.707107 \cdot h}{\operatorname{erfinv}(p)} \right)^2,$$

where erfinv is the inverse error function, h the half length of the interval, p the required probability, and d the deviation.

4.5 Distribution Update and Reset

A newly initialized ND is a sparse representation of the current conditions, as only one timing has been acquired so far. A ND is updated with further timings as they become available. The scheduler can be set up to accumulate more timings before a ND is considered meaningful, and included in the scheduling algorithm. However, a meaningful ND which has not been updated a long time loses its significance. We introduce the reset mechanism to trigger the acquisition if new timings for such a ND.

Timings are available if a QL scheduled for rendering was completed. However, a QL is not guaranteed to complete as rendering may be aborted. In this case, timings may still be available through estimation (Section 4.7). As rendering and network transfer are expensive, the scheduler may not schedule a QL in the first place (Section 4.3.1). Concluding, our system is not guaranteed to obtain timings for each QL in frequent intervals.

The scheduler adds timings to a ND using a time-based weighting system. The timings must not be retained, as mean and variance of a ND are updated incrementally [15]. We identify two characteristics our weighting method should have. First, it should be resistant to outliers. Second, newer timings should have more significance. Our approach takes the time

since the last timing into account to determine the weight for a new timing. A weighting function, with the passed time in milliseconds as a parameter, can be defined for each QL. The function returns the factor of weight difference between the last and the current timing. The new weight is calculated as

$$\text{weight} = \text{oldWeight} \cdot \text{wf}(\text{elapsedTime}).$$

Weights are increased linearly on default, but an exponential function or another approach can be used at will. There is a tradeoff between adapting to changing conditions quickly, and being outlier resistant.

The more time passes since a ND was updated, the less representative it becomes. Conditions might be different than before, and thus timings lose their significance over time. In terms of the weighting function, the larger the weight of a new timing is, the less significant the previous ones are. To account for this, the system allows to set up the maximum normalized weight of a timing. If this weight would be reached, a ND is reset to the uninitialized state to facilitate the acquisition of up-to-date timings. Previous timings have lost their significance, and we must assume startup outliers possibly happening again.

4.6 Obtaining Processing and Transfer Timings

This section describes how the timings to generate the NDs are obtained. We have to distinguish between measured timings, which are available if a QL completed, and estimated timings which may be available if rendering a QL was aborted (Section 4.7). There are two types of timings: processing and transfer time.

For processing time measurement, the render time, and in case of the server also the image encoding time, counts.

For the transfer time, the client starts a timer before requesting a list of server QLs. The timer is probed if the response for a QL is received. This is the waiting time. The response includes a fixed sized header containing the processing time amongst other information. For a completed QL, it includes the image data. Transfer time can be determined as follows:

$$t = (pt + w) - p, \quad (1)$$

where t is the transfer time. pt is the transfer time of the previous QL completed in the request. For the first requested QL, pt is zero. w is the waiting, and p the processing time.

pt must be included in Equation (1) for two reasons. First, the server immediately progresses after having sent the image for a completed QL. Consequently, rendering continues during the transfer time of previously completed images. Second, the latency when sending a rendering request from client to server is only included in the measurement for the first QL. It must be considered for subsequent QLs as well.

4.7 Rendering Abort and Timing Estimation

Here we describe the unfavorable effects of incomplete and lost timing data caused by rendering abort, and how our system can compensate the loss by estimating both processing (Section 4.7.1) and transfer time (Section 4.7.2).

In our system, rendering can be aborted to maintain interactivity. View refinement is interrupted if a new frame is about to start due to a user interaction. However, interrupting the rendering prevents the system from completing its time measurements. There is only a partial measurement for the QL directly aborted while rendering, and no measurement for consecutive QLs which are still to be processed. Missing timings can result in the NDs becoming unrepresentative, and prevents the scheduler from adapting. A warped schedule not reflecting the actual conditions could then be kept up. To mitigate these effects, timing estimation is in place.

4.7.1 Processing Time Estimation—To enable the estimation of render time, we introduce the concept of work units. For each QL, the render system defines a number of work units it requires. QL n has less units than QL $n + 1$. We expect each unit to require about the same render time under stable conditions. The mapping of actual rendering work to linear work units is up to the render system.

Such a mapping may only approximately be possible. For *Tuvok*, a direct mapping to the number of bricks to be rendered is used. However, there may be significant differences in the render time of bricks. While estimations may be off, they at least likely push a ND in the right direction. The advantage of keeping the state up-to-date outweighs the possible inaccuracy. The outlier-resistant weighting system (Section 4.5) is in place to absorb the impact of largely off estimations.

When rendering the list of QLs requested for a frame, the system tracks the average work unit completion time. On the server, the system also tracks the average encoding time. After an abort, the estimated remaining processing time for a QL computes as follows:

$$ep = (w \cdot wc) + e,$$

where ep is the estimated remaining processing time, w the number of work units to complete, wc the average work unit completion time. e is the average encoding time.

If the first requested QL is aborted, no image has been encoded yet. Encoding time is then ignored, assuming it is a substantially less significant factor than render time.

4.7.2 Transfer Time Estimation—Transfer time is dependent on latency, bandwidth, and the size of the image to send. These factors are taken into account to estimate transfer time.

For each aborted QL, the server estimates the encoded image size in bytes using the last image completed in the request. If the first QL is aborted, image size and thus transfer time can not be estimated.

The client tracks the rate at which image data can be transferred in bytes/ms, the transfer rate. Further, the client uses a monitor to approximate the round-trip time in milliseconds. The round-trip time is independent of the image size, and thus excluded from the transfer rate measurement.

Transfer time is estimated as follows:

$$et = \frac{ei}{tr} + rt,$$

where et is the estimated transfer time, ei the estimated image size, tr the transfer rate. rt is the round-trip time.

5 Results

We tested the hybrid rendering in a number of scenarios to demonstrate characteristics of our method.

In Section 5.1, we set out to demonstrate that our hybrid rendering method is able to react to latency and limited bandwidth, to an overloaded server, and that it is able to utilize the available resources by creating a synergy between client and server. We make a comparison with remote rendering, and expect the advantages of the hybrid method to be visible from the results.

In Section 5.2, we set out to demonstrate that our probabilistic scheduler is able to react to the change in the performance behavior of a render side, even if QLs are barely scheduled for rendering on that side. We make a comparison to a deterministic scheduler, which we expect to behave differently, and less accurate.

In Section 5.3, we set out to underline the flexibility of the QL concept by adaptively refining a large data set into a variable amount of QLs. We compare this approach to the standard non-adaptive version of the system in a specific scenario, and expect the adaptive approach to provide better performance in that scenario.

Table 1 lists the client and server machines used.

Tuvok was the renderer in the tests. The data set for Sections 5.1 and 5.2 is the *Visible Human (VH)* (Fig. 1), $512 \times 512 \times 1884$ 8-bit voxels). In Section 5.3, we use the large *Mandelbulb (MB)* data set ($8192 \times 8192 \times 8192$ 8-bit voxels). To enable remote rendering, we simply set a client up to not support any QLs.

For equal conditions and reproducibility, we automatized the tests by replaying a recorded interaction loop of four minutes length. There is a mixture of interactive phases where the data set is moved in place, and idle phases for examination. We repeated the loop then times for each scenario. The results are the averages from the ten runs (thus fractional parts appear). Time values are given in milliseconds.

For each scenario, if not stated otherwise, hybrid rendering is enabled, all QLs are supported on both sides, and no constraints were put on the network link or the rendering performance of either side.

5.1 Comparison with Remote Rendering

In the following scenarios, the resilience of the the hybrid rendering method towards latency, limited bandwidth and server load is tested. We compare each scenario to remote rendering to underline the advantages of the hybrid approach. Results for each scenario are first given for hybrid rendering.

We present the following information in the upcoming tables and statistics to back up our findings. The tables show for each QL how often it was scheduled for rendering on server and client. It further shows how often a scheduled QL was actually requested for rendering (it may not due to the request timer described in Section 4.3.1), whether it was requested as the interactive QL, and how often it actually completed (it may not due to rendering abort). Based on the tables, characteristics of the hybrid method are derived. For remote rendering, no tables are presented as the scheduling is obviously one-sided with only the server being utilized. We further provide statistics about the average time it took to display the interactive QL each frame (iQL display time), and the average time it took to refine the view with a noninteractive QL (niQL display time). The total processing time spend on each side (PT client/server), and the total transfer time (TT), are given as well.

In *scenario 1* (Table 2), we used the orion server and the thin client. Latency was around 73 ms, and bandwidth around 3.2 Mbit/s.

Frames. 564.5; iQL display time: 152.5; niQL display time: 375.4; PT client: 81168.5; PT server: 28352; TT: 80120.2.

As shown in Table 2, the scheduler reacts to the latency by shifting the interactive QL to the client. The server is still the more powerful machine, and the remaining QLs are mostly rendered on it. The about constant latency becomes less significant as the difference in rendering time between server and client increases for higher QLs.

In Table 2, we can see that especially higher QLs do not always complete if they are requested. QLs that take long to complete are more likely aborted. Further, QL3 and 4 provide detail which is not required for distant views. The renderer thus rejects them for such views (Section 3.2).

We repeated the test with remote rendering.

Frames. 346.9; iQL display time: 285; niQL display time: 440; PT server: 31357.8; TT: 132237.6.

The iQL display time is greatly increased compared to the hybrid case. The latency can not be bypassed by scheduling the interactive QL on the client. Also, the hybrid scheduler is able to create a schedule in which client and server complement each other. In Table 2, we can see the server does not have to bother with QL1 most of the time, and thus can complete

the non-interactive QLs for view refinement faster. This is reflected in the lower niQL display time in the hybrid case.

In *scenario 2* (Table 3), we used the orion server and the thin client. We simulated an additional latency of 150 ms on top of the actual 73 ms, and a bandwidth reduction to 500 KBit/s.

Frames. 551.4; iQL display time: 138.7; niQL display time: 731.4; PT client: 118169; PT server: 21657.8; TT: 172204.2.

The additional network constraints significantly increase the transfer time. The focus has thus shifted further to the client. Table 3 shows QL4 is still almost exclusively scheduled on the server, while QL1, 2 and 3 are concentrated on the client. Interactivity is not hurt by the additional network constraints, as shown by the iQL display time.

We repeated the test with remote rendering.

Frames. 133.2; iQL display time: 760.5; niQL display time: 1400.8; PT server: 16014.9; TT: 241157.8

The iQL display time shows that the system can barely maintain an interactive frame rate. We set up the system to allow aborting the interactive QL after waiting one second for its completion. The interactive QL is indeed regularly aborted for close views where completion time exceeds a second. For QL1, the abort-to-request ratio is 0.5. Aborting the interactive QL, which is the first QL of a frame, ultimately means display updates are lost, and the user input becomes decoupled from the visible result.

In *scenario 3* (Table 4), we used the lab server and the thin client. A latency of 200 ms was simulated. Bandwidth is a negligible factor for this high-speed local area link.

Frames. 585.4; iQL display time: 138.2; niQL display time: 558.7; PT client: 111128; PT server: 15116.8; TT: 126442.2.

This scenario demonstrates the synergy between client and server. Table 4 shows that QL1 and 2 are mostly scheduled on the client to bypass the latency. For QL3 and 4, which take substantially longer to render on the client than on the server, the server remains primarily in charge. The client unburdens the server, and vice versa.

We repeated the test with remote rendering.

Frames. 202.4; iQL display time: 531; niQL display time: 842.1; PT server: 16613.4; TT: 200104.4.

Neither the latency can be bypassed, nor can either side take load off the other. Thus, iQL and niQL display time are increased compared to the hybrid case.

In *scenario 4* (Table 5), we used the lab server and both clients. Measurements were captured on the thin client. 19 remote rendering only sessions were run on the fat client to generate server load. The fat client sessions kept running until the thin client finished.

Frames. 529.3; iQL display time: 150.4; niQL display time: 921.8; PT client: 136729.3; PT server: 148635.4; TT: 868.6

The scheduler responds to the server load by shifting QLs to the client (Table 5). Except for QL4, the focus is on the client. We observe a low completion rate of QL4, since both loaded server and thin client can mostly not deliver in time before the interaction loop progresses.

We repeated the test with remote rendering. The performance loss compared to hybrid rendering is substantial.

Frames. 303.7; iQL display time: 310.6; niQL display time: 1594.4; PT server: 226516.6; TT: 4316.

To conclude, we have demonstrated our hybrid method can overcome high latency, limited bandwidth and overloaded resources by shifting rendering load between client and server. The method is able to create a synergy of the two sides to improve the overall performance. We underlined the results by demonstrating the benefit compared to a remote rendering system. These findings confirm the expectations we originally formulated (Section 5).

5.2 Comparison with Deterministic Scheduler

In the following scenarios, our probabilistic scheduling method is compared to a deterministic approach. We examine the reaction to runtime condition changes. We identify similarities and differences, and outline the advantages of the probabilistic scheduler.

To enable deterministic scheduling, we removed the probabilities by always using the mean of a ND.

We modified the system to enable a runtime change in the rendering performance. Rendering of a QL can be delayed by a factor to simulate load. In each run, the condition change hits after two minutes. We performed the tests with a new loop of four minutes length, which represents a continuous rotation in a fixed distance. By using this simplified loop with a single QL, the behavior of the scheduler over time can be demonstrated most clearly. Results for each scenario are first given for the probabilistic scheduler. QL1 is supported on server and client.

In *scenario 1*, we used the thin client and the lab server. The condition change is a slow down of the server performance by a factor of 7. 5. Fig. 4 shows the schedule amount of QL1 on each render side over time in milliseconds. The schedule amount is the number of times a QL was scheduled for rendering. Note the scale between client and server is not 1:1. The server without delay is substantially faster, and thus a higher frame rate is present if it is the active side. The active side is defined as the render side scheduling is focused on. In contrast, the QL is infrequently scheduled on the idle side.

As expected, the server is the active side until the slow down occurs. Due to the probabilistic nature of the scheduler, the client side is still sampled from time to time. The distribution initialization process (Section 4.4) to adapt to the unknown conditions is reflected in the initial higher schedule amount on the client. Once the slow down hits, the client schedule

amount first decreases before starting to climb. We can see the transition to the client as the active side is not immediate. The focus initially stays on the slowed server, resulting in a lower FPS, and thus fewer chances for the QL to be scheduled on the client. The reason for the incremental shift is the proximity in rendering performance between the slowed server and the client. The advantage of the client is not substantial. The weighting function also affects how quickly new timings cause the overtake. The scheduler's behavior is thus valid given its parameters and the actual conditions.

We repeated the scenario with the deterministic scheduler (no histograms given as server is the only active side).

Scheduling on the client stops after the initialization phase until the slow down. Since the slow down occurs on the active server, the deterministic scheduler notices it. However, unlike our expectation, there is no switch to the client occurring. First, the server side distribution does indeed fall behind the client, causing a schedule of the QL on the client. However, the client timing is an outlier as there was no client-side rendering since the beginning. The client's distribution was thus also not updated since the beginning, and the outlier has a huge impact. This causes the server's distribution, even though it reflects the slowed down server state, to stay in front in the end. The problem here is the abrupt transition, with only a single guaranteed schedule on the client, which is prone to outliers.

In contrast, in the probabilistic approach, there is a smooth transition when one distribution approaches another. Each timing affects the consequent scheduling decisions, accelerating the shift in the direction of change. In a chain-reaction, the chances for one side gradually increase, and decrease for the other. This causes an outlier-resistant transition, with multiple samples taken from both sides, until a stable state representing the changed conditions has been reached. The approach is flexible, as a weighting function can be selected to either favor reacting to change fast or outlier-resistance. Concluding, the deterministic approach is not able to absorb client-side outliers, resulting in permanently keeping up a bad scheduling decision.

We repeated the scenario with the deterministic scheduler, replacing the linear with an exponential weighting function (Fig. 5).

Now the switch to the client is performed similarly to the probabilistic approach. The exponential weighting function causes a distribution reset (Section 4.5) on the idle client about every 45 seconds. This allows the scheduler to obtain several timings depending on the initialization parameters, which here is enough to absorb initial outliers.

Following, similar behavior can be achieved using a deterministic scheduler which probes the idle side with a certain interval. This also applies to the next scenario, which simulates change on the idle side. However, the probing frequency is arbitrary, and does not reflect the actual conditions. There is no chain-reaction to smooth and accelerate the adaption process. Timings do not affect the scheduling decision until one distribution finally overtakes the other, causing an abrupt transition. Probing is no generally applicable concept which can adapt to any situation. A low frequency may result in reacting to change late. In our scenario, switching to the client after a slow down at the 95 second mark would be about 40

seconds late. The timings obtained after a reset might not be enough to complete the shift of a distribution. They might also be heavily influenced by outliers. A high probing frequency might result in generating unnecessary load if conditions are stable, and the two sides far apart performance-wise. If they are close, the probabilistic scheduler is able to reflect this by equalizing the schedule, which is especially relevant to balance multiple-client scenarios. The deterministic approach would instead focus on one side while probing the other with an arbitrary interval, not taking the performance proximity into account.

This underlines the benefit of the generic concept behind our probabilistic scheduling method, which should adapt to any unknown situation not requiring titrated parameters or workarounds.

In *scenario 2* (Fig. 6), we used the thin client and the lab server. The condition change is a speed up of the server performance by a factor of 7.5. The server's performance is slowed by this factor initially.

As expected, the client is the active side until the speed up occurs on the idle side. The probabilistic scheduler is able to notice the change as it continues to schedule on the server from time to time. In contrast to *scenario 1*, the transition concludes quickly. Once the server regains full performance, the difference to the client is substantial, allowing server-side timings to quickly put the server ahead.

We repeated the scenario with the deterministic scheduler (no histograms given as client is the only active side).

The scheduler does not notice the condition change. The server is left alone entirely after initialization. Concluding, without a workaround, such as forcing to reactivate the idle side every N time steps, the deterministic scheduler is not able to react to change occurring on the idle side only.

We repeated the scenario with the deterministic scheduler, replacing the linear with an exponential weighting function (Fig. 7).

Now the switch to the server is performed similar to the probabilistic approach. The reset caused by the exponential weighting function causes the QL being auto-scheduled on the server. Several timings can be obtained, which are enough to trigger the transition, though at a later point. However, the result is again specific to the situation and the scheduling parameters chosen, and does not reflect a generically applicable concept.

Summarizing, the probabilistic scheduler has an advantage in absorbing outliers, especially in the transition phase when one side overtakes the other, and reacting to change on the idle render side. Still, a problem with the latter can occur if the probability for a QL to get scheduled on the idle side gets very low or even towards zero. Such a scenario is possible if the two sides are far apart performance wise, and timing variance is low. However, we assume condition changes can affect the situation any time. They can not be predicted given the current state represented by the distributions. Therefore, a mechanism independent of the state is required to guarantee change is noticed in a timely manner. The weighting function

can be customized to workaround the problem. A reset could be triggered faster, or in predefined intervals. It is up to the developer what kind of logic is put into the function. We provide predefined options, including the linear and exponential versions used in the paper.

The generic solution is called probabilistic auto-schedule. The more a QL's distribution approaches the reset threshold in terms of the weighting function, hence the longer the distribution has not been updated, the higher the probability for the QL to get auto-scheduled is. The decision is evaluated every frame for every QL. Parametrization is possible to control how fast and to what maximum the probability rises. It is an optional component, recommended in environments where substantial condition changes are expected. Note that using auto-scheduling extensively can be counter-productive, causing unnecessary load.

To conclude, our probabilistic scheduler is able to mask unexpected delay, and react to changes in the performance behavior of either side. While similar results can be achieved with a deterministic approach in some situations, such a method especially fails to adapt to condition change on the idle side reliably. Our system provides parameters and optional features to tune the scheduling if desirable. Probabilistic auto-schedule and an exponential weighting function have proven to be a viable options to accelerate the adaption process.

5.3 Large Data Set Scenario

Here, we demonstrate how our system enables the interactive rendering of a large data set on a client with limited resources. We used the thinnest client, and the fat client as the server. The thinnest client is restricted in disk space, with only 60 GB available in total. This is not enough to store the MB in its uncompressed form. In its compressed form, the MB still occupies more than 23 GB, making it infeasible to store on such a machine (considering an additional 15 GB for the operating system alone). We therefore only enable a subset of the QLs on the client.

Further, the client is restricted in rendering performance. The MB has seven LODs. When using a direct LOD-to-QL mapping as with the VH, even the first QL renders barely interactively (iQL display time of 620.2 with client-side only rendering of QL1 using the interaction loop of Section 5.1). To improve interactivity, and enable a more fine-grained view-refinement, we extended the QL mapping for *Tuvok* with an adaptive approach. Each LOD can be split into a variable amount of QLs. These QLs are distinguished by the sampling frequency and resolution at which they are rendered. Frequency and resolution are runtime parameters of the renderer. Fig. 8 shows example QLs for an LOD of the MB.

For the test, each LOD was split into two QLs, resulting in 14 QLs overall. We introduced 28 ms of latency to facilitate iQL rendering on the client. QL1-6 were supported on the client.

Frames. 734.5; *iQL display time:* 82.7; *niQL display time:* 286.8; *PT client:* 42401.6; *PT server:* 101725.6; *TT:* 35840.9.

The iQL is focused on the client (62.7 percent schedule share). Though, for close views, it is regularly scheduled on the server as rendering time increases, reducing the latency impact. niQLs are almost exclusively scheduled on the server.

We repeated the test without the adaptive QL mapping, thus ending up with seven QLs (QLN equivalent to $QL2 * N$ above). QL1-3 were supported on the client.

Frames. 514.1; iQL display time: 146; niQL display time: 487.1; PT client: 14797.4; PT server: 114073.5; TT: 37772.1.

Almost all rendering including the iQL is performed on the server. The latency has no impact on the schedule, as the performance difference between client and server is too substantial. Both iQL and niQL display time are increased.

The results underline the flexibility of the abstract QL concept, which allows an arbitrary, application-specific mapping of a data set to QLs. A substantially higher frame rate could be achieved using the adaptive mapping, especially on the client.

6 Normal Distribution Usage

Our system obtains discrete timings to gradually approach the underlying continuous distribution, which we assume to be normal. In general, the distribution characteristics are unforeseeable. They are dependent on unknown factors, especially attributed to renderer, operating system, hardware and network. The distribution type may differ between data sets, QLs and even views. Along with changing conditions, distribution characteristics may change. Insight could be obtained for a specific setup with conditions as stable as possible. However, such a pre-process assessment does not apply to our generic use-case. We support arbitrary server and client machines, and make no assumptions about the possibly changing conditions during a render session.

We performed a number of tests using the machines and data sets presented in Section 5 to confirm the ND is a reasonable choice in the majority of cases. We also used our progressive mesh based renderer with a data set we call the *Thai Statue (TS)*. The TS has six QLs, with the number of triangles ranging from 100,008 to 10,000,000.

In each test, a single QL was rendered repeatedly for four minutes in a static view, with no additional load on the test machine. We repeated the tests with the data set continuously slightly moving (rotating or zooming in and out). We visualize the timings gathered by the system in a histogram including a ND fitted to the data. A ND has proven to be a good fit in most cases. Figs. 9 and 10 show the results for selected scenarios.

When enabling the movement, NDs of higher variance could still be fitted to much of the data. In some cases, the different views caused by the movement resulted in several peaks in the histogram, and thus more than one distribution being present. This is expected, and we assume a ND can be fit to the timing sets representing a specific view.

We further performed tests using the fat client and the orion server to obtain transfer timings. Generally, a ND with very low variance could be fitted to the data. However, major

outliers were regularly present in some test runs. Remember the orion server is reached via the Internet, with possible external influences on the link.

The results show the ND represents the timing data we generated decently. It is thus a feasible choice to approximate the performance state with NDs. We ultimately decided to use the ND as the default distribution type in our generic use-case.

However, closed scenarios are possible where specific client, server and network components as well as renderers and data sets are used in a stable environment. In such a case, tests like the ones described could be performed to gain understanding about the distribution characteristics. Fig. 10 also fits a generalized extreme value distribution (EVD) and an error distribution to the timing data in comparison to the ND. We can see those distributions are more suitable in the depicted cases. A ND is still a reasonable fit. However, the ND deviates more clearly from the histogram in the middle scenario of Fig. 10.

While our system uses NDs on default to be deployed in a heterogeneous, uncertain setup, it is not bound to NDs. The pre-sampling strategy is independent of the distribution type, as it only relies on random sampling. Thus, the system is configurable, and allows the distribution type being replaced. There are other parameters like the weighting function, which can be used for adjustment. In the future, we will investigate adding built-in support for additional distribution types, especially the EVD, which has shown to fit some timing data well.

7 Conclusion and Discussion

We presented a generic hybrid rendering method, which distributes workload to server and client in terms of QLs of a data set. We use a probabilistic scheduling algorithm to account for the various uncertain factors when determining which QLs are to be rendered on which side. The system obtains and updates timing NDs for the QLs at runtime to allow the schedule adapting to initially unknown conditions, which may be subject to change. Outliers are absorbed by the weighting system used when adding timings. Our method enables us to find a balance between utilizing server and client resources. Local rendering capabilities on a client reduce dependency on server and network. Utilizing the client puts less load on server and network, allowing these components to scale well. We have successfully plugged in an interactive, LOD-based volume renderer, *Tuvok*, and a progressive mesh renderer into the system.

In a steady situation, the scheduling algorithm converges to a state reflecting the current conditions. However, the process of adapting may not be immediate due to the possible irregularity in the scheduling of QLs (Section 4.3.1). While this is expected behavior in terms of our probabilistic view on the system, it implies two problems.

First, if the probability for QLs to be scheduled on a side is low, the scheduler may only slowly react to a condition change on the side. This especially applies to server QLs, where substantial timing fluctuation is more likely due to arbitrary server and network load. The user does not benefit from an improved server performance until our schedule incorporates the change. How fast adaption progresses is also dependent on whether and in what direction

conditions change on the other side. If client performance decreases, the probability for server QLs to be scheduled is indirectly increased. Further, the weighting function determines how much a timing impacts a ND. But increasing weights fast makes a ND less resistant to outliers. Also, a timing must first be acquired. With arbitrarily low probabilities being possible, resetting a ND and probabilistic auto-schedule ultimately enable updating the state.

Second, if conditions on a side change rapidly, the scheduler may constantly be behind with its predictions, even if QLs are scheduled regularly. An ongoing shift of a ND may be counter-productive if conditions quickly change in the opposite direction. This is more likely for server QLs, where a sudden increase or drop in load is possible.

Non-interactive QLs may be scheduled, but not requested for rendering due to the request timer (Section 4.3.1). Measured or estimated timings may thus only sporadically be received for them. Consequently, setting up QLs, which have a high expected completion time, to reach the initialized and meaningful state with fewer timings, along with giving new timings more weight, can be reasonable.

8 Future Work

We have plugged two renderers into the system. For these renderers we can demonstrate the usability of our approach. In the future we will investigate how other types of renderers can be mapped to the QL concept.

We presented results for several test scenarios. These results shall be extended. We want to deploy the system more widespread by incorporating additional devices, especially mobile ones, which are placed at different locations. We want to incorporate the interleaved data transfer. We are also considering a cloud computing environment for deployment. The cloud concept becomes increasingly popular to provide services to heterogeneous clients. Hybrid rendering could be such a service. An example is *Amazon EC2*.

We described two scheduling strategies. In both, the goal is to provide the user with the next QL as soon as possible. We want to investigate how an alternative strategy could be designed, which gives more priority to the scalability of the system. The strategy would not necessarily select the schedule optimal for the user, but find a compromise, which is still acceptable in terms of user perception, but puts less load on server and network.

Further, when determining the next non-interactive QL to schedule, the scheduler always selects the one expected to complete earliest. However, the time difference to following QLs may be minimal in terms of user perception. It could be beneficial to prefer a higher QL, or the client instance, if a delay or missing refinement step would not or barely be noticeable. It would enable us to put less load on the system, while possibly reaching the highest QL faster.

References

1. The Opportunities and Challenges of Exascale Computing. 2010. Report of the advanced scientific computing advisory committee (ascac) subcommittee, u.s. department of energy.

2. Office of advanced scientific computing research (ascr), u.s. department of energy. Scientific Discovery at the Exascale: Report from the DOE ASCR 2011 Workshop on Exascale Data Management, Analysis, and Visualization; 2011.
3. Aranha, M.; Dubla, P.; Debattista, K.; Bashford-Rogers, T.; Chalmers, A. A Physically-Based Client-Server Rendering Solution for Mobile Devices. Proc. Sixth Int'l Conf. Mobile and Ubiquitous Multimedia (MUM '07); 2007. p. 149-154.
4. Balasubramanian J, Grossmann IE. A Novel Branch and Bound Algorithm for Scheduling Flowshop Plants with Uncertain Processing Times. Computers and Chemical Eng. 2001; 26:4–1.
5. Bethel, EW.; Childs, H.; Hansen, C. High Performance Visualization: Enabling Extreme-Scale Scientific Insight. 1. Chapman & Hall/CRC; 2012.
6. Bethel, W.; Tierney, B.; lee, J.; Gunter, D.; Lau, S. Using High-Speed Wans and Network Data Caches to Enable Remote and Distributed Visualization. Proc. 2000 ACM/IEEE Conf. Supercomputing (Supercomputing '00); 2000.
7. Bharadwaj V, Ghose D, Robertazzi TG. Divisible Load Theory: A New Paradigm for Load Scheduling in Distributed Systems. Cluster Computing. Jan; 2003 6(1):7–17.
8. Butson CR, Tamm G, Jain S, Fogal T, Krüger J. Evaluation of Interactive Visualization on Mobile Computing Platforms for Selection of Deep Brain Stimulation Parameters. IEEE Trans Visualization and Computer Graphics. Jan; 2013 19(1):108–117.
9. Callahan SP, Bavoil L, Pascucci V, Silva CT. Progressive Volume Rendering of Large Unstructured Grids. IEEE Trans Visualization and Computer Graphics. Sep; 2006 12(5):1307–1314.
10. Diepstraten J, Gorke M, Ertl T. Remote Line Rendering for Mobile Devices. Proc Computer Graphics Int'l (CGI '04). 2004:454–461.
11. Dyken, C.; Lye, K.; Seland, J.; Bjornnes, E.; Hjelmervik, J.; Nygaard, J.; Hagen, T. A Framework for OpenGL Client-Server Rendering. Proc. IEEE 4th Int'l Conf. Cloud Computing Technology and Science (CloudCom); 2012. p. 729-734.
12. Engel K, Ertl T, Hastreiter P, Tomandl B, Eberhardt K. Combining Local and Remote Visualization Techniques for Interactive Volume Rendering in Medical Applications. Proc Conf Visualization (VIS '00). 2000:449–452.
13. Engel K, Oellien F, Ertl T, Ihlenfeldt WD. Client-Server-Strategien Zur Visualisierung Komplexer Struktureigenschaften in Digitalen Dokumenten Der Chemie (Client-Server Strategies for Visualization of Complex Molecular Structure Properties in Digital Documents in Chemistry). it +ti - Informationstechnik und Technische Informatik. 2000; 42(6):17–23.
14. Engel K, Westermann R, Ertl T. Isosurface Extraction Techniques for Web-Based Volume Visualization. Proc Conf Visualization (VIS '99). 1999:139–146.
15. Finch, T. Incremental Calculation of Weighted Mean and Variance. 2009.
16. Fogal, T.; Krüger, J. Tuvok - An Architecture for Large Scale Volume Rendering. Proc. 15th Vision, Modeling and Visualization Workshop; 2010.
17. Hendrickson, C.; Au, T. Project Management for Construction. Prentice Hall; 1989.
18. Hoppe, H. Progressive Meshes. Proc. ACM SIGGRAPH; 1996. p. 99-108.
19. Ierapetritou, M.; Li, Z. Modeling and Managing Uncertainty in Process Planning and Scheduling. In: Chaovalitwongse, W.; Furman, KC.; Pardalos, PM., editors. Optimization and Logistics Challenges in the Enterprise. Vol. 30. Springer, Springer Optimization and Its Applications; 2009. p. 97-144.
20. Janak SL, Lin X, Floudas CA. A New Robust Optimization Approach for Scheduling under Uncertainty: II. Uncertainty with Known Probability Distribution. Computers and Chemical Eng. 2007; 31(3):171–195.
21. Li, M.; Schmitz, A.; Kobbelt, L. Pseudo-Immersive Real-Time Display of 3D Scenes on Mobile Devices. Proc. Int'l Conf. 3D Imaging, Modeling, Processing, Visualization and Transmission (3DIMPVT '11); 2011. p. 41-48.
22. Li Z, Ierapetritou M. Process Scheduling under Uncertainty: Review and Challenges. Computers and Chemical Eng. 2008; 32(4/5):715–727.
23. Luke EJ, Hansen CD. Semotus Visum: A Flexible Remote Visualization Framework. Proc Conf Visualization (VIS '02). 2002:61–68.

24. Okamoto Y, Oishi T, Ikeuchi K. Image-Based Network Rendering of Large Meshes for Cloud Computing. *Int'l J Computer Vision*. Aug; 2011 94(1):12–22.
25. Prohaska S, Hutanu A, Kahler R, Hege H-C. Interactive Exploration of Large Remote Micro-Ct Scans. *Proc Conf Visualization'04 (VIS '04)*. 2004:345–352.
26. Teresco J, Fair J, Flaherty J. Resource-Aware Scientific Computation on a Heterogeneous Cluster. *Computing in Science Eng*. 2005; 7(2):40–50.
27. US National Library of Medicine. The Visible Human Project. Apr. 2012 <http://www.nlm.nih.gov/research/visible>
28. Vassiliadis C, Pistikopoulos E. Maintenance Scheduling and Process Optimization under Uncertainty. *Computers and Chemical Eng*. 2001; 25(2/3):217–236.
29. Viswanathan S, Veeravalli B, Robertazzi TG. Resource-Aware Distributed Scheduling Strategies for Large-Scale Computational Cluster/Grid Systems. *IEEE Trans Parallel and Distributed Systems*. Oct; 2007 18(10):1450–1461.

Biographies

Georg Tamm studied games software development at Sheffield Hallam University in England and finished with a master's degree in 2008. In addition, he studied media and communication computer science at Reutlingen University and graduated with a master's degree in 2009. His research topics at the Interactive Visualization and Data Analysis group in Saarbrücken focus on interactive rendering of large data sets, and scalable client-server rendering systems.

Jens Krüger received the diploma in computer science from the Rheinisch-Westfälische Technische Hochschule Aachen in 2002, and the PhD degree from the Technische Universität München in 2006. After attaining postdoc positions in Munich and at the Scientific Computing and Imaging Institute, he became the research assistant professor at the University of Utah. In 2009, he joined the Cluster of Excellence to head the Interactive Visualization and Data Analysis Group. Since 2013, he has been a chair of the High Performance Computing Group at the University of Duisburg-Essen. In addition to his position in Duisburg-Essen, he also holds an adjunct faculty title of the University of Utah and is a principal investigator of multiple projects in the Intel Visual Computing Institute. He is a member of the IEEE.

Fig. 1.
QLs (4 to 1) of the male *Visible Human* volume data set [27] rendered with *Tuvok*.

Fig. 2.

Determining the QLs a client supports during the handshake phase, and transferring the corresponding data to enable rendering on the client.

Fig. 3.

Illustration of a frame's rendering. Scheduling runs within the main display thread concurrently to the local rendering thread. On the server, each client is handled in a separate rendering thread. Once user input is received, the scheduling algorithm determines the QLs to be rendered for the upcoming frame. The display thread simultaneously requests server and client QLs for rendering. It then gradually receives rendered images and corresponding timings. The scheduler update its state with the timings. Arriving QLs are displayed to refine the view. Should new user input arrive before view refinement concludes, the client can prompt to interrupt the rendering process to maintain interactivity.

Fig. 4.
Schedule amounts in response to a server slow down using the probabilistic scheduler.

Fig. 5. Schedule amounts in response to a server slow down using the deterministic scheduler with an exponential weighting function.

Fig. 6.

Schedule amounts in response to a server speed up using the probabilistic scheduler.

Fig. 7. Schedule amounts in response to a server speed up using the deterministic scheduler with an exponential weighting function.

Fig. 8. LOD1 of the MB split into three QLs. QL3 (right) is rendered at full resolution and with the default sampling frequency. These parameters are reduced for the lower QLs, enabling faster rendering.

Fig. 9.

A ND fitted to the histogram of processing timings gathered in six scenarios.

Fig. 10.

A ND fitted to the histogram of processing timings in comparison to a EVD and an error distribution.

TABLE 1

Server and Client Machines Used in the Test Scenarios of Section 5

Site 1 (Saarbrücken, Germany)		
Name	CPU & Memory	GPU & Network
lab server	Intel i7-2600K @ 3.4GHz 16GB	GeForce GTX 680 1GBit/s
fat client	Intel i7-860 @ 2.8GHz 8GB	GeForce GTX 560 1GBit/s
thin client	Intel Pentium E5500 @ 2.8GHz 2GB	GeForce GT 420 100MBit/s
thinnest client	AMD E-450 @ 1.65GHz 3.6GB	Radeon HD 6320 1GBit/s
Site 2 (Salt Lake City, USA)		
orion server	Intel i7-2600 @ 3.4GHz 16GB	GeForce GTX 560 Ti 1GBit/s

TABLE 2

Scheduling Statistics for Test Scenario 1

	Scheduled	Requested	Interactive	Completed
QL1	S 48	48	48	43.2
	C 489.8	489.8	489.8	471.2
QL2	S 317.4	59.4	19.1	53.7
	C 181.1	32.3	13.6	28.3
QL3	S 432.2	54.5	0	43
	C 76.8	9.3	0	4.9
QL4	S 558.5	68.2	0	20.1
	C 161.7	16.5	0	0.3

TABLE 3

Scheduling Statistics for Test Scenario 2

		Scheduled	Requested	Interactive	Completed
QL1	S	7	7	7	1.8
	C	490.6	490.6	490.6	475.7
QL2	S	55.1	9.3	0	7.2
	C	489.7	113.8	59.8	86.7
QL3	S	208.2	30.8	0	18.7
	C	352.9	39.5	0	17.1
QL4	S	542.7	65.9	0	18.3
	C	152	16	0	0.1

TABLE 4

Scheduling Statistics for Test Scenario 3

	Scheduled	Requested	Interactive	Completed
QL1	S 8.5	8.5	8.5	3.1
	C 523	523	523	499.3
QL2	S 113	18.8	1.3	18.2
	C 435.7	102.8	57.6	81.5
QL3	S 335.9	42.2	0	35.3
	C 238.9	25.5	0	12.3
QL4	S 582.4	69.6	0	29.1
	C 159.1	16.3	0	0.1

TABLE 5

Scheduling Statistics for Test Scenario 4

		Scheduled	Requested	Interactive	Completed
QL1	S	202	202	202	164
	C	453.1	453.1	453.1	445.6
QL2	S	110.3	19.3	3.8	16.2
	C	407.7	102.2	57.8	84
QL3	S	222.7	30.3	0.3	20.1
	C	323.3	39	0	16.5
QL4	S	386.8	48.3	0	5.8
	C	296.3	36.8	0	2.8