D5.3.4 – Remote hybrid rendering: revision of system and protocol definition for exascale systems

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 Exascale computing, software and simulation

Due date:	M24
Submission date:	30/09/2013
Project start date:	01/10/2011
Project duration:	36 months
Deliverable lead organisation	USTUTT
Version:	1.0
Status	Final
Author(s):	Martin Aumüller (USTUTT)
Reviewer(s)	Tobias Hilbrich (TUD), David Lecomber (ASL)

Dissemination level	
PU	*PU - Public*

Version History

Version	Date	Comments, Changes, Status	Authors, contributors, reviewers
0.1	02/09/2013	First version of the deliverable	Martin Aumüller (USTUTT)
0.2	16/09/2013	Consider reviewer comments	Martin Aumüller (USTUTT), Tobias Hilbrich (TUD), David Lecomber (ASL), Lorna Smith (UEDIN)
1.0	17/09/2013	Final version for submission	Martin Aumüller (USTUTT)

Table of Contents

Index of Figures

1 Executive Summary

Remote hybrid rendering (RHR) is developed 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. The display system forwards user input to the visualisation 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 before they are shown. This technique decouples interaction from rendering of large data and is able to cope with growing data set sizes as the amount of data transfer scales with the size of the output images.

Since D5.3.3, a prototype of RHR is available. This document describes its implementation and the algorithms developed for this prototype, especially for compressing depth images. Also, while implementing the prototype, some changes to the protocol draft in D5.3.2 for RHR became necessary. This document lists the necessary revisions. In addition, the performance of the prototype is examined.

Future versions of the RHR tool will be improved regarding bandwidth requirements and scalability.

2 Introduction

The structure of this document is as follows: Section 3 gives a short description of remote hybrid rendering. The following section describes its prototypical implementation. Section 5 presents the experience gained from implementing and testing the prototype. The next section summarizes the changes required to the protocol described in D5.3.2 and gives an updated list of RFB protocol extensions. Section 7 concludes by listing the work that is planned for the future.

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
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 colour and opacity)
VNC	Virtual Network Computing
WP	Work Package

3 Remote Hybrid Rendering

As transferring the results of a large-scale simulation to a local system for rendering is not viable [1], one often takes recourse to remote rendering: instead of post-processed data, rendered images are transmitted to the display. The highly lowered bandwidth and processing requirements of remote rendering allow for making efficient use of remote compute resources by a much larger user base.

Head-tracked immersive virtual environments, where the 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 [2]. These immersive visualisation environments provide more intuitive ways for specifying the location of regions of interest, cutting planes, seed points for particle traces, or reference points for iso surface extraction than desktop-based systems. We aim to enable users to experience exascale simulations in such immersive environments over the Internet.

To improve frame rate and reaction times, we decouple interaction from network latencies as far as possible, but still without requiring transferring huge data to the client. Only extracted features from simulation results are rendered either directly on the simulation host or on a remote visualisation cluster employing scalable methods. But "context information" – essentially static geometry, as e. g. turbine shapes, interaction cues for the parameters controlling the visualisation algorithms applied on the visualisation cluster and menus – is rendered locally, at a rate independent of the remote rendering. As both remotely and locally rendered images are composited for the final display, we call this technique "remote hybrid rendering". This compositing usually takes pixel depth into account, but it might also use opacity information.

Figure 1 illustrates how the image presented to the user results from local context information and remote simulation data.

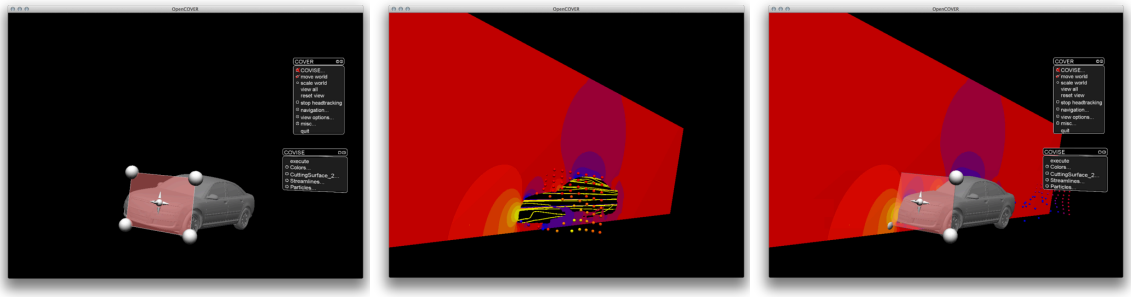

Figure 1: Local context information (left), remote simulation data (middle), fused image shown to the user (right).

4 First Prototype Implementation of Remote Hybrid Rendering

As part of D5.3.3 [10] a prototype demonstrating remote hybrid rendering (RHR) was implemented.

4.1 Implementation Details

Both, the client (local) as well as the server (remote) side of the prototype for remote hybrid rendering are implemented based on the same software: OpenCOVER [4], the virtual reality renderer of the visualisation system COVISE [5].

The server has to load the VncServer plug-in, while the client needs the plug-in VncClient. There are no other differences between server and client.

4.1.1 VncServer Plug-in

The VncServer plug-in for OpenCOVER provides 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 [8] 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 for enabling 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

These additional features can only be exploited by specially adapted VNC clients.

Colour image data is compressed using VNC's possibilities as implemented by LibVNCServer [8]. For compressing depth data, the methods described in section 4.3 are employed.

There are 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]. Additionally, this allows for off-loading parts of the compression algorithm to the GPU and reducing load on the PCI Express bus.

4.1.2 VncClient Plug-in

The VncClient plug-in for OpenCOVER is such a specially adapted VNC client. It retrieves both colour 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.

4.2 Changes since D5.3.3

Since D5.3.3 [10], a first prototype of the system for remote hybrid rendering is available. The prototype submitted for D5.3.3 has been improved regarding the following aspects:

- Server framebuffers are automatically resized to match client window size
- Compression algorithms for depth data have been improved and partially implemented on the GPU
- Integration with the prototype tool *Vistle* for massive parallel visualization that is currently being developed

Additionally, the prototype from D5.3.3 has also been instrumented for analysing run-time and compression performance.

The reported performance and experience is based on this enhanced prototype.

4.3 Depth Image Compression Algorithm

Most colour image codecs are not viable for depth image compression, as most of these handle only channels with 8 bit precision. It has been tried to adapt colour image codecs to depth compression, albeit with limited success [11]. Implementations of algorithms dedicated to depth image compression do not seem to be widely available. Hence, we implemented our own.

Our implementation is based on two orthogonal components: a lossy compression implemented on the GPU followed by a lossless entropy encoding on the CPU.

Due to the limit of one month on the implementation time, we did not consider developing a compression algorithm taking inter-frame coherence into account. But we expect huge improvements in compression ratio from such an approach.

4.3.1 GPU Depth Compression

A novel algorithm for depth data compression has been developed. Similar to S3TC texture compression [9], the algorithm operates independently on image patches consisting of 4x4 pixels. For each patch, we store two depth values, the minimum and maximum within the patch. For each pixel in a patch we store a weight for interpolating between the two stored depth values. In addition, for the common case where within a patch the background (maximum framebuffer) depth occurs, an optimization is implemented: if the maximum depth value is stored first, the highest interpolation weight is interpreted as background depth. Hence, the algorithm is able to resolve background and two depth planes.

The regular data pattern allows for an easy and efficient parallel implementation on GPUs, the lowered data rate reduces the transfer overhead from GPU to CPU.

Compression ratio and quality depend on the precision of the original image and the precisions of the stored depth values and interpolation weights.

4.3.2 CPU Depth Compression

Orthogonally to the lossy GPU based depth compression, entropy based compression is employed for lossless depth compression on the CPU. Because of its high compression speed, the snappy compressor library [7] has been selected for this purpose. The implementation of the lossy GPU depth compression tries to ensure good compression ratios by emitting a uniform pattern for patches consisting of only the background depth: the compressed data for such a patch has all bits set.

4.4 Adaptivity to Available Network Bandwidth and Latency

Adapting to network latency is not a necessity for RHR, as RHR naturally hides network latency by decoupling interaction from receiving the remotely generated images.

Adapting to limited network bandwidth is easily possible by rendering the remote image with decreased resolution. We tested our implementation with such a set up successfully, but we do not have mechanisms for dynamic resolution changes in place.

5 First Experience with Remote Hybrid Rendering

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

5.1 Choice of RFB as Base Protocol

Due to limited commonalities between VNC and remote hybrid rendering, the choice of RFB – the protocol employed by VNC – as base protocol did not allow for much protocol reuse. It still provides backward compatibility with regular VNC clients. Also, RFB has the advantage of inducing low overhead and providing a mechanism for protocol extensions. In addition, LibVNCServer [8] provided an extensible implementation of the RFB protocol on which we could build as well as an established means for transferring colour data.

5.2 Implementation of Protocol Draft

During the implementation of the protocol draft, we noticed that remote hybrid rendering is very well suited for the use case of in-situ visualization: the largest part of the application logic of the display program resides on the computer receiving user input, and the remote application may be a rather dumb render server that receives only updated view points from the client. This allows for easy integration with simulation codes that already have their own renderer. Also, while we implemented the prototype based on the same local and remote application, this is not a necessity.

5.3 Performance of Prototype

5.3.1 Bandwidth

With LibVNCServer's default of using JPEG compression for colour images, transmitting one Full HD frame (1920x1080 pixels) required about 1 MB for both colour and depth images.

5.3.2 Frame Rate

Framerate has been measured for Full HD frames when transmitting from a system with a Quadro 5800 GPU to another over a QDR Infiniband network. The images received from the remote server have been updated at a rate of about 20 Hz, while the local renderer updated its contents at a rate of ca. 50 Hz. This shows that the goal of decoupling local and remote update rates has been achieved.

The low frame rate of 20 Hz is not due to the required bandwidth of about 20 MB/s, but mostly due to the slow depth buffer read-back performance of about 40 Hz on the Quadro 5800.

5.3.3 Latency

In the setting of 5.3.2, latency has been measured to increasy by 0.1 s for a Full HD frame.

5.3.4 Compression Ratio and Quality

The implementation of the prototype is based on LibVNCServer [9]. Colour image transfer relies solely on what is provided by LibVNCServer. Support for depth was implemented as a VNC extension.

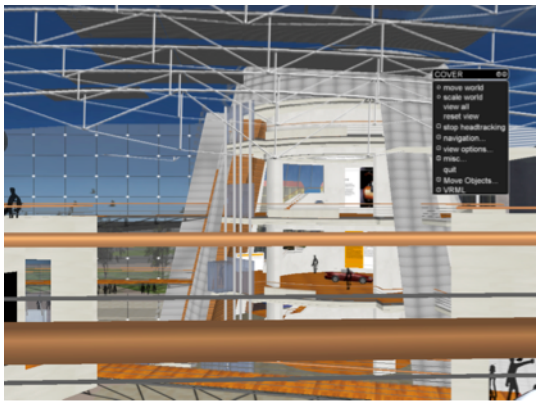

Figure 2: Reference image for depth buffer compression quality assessment.

bits/pixel		2	3	4	6	8
24 bits min/max	compressed size	20.8%	25.0%	29.1%	37.4%	45.8%
	PSNR (dB)	67.7	69.4	77.6	86.6	97.4

Table 1: Compression ratio and quality for lossy GPU based depth compression for the image in Figure 2.

Table 1 shows the compression ratios and qualities for the 24 bit depth image corresponding to the colour image shown in Figure 2. The peak-signal to noise ratio (PSNR) is relatively high compared to codecs for colour images. But the visual errors resulting from wrong reconstructed depth values differ from the errors in colour image compression: based on the depth value of a pixel, its colour value is chosen from either the remote colour image or the local rendering. Hence, a pixel is either displayed correctly or in a completely unrelated colour. As these artefacts can appear and disappear from frame to frame, they might be more noticeable as the PSNR suggests. Figure 3 illustrates these artefacts.

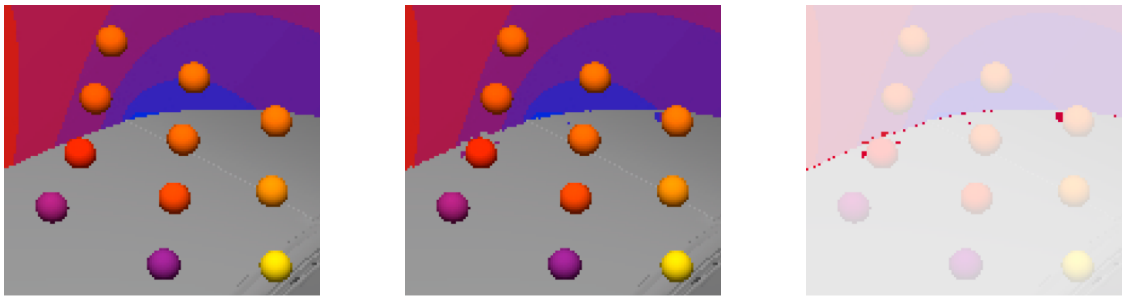

Figure 3: Depth buffer compression quality – left: original image, middle: with compressed depth, right: differences highlighted in red.

5.3.5 Framebuffer Read-back Performance

A crucial component of the remote hybrid rendering system is retrieving the image data from the GPU framebuffer. As Table 2 shows, depending on the model of the GPU, achieving good read-back performance requires different methods.

Whereas OpenGL colour read-back (*RGBA (GL)*) is fast on Quadro GPUs, it is slow on GeForce GPUs. However, when transferring the OpenGL colour framebuffer to CUDA via CUDA-OpenGL-interoperability and using `cudaMemcpy` to retrieve the image (*RGBA (CUDA)*), the GeForce card achieves best performance, while the older Quadro model falls behind.

OpenGL depth read-back on Quadro GPUs is also fast, and also slow on GeForce GPUs (*Depth (GL)*). However, transferring the depth buffer from OpenGL to CUDA (*Depth (CUDA)*) is also slow, such that this yields bad performance on all GPUs.

Performing the lossy depth compression on the GPU (*Depth (CUDA compressed)*) entails a slightly reduced read-back rate, however a subsequent entropy compression step has to work on considerably less data.

Depth read-back is uniformly fastest when the z-buffer is copied to the colour framebuffer using the `NV_copy_depth_to_color` OpenGL extension and then read back as RGB data (*Depth (as colour, GL), (as colour, CUDA), (col. CUDA comp.)*).

From a performance point of view, the preferable approach in most cases is to copy the depth to the colour buffer and employ lossy GPU compression before transferring the data to the CPU.

MPix/s	Quadro 5800	Quadro 6000	Quadro K5000	GeForce GTX 680
RGBA (GL)	470	530	470	150
RGBA (CUDA)	131	570	560	710
Depth (GL)	380	460	395	127
Depth (CUDA)	112	202	159	97
Depth (CUDA compressed)	86	187	149	92
Depth (as colour, GL)	560	740	690	280
Depth (as colour, CUDA)	124	530	550	670
Depth (col., CUDA comp.)	93	420	443	618

Table 2: Framebuffer read-back performance in million pixels per second.

6 Revised Protocol

6.1 Summary of Changes to Protocol Drafted in D5.3.2

While implementing the remote hybrid rendering server and client, it became apparent that the protocol drafted in D5.3.2 [1] has to be revised:

- It is not necessary to send input data, such as from multi-touch or 6DOF devices, from client to server. Instead, input events have to be processed by the local client application.
- Instead, the state of the local application has to be synchronized with the remote application, e. g. matrices describing the viewer's position and the transformation of objects have to be transmitted to the remote server, such that locally rendered images and remote images can be matched during compositing.
- Some operations have to be carried out collectively on the remote and local renderer, e. g. the scene bounding sphere has to comprise bounding spheres for both local and remote data, this requires support from the protocol.
- Application state has to be synchronized between client and server applications, this requires application-specific protocol extensions. E. g., the rendering and lighting modes configured on the client also have to be applied on the server.
- Some actions have to be carried out cooperatively between client and server. E.g. moving a cutting plane to another position requires sending the updated parameters from client to server, where the application has to extract the corresponding data and to update the rendered image. This requires protocol support for sending the data describing the interaction capabilities from server to client as well as for updating parameters from client to server.

6.2 Revised List of RFB Protocol Extensions

Based on the requirements established in D5.3.2 and the implementation experience, we revised the extensions to the RFB protocol that are required for remote hybrid rendering. Some of them are already implemented in the RHR prototype.

6.2.1 Multiple Display Surfaces

Support for handling multiple display surfaces can be added by providing a coordinate mapping for each surface to the available 65536x65536 pixel coordinate space. The protocol has to be augmented for establishing this mapping.

6.2.2 3D Stereo Rendering

Additional image codecs for supplying images for both the left and right eye for one display surface will have to be implemented. If it is not necessary to exploit the coherence between the images for the left and right eye for better compression efficiency, then it is sufficient to provide different coordinate mappings for both eyes.

6.2.3 Frame Barrier

In order to advance the display to the next rendered frame synchronously on all display surfaces and for both the left and right eye, the server has to send an event when a frame is fully transmitted. This should include a frame counter and a time stamp.

6.2.4 Server Controlled Framebuffer Updates

In order to minimise latency, rendered frames should be sent to the client as soon as they are available. The client has to be able to request continuous framebuffer updates from the server, i. e. the communication flow has to become more asynchronous than with standard RFB. For that purpose, TigerVNC [12] already has an extension containing an `EnableContinuousUpdates` message, this will be reused.

6.2.5 Encodings for Depth and Transparency Data

The available image encodings shall be augmented by appropriate codecs for transparency and depth data, in order to composite context information display with the visualisation image on the client.

6.2.6 Efficient Image Codecs

Compression quality can be enhanced by employing efficient image and video codecs. The available image codecs shall be augmented by video streaming codecs for which GPGPU implementations are available.

6.2.7 Image Data Back-Channel

In order to be able to generate images where the context information is correctly composed with the visualisation data, the rendering of the context information together with corresponding depth and opacity data has to be made available to all visualisation nodes. The *FrameBufferUpdate* message, which is normally sent from server to client only, will be used for that purpose.

7 Future Work

Future versions of the software will be improved regarding the following aspects:

- Improved compression algorithms for depth data
- Selective updates for depth data
- Interoperability with server-side parallel rendering as described in D5.3.1 [3]
- Reprojection of 2.5D data according to current view point
- Better synchronization of client and server state

While the first prototype of remote hybrid rendering has been implemented within COVISE and OpenCOVER, the light-weight nature of the server side makes it easy to interoperate with other applications: the main requirements are that the remote application be able to accept camera parameters (position, orientation, field of view and off-center projection) and to generate 2.5D images (colour and depth). A simulation code with an integrated ray-tracer, such as HemeLB [13], meets these requirements. But as *OpenFOAM* [14] is our main test-bed for the development of the massively parallel visualisation tool *Vistle*, we aim to demonstrate the applicability of RHR by coupling to a running *OpenFOAM* simulation and observing it online.

8 References

[1] M. Aumüller: CRESTA D5.3.2: Remote hybrid rendering: protocol definition for exascale systems, 2012.

[2] 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.

[3] F. Niebling, J. Hetherington, and A. Basermann, “D5.3.1 – Remote hybrid rendering: analysis and system definition for exascale systems,” CRESTA, Mar. 2012.

[4] U. Woessner, D. Rantzau, and D. Rainer, “Interactive Simulation Steering in VR and Handling large Datasets,” *IEEE Virtual Environments*, Jan. 1998.

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

[9] K. I. Iourcha, K. S. Nayak, and Z. Hong, “Fixed-rate block-based image compression with inferred pixel values,” 2003.

[10] M. Aumüller: CRESTA D5.3.3: Remote hybrid rendering: first prototype tool, 2013.

[11] F. Pece, J. Kautz, and T. Weyrich, *Adapting Standard Video Codecs for Depth Streaming*. The Eurographics Association, 2011, pp. 59–66.

[12] “The RFB Protocol,” *tigervnc.org*. [Online]. Available: http://www.tigervnc.org/cgi-bin/rfbproto. [Accessed: 15-Oct-2012].

[13] M. D. Mazzeo and P. V. Coveney, “HemeLB: A high performance parallel lattice-Boltzmann code for large scale fluid flow in complex geometries,” Computer Physics Communications, vol. 178, no. 12, pp. 894–914, Jun. 2008.

[14] H. G. Weller, G. Tabor, H. Jasak, and C. Fureby, “A tensorial approach to computational continuum mechanics using object-oriented techniques,” Computers in physics, 1998.