D5.3.6 – Remote hybrid rendering: tool evaluation and investigation with application data

WP5: User tools

Project Acronym	CRESTA
Project Title	Collaborative Research Into Exascale Systemware, Tools and Applications
Project Number	287703
Instrument	Collaborative project
Thematic Priority	ICT-2011.9.13 Exa-scale computing, software and simulation

Due date:	M39
Submission date:	31/12/2014
Project start date:	01/10/2011
Project duration:	39 months
Deliverable lead organization	USTUTT
Version:	1.0
Status	Final
Author(s):	Martin Aumüller (USTUTT)
Reviewer(s)	Luis Cebamanos (EPCC), Dmitry Khabi (USTUTT)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	28/11/2014	First version of the deliverable	Martin Aumüller (USTUTT)
0.2	05/12/2014	Address reviewer comments	Martin Aumüller (USTUTT), Luis Cebamanos (EPCC), Dmitry Khabi (USTUTT)
1.0	08/12/2014	Final version for submission	Catherine Inglis (UEDIN)

Table of Contents

Index of Figures

Index of Tables

1 Executive Summary

Remote hybrid rendering (RHR) is used to access remote exascale simulations from immersive projection environments over the Internet. The display system may range from a desktop computer to an immersive virtual environment such as a CAVE [10]. The display system forwards user input to the visualization cluster, which uses highly scalable methods to render images of the post-processed simulation data and returns them to the display system. The display system enriches these with context information rendered locally, before they are shown. RHR decouples local interaction from remote rendering and thus guarantees smooth interactivity during exploration of large remote data sets.

The protocol for RHR only sends viewing parameters, derived from user interaction and head tracking, from client to server, which responds with 2.5D images, which are merged with locally rendered content. This design enables the cooperation of light-weight renderers with display programs that contain most of the application logic and interaction handling. This allows for easy integration of RHR with a multitude of applications that operate on a 3-dimensional domain. The sole requirement is that the application is able to generate color images together with depth data describing the distance of the visible pixels to the viewer.

For evaluating RHR, the distributed memory parallel visualization tool Vistle [2] has been implemented. RHR is composed with a scalable rendering system employing sort-last parallel rendering. With a CPU based remote ray caster [21], extraction of isosurfaces and cutting surfaces can be controlled interactively from virtual environments. This system was used successfully on various display systems. Interaction is smooth due to high local display update rates. Cutting surfaces and isosurfaces are generated based on input from within the virtual environment. The goal of decoupling interaction from remote rendering latencies has been achieved.

Compared to classic remote rendering, RHR allows for lightweight rendering server implementations. In a context where the rendering server is replicated many times, e.g. for in situ visualization tasks, this is an advantage.

2 Introduction

Remote hybrid rendering is used to make the post-processing resources used in large-scale cluster systems available to remote users. This document describes the experience gained from implementing and evaluating the prototypical tools developed to this end.

This document is structured as follows: following this introduction, we describe the motivation for the system in section 3. Section 4 describes the system. In section 5, we evaluate the tool both quantitatively and subjectively. We conclude with a comparison of classic remote rendering and remote hybrid rendering, and a discussion of unsolved problems in section 6.

2.1 Glossary of Acronyms

2.5D	Image data together with depth data
6DOF	6 degrees of freedom, usually position and orientation
API	Application Programming Interface
CAVE	Cave automatic virtual environment
CPU	Central processing unit
CUDA	Compute Unified Device Architecture (general purpose parallel GPU programming platform)
Full HD	1920x1080 pixels
GPU	Graphics processing unit
HD	High Definition
JPEG	Joint Photographic Experts Group
OpenGL	Open Graphics Library (graphics rendering API)
PSNR	Peak-signal to noise ratio
QDR	Quad-Data Rate (InfiniBand at 40 Gbit/s)
RFB	Remote Framebuffer Protocol (used by VNC)
RGBA	Red/Green/Blue/Alpha (framebuffer format for color and opacity)
RHR	Remote hybrid rendering
VNC	Virtual Network Computing
WP	Work package

3 Motivation

Output data of simulations can be large. The Institute of Fluid Mechanics and Hydraulic Machinery (IHS) at the University of Stuttgart uses OpenFOAM to simulate the flow in an entire hydro turbine. Based on the estimated requirement for a dependable simulation of about 1 billion nodes for the whole turbine, the size of a single time step is about 1/4 TB. Storing a full turbine rotation with steps of one degree requires about 90 TB. Transferring that amount of data across a high-speed link (10 GigE) for off-line processing on a user workstation would take more than one day – and would require huge amounts of local storage and processing power. This shows that for exascale data the traditional way of transferring the post-processed geometry data to the display system for local rendering is not possible anymore. In comparison, streaming rendered images of the data set can save bandwidth. Sending uncompressed HD-resolution (1920x1080 pixels) images at 30 frames/s for a whole day would require less than 15 TB of bandwidth – and the image the user is interested in is available immediately, not just after a lengthy preparatory transfer. Additionally, employing image compression techniques can significantly reduce this amount of data without even incurring a visible loss. This technique of transmitting rendered images instead of post-processed data to the display system is called remote rendering. The significantly lowered bandwidth and processing requirements of remote rendering allow efficient use of remote compute resources by a much larger user base.

Head-tracked immersive virtual environments, where rendering is constantly updated according to the user's current head position, require high frame rates and low reaction latencies to achieve a high sensation of presence and to avoid motion sickness [4]. These immersive visualization environments provide more intuitive ways than desktop-based systems to specify the location of regions of interest, cutting planes, seed points for particle traces, or reference points for isosurface extraction. We aim to allow users to experience exascale simulations in such immersive environments over the Internet.

To improve frame rate and reaction times, we try to decouple interaction from network latencies as far as possible, while still not requiring huge volumes of data to be transferred to the client. Only extracted features from simulation results are rendered either directly on the simulation host or on a remote visualization cluster employing scalable methods. But "context information" such as essentially static geometry (e.g. turbine shapes), interaction cues for the parameters controlling the visualization algorithms applied on the visualization cluster, and menus are rendered locally, at a rate independent of the remote rendering. As both remotely and locally rendered images are composited for final display, we call this technique "remote hybrid rendering" (RHR) or "hybrid remote rendering" [1]. This compositing usually takes pixel depth into account, but it might also use opacity information. Figure 1 illustrates the differences between a pure remote rendering and a remote hybrid rendering visualization pipeline.

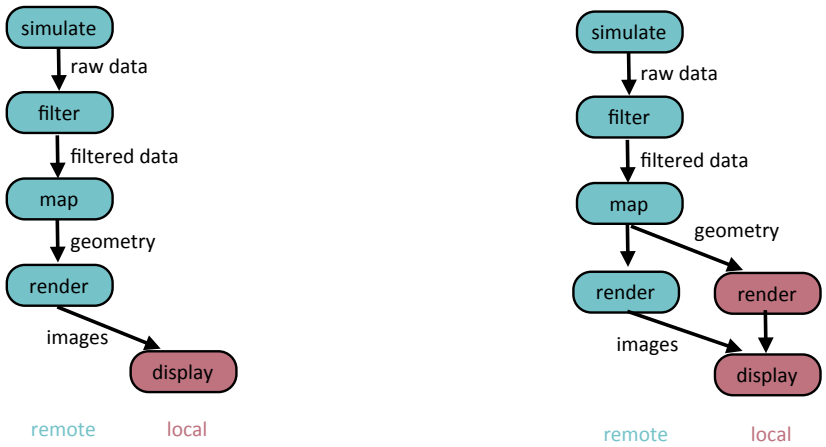

Figure 1: a pure remote vs. a remote hybrid rendering workflow.

Figure 2 shows a visualization of the simulated air flow around a car and illustrates how the image presented to the user results from local context information and remote simulation data.

The remote system is used for post-processing the results of the flow simulation and rendering the corresponding visualizations, such as streamlines as well as a plane cutting through the flow field colorized according to air pressure. The local system renders context information. This comprises the menu and interaction elements, e.g. for moving the cutting plane. But the static geometry of the car is also rendered locally. In a final step before displaying the result, locally and remotely rendered images are composited, taking into account the distance to the viewer of the geometry object contributing the pixel's color: the closer pixel of the two images is copied into the final image.

All the interactive features of the visualization system are available even though parts of the rendering are delegated to a remote system, e.g. new seed points for streamlines can be placed by interacting with the visualization. Only the fact that the remote parts of the image are updated less frequently makes this visualization distinguishable from a purely local visualization.

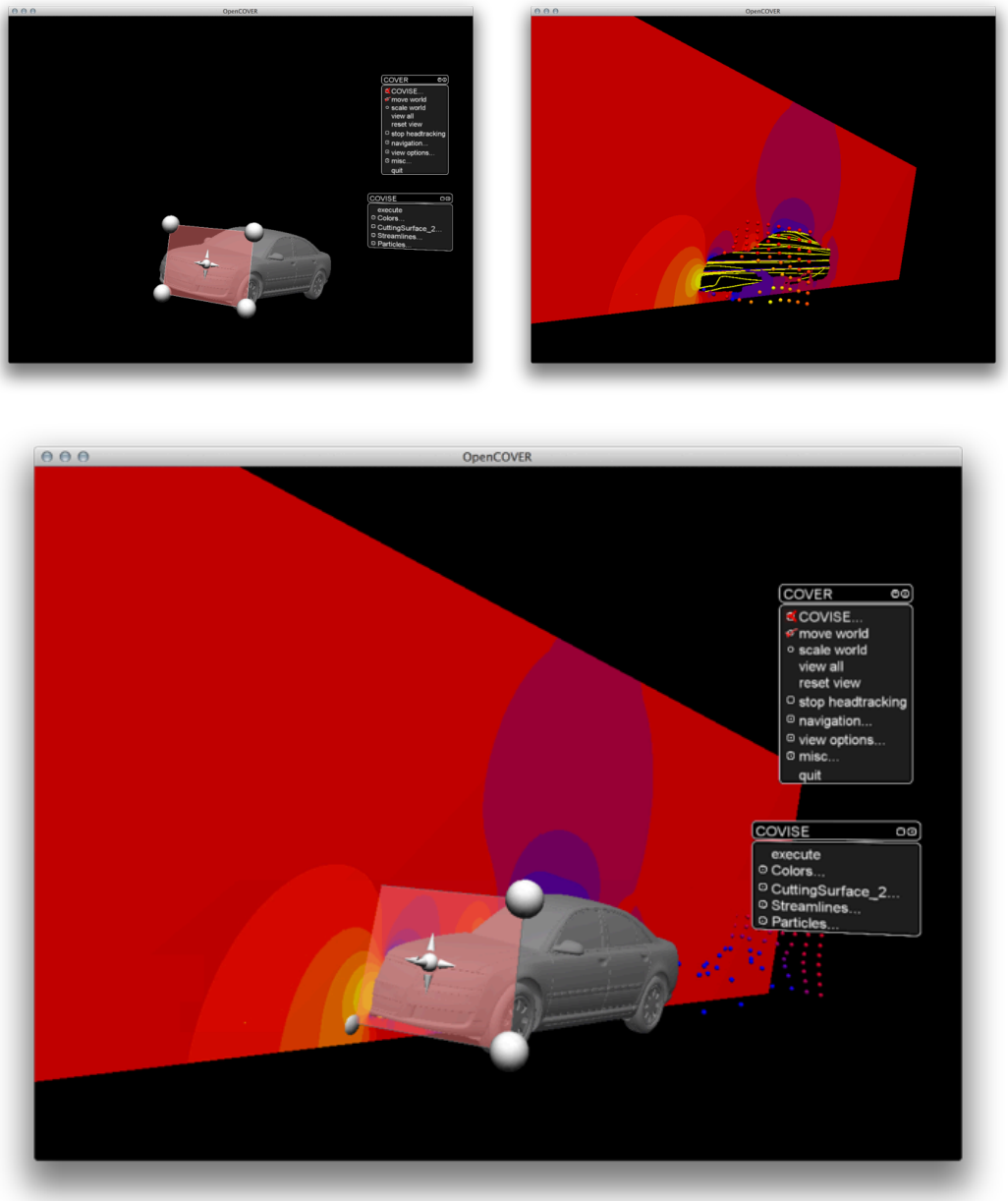

Figure 2: Local context information (top left), remote simulation data (top right), fused image shown to the user (bottom).

4 Design & implementation

4.1 Requirements

4.1.1 Considerations for exascale systems

The environments to which we try to adapt our remote hybrid visualization system are comprised of the following parts:

- a remote exascale compute resource,
- possibly a remote visualization cluster, tightly coupled to the compute resource,
- a local display system.

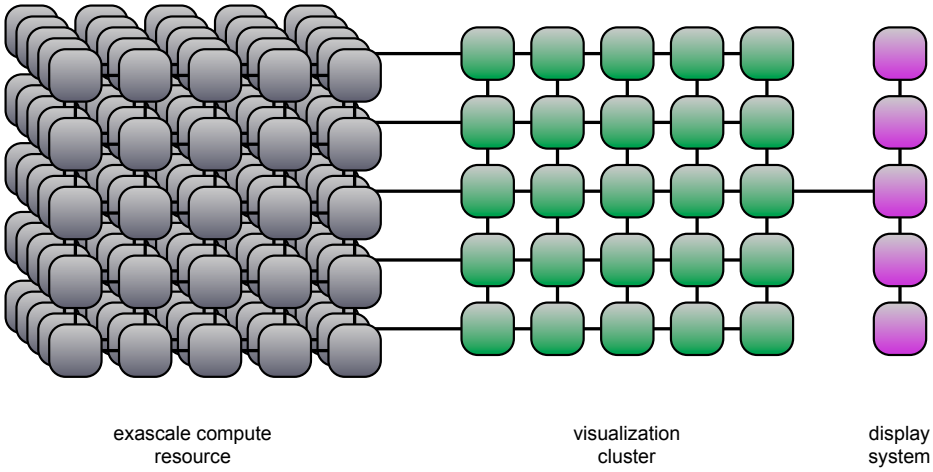

Figure 3: Typical network topology for a remote visualization task, the RHR protocol will be employed on the single link from the visualization nodes (green) to the display system (purple).

In some cases, e.g. when there are GPUs inside each node of the exascale system or with CPU based rendering, the compute system and the visualization system might be the same resource and the GPUs might be used for both simulation and visualization. For all other cases, we assume a high-bandwidth low-latency link of a quality comparable to the exascale cluster interconnect between compute and visualization system. The network connection between remote visualization cluster and display system will provide considerably lower bandwidth and higher latency. While it is desirable to have a higher quality link between visualization and display, this will not always be possible in the case where remote hybrid rendering is used, as this connection will usually be across the Internet.

The network infrastructure might allow for direct connections from each node of the visualization cluster to each node of the display system, but in the general case the network topology or firewalls prohibit this. Hence, we design our system to cope with a point-to-point connection between the head node of the visualization cluster and the head node of the local display system. The protocol should keep the number of simultaneous network connections to a minimum; the establishment of a connection should be possible from client to server and vice versa in order to cater for all possible circumstances.

Sort-last [11] has been selected as the method for parallelizing the render process, as this allows the renderer to be scaled with the application in a data parallel setting. This means that flat pixel images as present in a framebuffer are the result of the rendering phase. The available data for remote rendering is one color value including opacity per pixel together with possibly one depth value. Remote sort-last parallel rendering in a system with the described network topology provides the best performance if compositing happens on the visualization cluster, as this saves bandwidth on the slower link between visualization and display system. The requirement of a point-to-point connection between the head nodes of the visualization cluster and the local display system makes it necessary for the composition of the rendered image data to

happen on the visualization cluster. This ensures that remote hybrid rendering can be composed with scalable rendering methods.

However, rendering context information locally requires a final compositing step in the display system. Depending on the context information to be shown and the rendered data, this requires depth or opacity in addition to the color information for each image pixel.

4.1.2 Requirements for immersive display systems

The display system might be a traditional desktop computer. But the focus of this work is to enable access to remote exascale visualizations from within immersive projection systems. These are distinguished from desktop systems by:

- input devices which record their position and orientation and input methods which exploit this additional information;
- tracking of the user's head position and continuous updating of the rendered image according to the changing point of view (POV);
- 3D stereoscopic imagery, where each eye is presented with an image that is adapted to its position;
- multiple display surfaces, which are used to enhance the resolution (e.g. in powerwalls, where several screens are tiled in one plane to form a larger display area) or to surround the viewer with images (e.g. in a CAVE, where the sides of a cube around the viewer are used as projection surfaces).

It is sufficient to serve one display system at a time. But such a system might possibly consist of several display surfaces, each of which may be a stereographic display. Updates to different display surfaces have to be synchronized in order to enable correct 3D stereoscopy across all surfaces. This might incur longer latencies, when the images for all tiles are not available at the display client at the same time, but this synchronization is vital for immersive display environments. With our design, where all data transfer is funneled through the head nodes of the local and remote systems, synchronization between the nodes attached to a tiled display naturally happens in the client application. On the other hand, reprojection of 2.5D images according to current viewing parameters automatically brings all tiles into a matching state, so that synchronization becomes unnecessary.

4.1.3 Performance requirements

Communication overhead should be minimal. Network round trips, e.g. waiting for acknowledgement of successful delivery of messages, have to be avoided in order to guarantee good performance. For local area connections, TCP based protocols have proven superior, whereas in wide area networks, UDP based protocols seem to have an advantage [14]. We expect the principal use case to be from within local area networks or within networks providing a similar connection quality, such that we prefer TCP to UDP.

In order to be able to balance visual accuracy with performance, the RHR protocol has to allow for different encodings and compression algorithms. And to accommodate changing network circumstances (bandwidth and latency variations), these have to be switchable at run-time. Compression should not visibly decrease image quality for either line drawings or images with huge amounts of gradients, e.g. from volume rendering.

The RHR protocol should not hinder the off-load of suitable tasks, such as image compression or decompression, to accelerators, such as GPUs. This mostly concerns the image codecs to be used. Hence, we want to allow the easy addition of new codecs. This also allows the use of codecs adapted to the requirements of the processing of the transmitted data on the display system, e.g. when the 2.5D image is reprojected [15]. Additionally, this allows the system to profit easily from algorithmic improvements available in new video codecs, such as H.265 [16], as soon as GPGPU solutions for real-time compression at high resolutions are available.

4.2 Protocol for remote hybrid rendering

The purpose of the protocol for remote hybrid rendering is to define the communication between the visualization cluster and the local display system. i.e., the protocol for hybrid remote rendering connects the rendering stage to the display stage of the post-processing phase of the visualization pipeline. The data to be sent comprises viewing matrices, lighting configuration, desired image resolution and current animation step from client to server, which sends color and depth images in response.

Integrating the remote rendering facility with the application might enable further optimizations, as the application has more knowledge about which data is important. The application might chose to update the significant regions more often or at lower compression level with higher image fidelity. But as application independence is also a goal of this system, we do not take into account solutions that require tight coupling with the application, such as described above, e.g. IBRAC [13], or as implemented in Visapult [12].

Based on an assessment of the requirements listed above, we opted to implement RHR as extensions to the RFB protocol [17], as it allows for backward compatibility with regular VNC (Virtual Network Computing) clients by building on the extensible protocol implementation LibVncServer [3].

4.3 Implementation

4.3.1 Local display client

The client for remote hybrid rendering is implemented as a plug-in to OpenCOVER [5], the virtual reality renderer of the visualization system COVISE [9], and its data-parallel successor Vistle [2], which is currently being developed. It retrieves both color image and depth data from the server and renders these as an additional node in its scene graph. This achieves compositing of remote and local content. During each frame, the current values of the matrices describing the positions of the user's head and hand are sent to the server. In addition, the results of user interactions, e.g. new seed points for particle traces, are transmitted to the server.

While viewing the color image generated by an RHR server is possible with any VNC viewer, taking advantage of the compositing of local and remote data requires such a specially adapted VNC client.

4.3.2 Remote rendering server

For the RHR server, there are two implementations: one is realized as a plug-in for OpenCOVER. As such, it is compatible with COVISE and Vistle. The other implementation is a light-weight rendering module for Vistle, which uses the CPU for interactive ray casting.

Both server implementations can make use of a cluster of rendering resources by means of sort-last parallel rendering: a complete image is composited from renderings of all parts of decomposed data sets. This requires 2.5D image data (color and depth) for each partial image. The final image is obtained by selecting the color of each pixel from the partial image with the smallest corresponding depth value, i.e. that which is closest to the viewer. This step is executed by the IceT compositor framework [8], a library which provides highly efficient algorithms for combining images over MPI.

OpenCOVER uses a plug-in for this purpose, while compositing is an integral part of the Vistle ray caster. As the ray caster does not depend on GPU support, it allows scaling with the simulation even when there are no GPUs in the compute nodes.

The RHR servers provide a full implementation of a VNC server: every VNC client can connect to it and interact with the visualization with keyboard and mouse. For implementing this functionality, the library LibVNCServer [3] has been used.

For remote hybrid rendering, it has been augmented with the following features:

- Transmission of depth data (z-buffer) from server to client to enable compositing with image contributions rendered on the client
- Reception of 3D viewer and pointer positions sent by client
- Reception of interaction data sent by client

For compressing depth data the Snappy entropy compressor library is used [7]. For CPUs and CUDA capable GPUs, we implemented a method for lossy depth compression similar to DirectX texture compression, which operates orthogonal to the entropy encoding. The development of our own algorithm for depth compression was necessary, as we did not find a high bandwidth compression algorithm for image data with more than 8 bit precision per channel. Although VNC has mechanisms for sending color images, we added our own extension for sending JPEG compressed image tiles in order to be able to synchronize color and depth frames, which is necessary for correct compositing of local and remote images.

When rendering with OpenGL, image data has to be transferred from GPU to CPU memory before compositing. We employ two methods for copying the image data from GPU to CPU: one that relies purely on the OpenGL API call *glReadPixels*, and another one that employs CUDA for the transfer from GPU to CPU memory. Especially on gaming class hardware, resorting to CUDA provides better performance [6].

The CPU based data-parallel ray casting render module for Vistle is based on the ray tracing framework Embree [18], which makes use of the SIMD units of CPUs to reach interactive frame rates. The sole purpose of this render module is to provide the remote hybrid rendering service. Because of this, a rather light-weight implementation was possible, as most of the application logic resides in the RHR client.

4.4 Controlling RHR behavior

There are several ways of influencing remote hybrid rendering behavior.

- **Data Distribution:** The user can choose how to split the data between local and remote systems. On the one extreme, only interaction elements such as menus are rendered locally, while all the simulation data is kept on the remote system. If the local system is powerful enough, a large part of the static geometry can be rendered locally in order to provide lower interaction latency with these parts of the data. This is handled by the distributed visualization system.
- **Image Quality:** Image quality can be traded for bandwidth reduction and higher frame rates. Less precise lossy compression reduces bandwidth requirements. And reducing the resolution of the rendered image both reduces bandwidth requirements while simultaneously reducing the load on the image generation pipeline. Several different ways to compress depth and color images have been implemented.
- **Composition:** For high image fidelity, it is possible to combine the remote image with the local elements in a pose that corresponds to the viewing parameters used during remote image generation. The resulting behavior shows the limited interactivity of classic remote rendering. We did not implement that approach. Another possibility is to overlay the remote content with local imagery for the current viewing parameters. A third possibility is to warp or reproject the remote image based on the available depth data according to the current head position, thereby generating the lowest latency for head movements. However, this comes at the cost of holes in the warped surface and polygons that are shaded according to a previous viewer position. These two possibilities are implemented.

5 Tool evaluation

5.1 Data set and test configuration

For evaluation, the Institute of Fluid Mechanics and Hydraulic Machinery provided us with the results of a flow simulation in a water pump turbine. This unsteady simulation was conducted with OpenFOAM on 128 CPU cores on an unstructured grid consisting of 5.9 million hexahedron cells. Accordingly, the domain has been partitioned into 128 blocks. 273 time steps are available, a subset of which has been used.

From this dataset, the pressure has been used to extract an isosurface (green, value -1.86929416656) and the turbulence measure nuSgs has been used to colorize a cutting surface through the rotation axis of the turbine.

For the image compression tests, time step 0 of this data set has been used with the view shown in Figure 4. To make better use of the display area, a square view has been used: with the test data set this could avoid many “empty”/black pixels which do not show any significant data. A size of 1440x1440 pixels has been used, because this amounts to the same number of pixels as one Full HD image (1920x1080 pixels). The menu bar at the top of the image has been rendered locally.

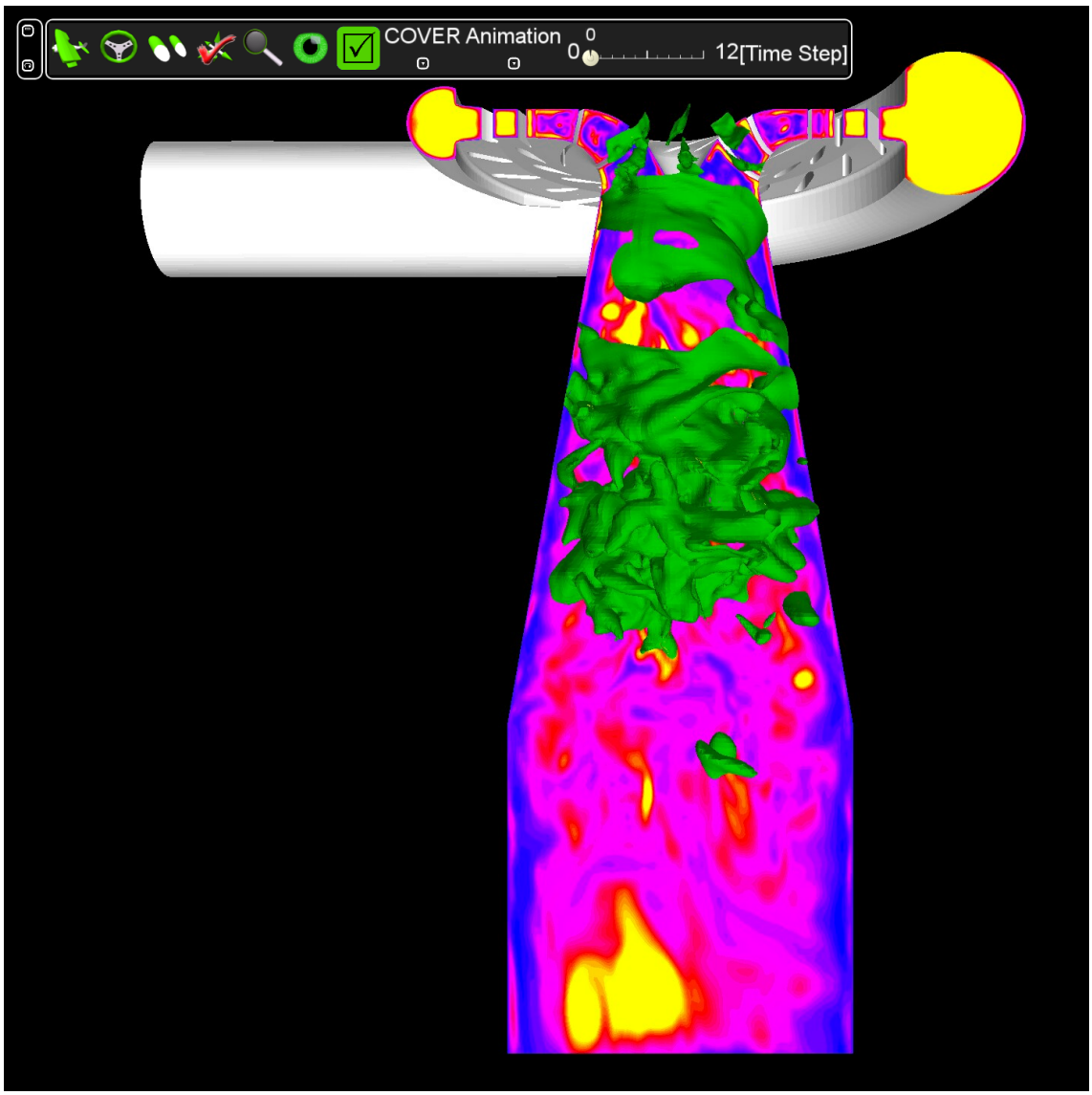

Figure 4: The view of the IHS pump turbine test case selected for image compression assessment.

The behavior of the system together with CPU based ray casting, a rendering method which for each pixel in the image traces rays from the viewer to the scene, but without taking into account shadowing, was studied. 16 nodes with 2 sockets holding 4 Sandy

Bridge cores each (Xeon E5-2643, 3.30GHz) have been used during the evaluation. The nodes are connected with a QDR InfiniBand network. For each time step, 8 blocks of the original partitioned data set are processed on one node. No data has been replicated across nodes. We used one MPI rank on each node. Rendering was done with a hybrid sort-first/sort-last algorithm: on each node, an image covering the whole screen area and containing the local parts of the data set has been rendered. This image has been subdivided into tiles of 64 by 64 pixels. These tiles are distributed to the available cores for rendering. After all tiles have been rendered, the IceT compositor assembled a complete 2.5D image on rank 0. Rank 0 then compresses the image and sends it to the client. Compression can happen in parallel with rendering images for other display surfaces requested by the client, e.g. for other eyes or other screens of a tiled display. Figure 5 shows the contributions to the final image from individual nodes in different colors together with the final composited image.

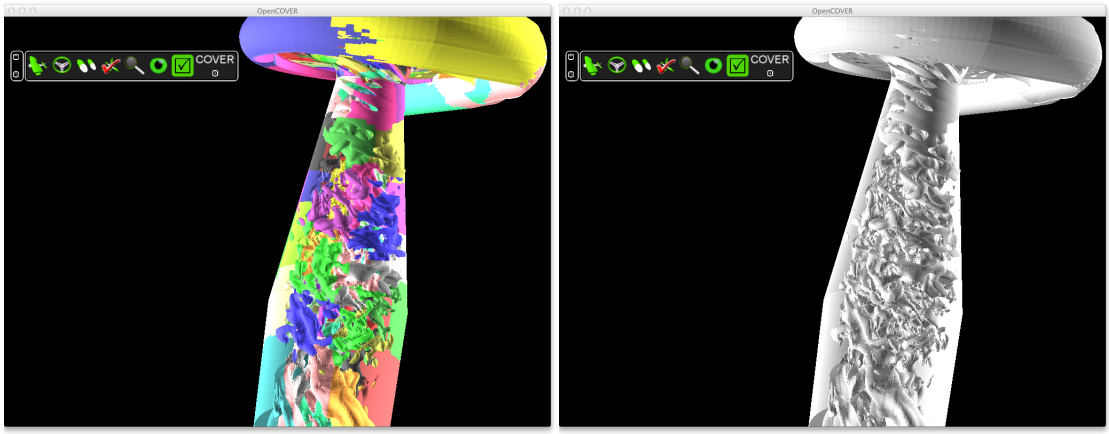

Figure 5: Contribution of nodes in different colors (left) and final composited image (right) of IHS pump turbine test case.

The display system was a Core i7 2600K (3.40GHz) with a Quadro K6000 GPU. It was connected with a 10 Gbit Ethernet link to the visualization cluster.

5.2 Quantitative evaluation

In order to allow for performance measurements, the tool has been instrumented to collect timing information, compression ratios and image quality metrics.

5.2.1 Image compression

The compression rates given in frames/s refer to frames sized 1440x1440 pixels, i.e. Full HD frames/s.

5.2.1.1 Color images

There are well-established means for compressing color images for remote rendering. We resorted to the approach used by VirtualGL [19]: compression with the JPEG still image codec. Like VirtualGL, we also used the SIMD-accelerated libjpeg-turbo [20]. We used 4:2:0 chrominance sub-sampling, i.e. the two chrominance values have been generated for each 2x2 pixel block of luminance data. We set the JPEG compression quality to 90. As we did not experience visible compression artifacts and as the amount of color data did not dominate the required bandwidth, we did not experiment with these settings. Compression happens at about 210 MPix/s (about 101 frames/s). The image size was reduced from 5.9 MB/frame to 0.18 MB/frame (3 %).

When bandwidth is not an issue, Snappy can provide somewhat lower latencies. In addition, this allows the remote image to be reproduced exactly. Compression occurs at a rate of about 420 MPix/s (about 203 frames/s). The image was compressed to 0.85 MB/frame (14.4%).

5.2.1.2 Depth images

Quantization of depth data happens at a rate of about 70 million pixels/s (about 34 frames/s) on a single core, whereas the reverse process takes place at a rate of about

120 million pixels/s (about 58 frames/s). There is some dependency on the input: more pixels at the far plane accelerate the process slightly. The measurements have been taken for the reference view. In its normal configuration, compression and decompression is distributed onto the available cores in tiles of size 256 x 256 pixels.

When using 24 bits for minimum and maximum per quantized depth tile of 4x4 pixels together with additional 3 bits/pixel, we reach a peak-signal to noise ratio (PSNR) of about 81.8 dB. One depth frame gets compressed from 5.9 MB to 1.48 MB (25.0%). The compression rate is independent of the image contents.

The observed PSNR is relatively high compared to codecs for color images. Hence, we did not notice any artifacts resulting from low depth fidelity based on the positions that pixels are reprojected to.

However, there is another source of error: based on the depth value of a pixel, its color value is chosen during compositing from either the remote color image or the local rendering. Hence, a pixel is either displayed correctly or in a completely unrelated color. As these artifacts can appear and disappear from frame to frame, they might be more noticeable than the PSNR suggests. Figure 6 illustrates these artifacts.

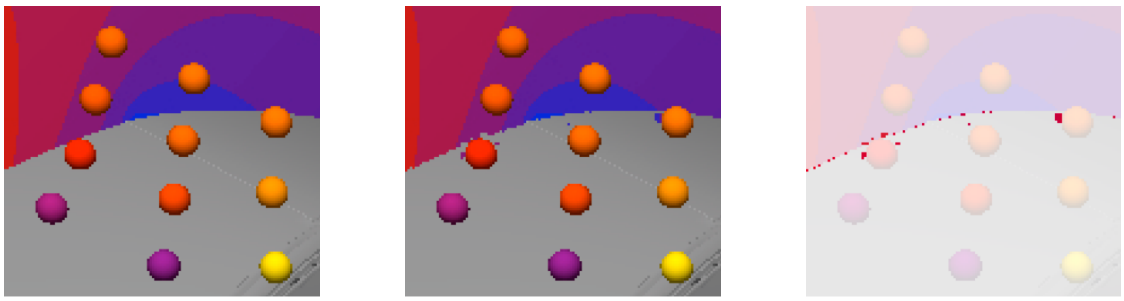

Figure 6: Depth buffer compression quality – left: original image, middle: with compressed depth, right: differences highlighted in red.

If depth quantization is followed by Snappy entropy encoding, then this occurs at a rate of about 1500 MPix/s (about 725 frames/s). This achieves a reduction to 0.25 MB/frame (4.2%). If depth is encoded with Snappy without preceding data reduction by quantization, then again a rate of about 420 MPix/s (about 203 frames/s) as for RGB images is achieved. The depth image is reduced to 1.09 MB/frame (18.8%).

5.2.2 Bandwidth requirements and latency

The bandwidth required for one frame is the sum of the compressed sizes of color and depth images. Please consult Table 1 for the values.

5.2.3 Latency

Latency varies with the codecs used for color and depth images. This is probably mostly due to the time consumed on rank 0 for compressing the image streams. Please refer to Table 1 for the details.

5.2.4 Frame rates

Remote frame rates are mostly limited by the performance of the CPU based ray caster. But the compression time also plays an important role, as compression is handled only by rank 0. We could generate about 7–11 updates per second for our reference view. Because of the hybrid sort-first/sort-last approach in the current implementation, this rate could be improved with a higher core count on a single node.

Local frame rates vary depending on whether reprojection is used. With reprojection, we achieved about 45 frames/s, without about 560 frames/s. Both results demonstrate that the goal of decoupling local updates from remote updates has been reached.

5.2.5 Summary

Table 1 summarizes the results for some combinations of compression settings. When “raw” data is transferred, then the current implementation of the system sends one

redundant byte for each depth and RGB pixel. Hence, the data for one image is higher than the uncompressed size mentioned above.

Color codec	Raw (4 byte/pix)	snappy	JPEG	JPEG	JPEG
Depth codec	Raw (4 byte/pix)	snappy	snappy	quant	quant+snappy
Latency (s)	0.141	0.125	0.106	0.173	0.161
Remote F/s	10.6	11.3	11.0	7.3	7.3
Color (MB/F)	7.91	0.85	0.18	0.18	0.18
Depth (MB/F)	7.91	1.09	1.09	1.48	0.25
Total (MB/F)	15.82	1.94	1.27	1.66	0.43
Total bandwidth (MB/s)	168	26.0	17.9	12.2	3.1

Table 1: Summary of measurements.

5.3 Reprojection artifacts

When just displaying the images that have been rendered remotely, the system is slow to react to view point changes, e.g. due to a new head position or when the object has been rotated. This is mitigated by reprojection. However, this induces artifacts when regions of the objects become exposed which have not been visible from the original vantage point: holes appear at those areas. This is illustrated in Figure 7. While the small hair lines can be covered by simply drawing points covering more than one screen pixel [1], this is not possible when large areas become visible, such as parts of the isosurface or areas previously obstructed by the isosurface.

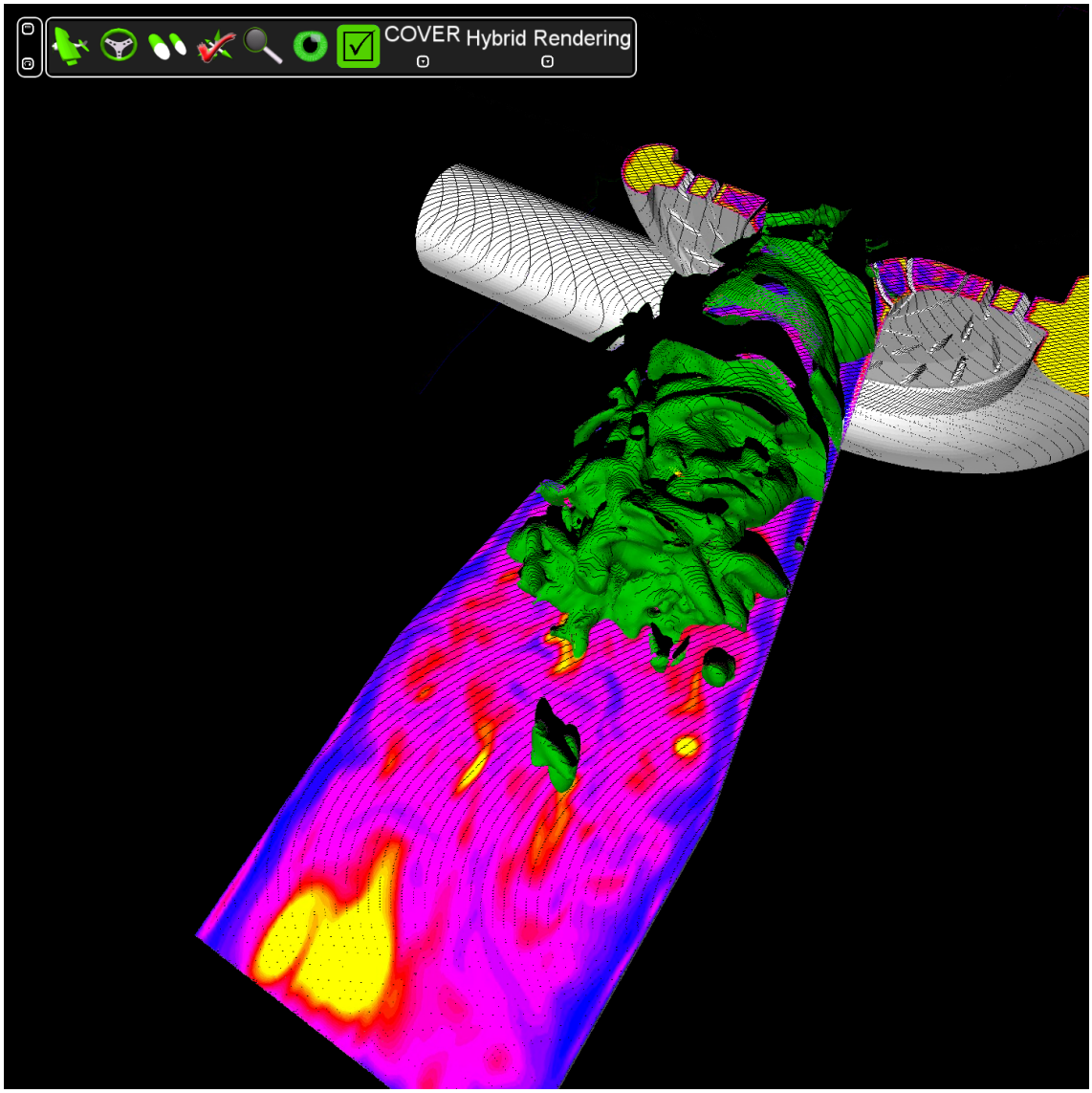

Figure 7: Rendering artifacts due to reprojection of 2.5D data for new view points.

Similar effects happen at the edges of display surfaces. With planar tiled displays, these holes could be filled in with pixels from neighboring displays. However in a system such as a CAVE, this is not possible: neighboring surfaces are usually oriented perpendicular to each other, hence the reprojection of the pixels from neighbor displays would only cover a very small area of the screen when the view point is changed slightly. Only increasing the size of the images could mitigate this effect, but this comes at a higher render and transfer cost.

5.4 Subjective experience

The system was demonstrated successfully at the HLRS booth (see Figure 8) at this year's Supercomputing conference in New Orleans: post-processing and rendering took place on a cluster at HLRS in Stuttgart, Germany. The results have been displayed on a stereo 3D display with 1400x1050 pixels (2.94 MPix per stereo frame) with head tracking. Interaction was smooth due to high local display update rates. Placing cutting surfaces and changing the isovalue was possible from within the virtual environment. After less than a second, updated images for the new parameters have become available, even though network round-trip times to Stuttgart were about 200 ms. A shared network connection to Stuttgart was used. The available bandwidth was about 10 MB/s. With full compression, display updates occurred at rates of about 10 frames/s.

Figure 8: HLRS booth at Supercomputing '14 in New Orleans with a remote hybrid rendering of the IHS pump turbine from Stuttgart, Germany, in stereo 3D.

We also used the system in our 5 wall CAVE: each side shows square 1200x1200 pixel images for each eye. Together with a head node, this sums up to 15 million pixels. We could achieve a remote frame rate of about 3/s. While it became clear that higher update rates are desirable, interaction with the data was still possible.

With full compression, the system is also usable across a broadband Internet connection (50 Mbit/s DSL) on a single screen system, e.g. a laptop computer.

These three use cases show that remote hybrid rendering is a promising approach: it is applicable to long-distance links, display systems with high pixel counts and multiple surfaces, and low bandwidth connections.

6 Discussion

6.1 Lessons learned

6.1.1 Pure remote rendering vs. remote hybrid rendering

Classic remote rendering couples a large server application on the remote system to a small display client on the local system. With remote hybrid rendering, this situation is reversed: all the application logic can reside on the local system, and the server application is only responsible for image generation according to updated view points from the client. The result is a lean server, which can be easily integrated with different applications, especially if the application already includes its own renderer. This benefits massively parallel systems, where the remote application is replicated across many nodes. This should make RHR very well suited for in-situ visualization.

6.1.2 Choice of RFB as base protocol

When designing the system, VNC's RFB protocol seemed to be a good choice as a base protocol for server/client communication. But during development, we replaced all parts of the VNC protocol with our own implementation, such that RFB merely served as a transport channel for our own protocol. Fortunately, this only incurs an overhead of one byte per request. But using direct socket communication instead of introducing LibVNCServer as another layer would have allowed for more control of TCP behavior. However, RFB still provides backward compatibility with regular VNC clients.

6.2 Open challenges

Based on the experience gained with implementing and using the current tool for remote hybrid rendering, we see the following gaps where the software could be improved:

- commonalities between the server-side implementations of RHR (OpenCOVER plug-in and Vistle CPU ray caster) should be identified to reduce code duplication and to provide a basis for integrating RHR support into other rendering software;
- while the lossy depth compression is a considerable improvement over only entropy based compression for low-bandwidth connections, bandwidth requirements and latency could benefit from further improvements of the compression algorithms for depth data, e.g. by exploiting inter-frame coherence.

7 References

[1] C. Wagner, M. Flatken, F. Chen, A. Gerndt, C. Hansen, and H. Hagen, “Interactive Hybrid Remote Rendering for Multi-pipe Powerwall Systems,” in *Virtuelle und Erweiterte Realität - 9. Workshop der GI-Fachgruppe VR/AR*, C. Geiger, J. Herder, and T. Vierjahn, Eds. Aachen: Shaker Verlag, 2012, pp. 155–166.

[2] Vistle GitHub repository [Online], Available https://github.com/vistle/vistle, [Accessed: 01 Mar. 2014].

[3] LibVNCServer/LibVNCClient [Online], Available: http://libvncserver.sourceforge.net, [Accessed: 20 Feb. 2013].

[4] M. Usoh, K. Arthur, M. Whitton, R. Bastos, A. Steed, M. Slater, and F. Brooks, “Walking > walking-in-place > flying, in virtual environments,” *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, Jul. 1999.

[5] D. Rantzau, K. Frank, U. Lang, D. Rainer, and U. Woessner, “COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data,” *2nd Workshop on Immersive Projection Technology*, 1998.

[6] F. Niebling, A. Kopecki, and M. U. Aumüller, “Integrated Simulation Workflows in Computer Aided Engineering on HPC Resources”, *International Conference on Parallel Computing,* 2011, Ghent.

[7] Snappy – a fast compressor/decompressor [Online], Available: https://code.google.com/p/snappy/, [Accessed: 23 Feb. 2013].

[8] K. Moreland, W. Kendall, T. Peterka, and J. Huang, “An image compositing solution at scale,” *presented at the 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC),* 2011, pp. 1–10.

[9] A. Wierse, U. Lang, and R. Rühle, “A system architecture for data-oriented visualization,” *Database Issues for Data Visualization*, vol. 871, no. 13, pp. 148–159, 1994.

[10] C. Cruz-Neira, D. J. Sandin, and T. A. DeFanti, “Surround-screen projection-based virtual reality: the design and implementation of the CAVE,” *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, p. 142, 1993.

[11] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, “A sorting classification of parallel rendering,” *Computer Graphics and Applications, IEEE*, vol. 14, no. 4, pp. 23–32, 1994.

[12] E. W. Bethel, B. Tierney, J. Leigh, D. Gunter, and S. Lau, “Using high-speed WANs and network data caches to enable remote and distributed visualization,” *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM*, Nov. 2000.

[13] I. Yoon and U. Neumann, “IBRAC: Image-Based Rendering Acceleration and Compression,” *Computer Graphics Forum*, Sep. 2000.

[14] B. Jeong, J. Leigh, A. Johnson, L. Renambot, M. Brown, R. Jagodic, S. Nam, and H. Hur, “Ultrascale Collaborative Visualization Using a Display-Rich Global Cyberinfrastructure,” *Computer Graphics and Applications, IEEE*, vol. 30, no. 3, pp. 71–83, 2010.

[15] D. Pajak, R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidel, “Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming,” *Computer Graphics Forum*, vol. 30, no. 2, pp. 415–424, 2011.

[16] G. J. Han, J. R. Ohm, W.-J. Han, W.-J. Han, and T. Wiegand, “Overview of the High Efficiency Video Coding (HEVC) Standard,” *Circuits and Systems for Video Technology, IEEE Transactions on*, no. 99, p. 1, 2012.

[17] T. Richardson, “"The RFB Protocol,” *realvnc.com*, 2010. [Online]. Available: http://www.realvnc.com/docs/rfbproto.pdf. [Accessed: 06-Sep.-2012].

[18] S. Woop, L. Feng, I. Wald, and C. Benthin, “Embree ray tracing kernels for CPUs and the Xeon Phi architecture”, *SIGGRAPH Talks*, p. 44, 2013.

[19] D. R. Commander, “VirtualGL: In Depth Background,” virtualgl.org. [Online]. Available: http://www.virtualgl.org/About/Background. [Accessed: Aug.-2011].

[20] D. R. Commander, “libjpeg-turbo Performance Study”, libjpeg-turbo.org. [Online]. Available: http://www.libjpeg-turbo.org/About/Performance. [Accessed: Nov.-2014].

[21] J. F. Hughes, S. K. Feiner, A. Van Dam, M. McGuire, D. F. Sklar, J. D. Foley, and K. Akeley, *Computer Graphics: Principles and Practice*, 3rd ed. Addison-Wesley, 2013, pp. 1–1268.