Eurographics Symposium on Parallel Graphics and Visualization (2019)
H. Childs, S. Frey (Editors)

Hybrid Remote Visualization in Immersive Virtual Environments with Vistle

Martin Aumüller[1]

[1]HLRS, University of Stuttgart, Germany

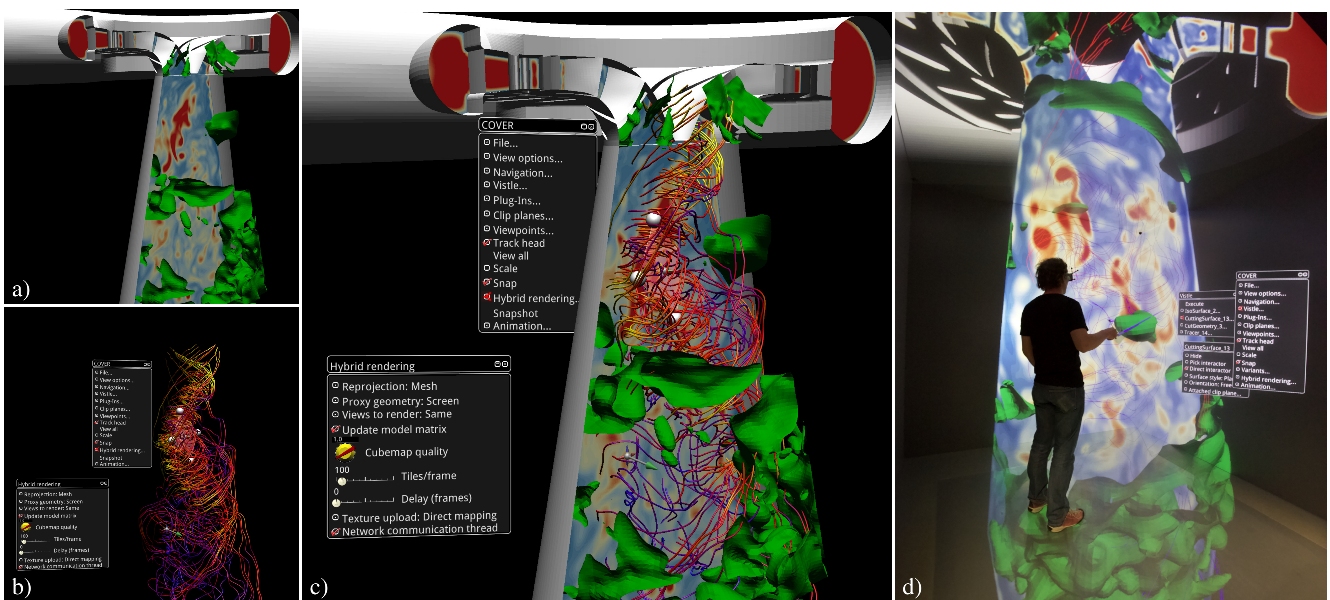

Figure 1: *Hybrid remote visualization of a pump turbine (courtesy of IHS, University of Stuttgart):* ***a)*** *surface components of remotely rendered simulation data,* ***b)*** *local menus, user interaction elements and lower-dimensional simulation data,* ***c)*** *composited hybrid image,* ***d)*** *the same hybrid remote visualization in the CAVE at HLRS.*

Abstract

Because of the spatial separation of high performance compute resources and immersive visualization systems, their combined use requires remote visualization. Remote rendering incurs increased latency from user interaction to display. For immersive virtual environments, this latency is a bigger problem than for desktop visualization. With hybrid remote visualization we enable the exploration of large-scale remote data sets from immersive virtual environments. This is based on three factors: When appropriate, we enable the local rendering of remote objects. We decouple local interaction from remote rendering as far as possible by depth compositing of remote and local images at a rate independent from remote rendering. Finally, we try to hide this latency by reprojecting 2.5D images for changed viewer positions. In this paper we describe the integration of hybrid remote rendering into the data-parallel visualization system Vistle as well its extension to a distributed system. Thereby arbitrary combinations of object-based and image-based remote visualization become possible.

Categories and Subject Descriptors (according to ACM CCS): Distributed Systems [C.2.4]: Distributed applications—I.3.2 [Computer Graphics]: Graphics Systems—Remote systems I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality I.4.2 [Image Processing and Computer Vision]: Compression (Coding)—Approximate methods

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

DOI: 10.2312/pgv.20191113

delivered by
EUROGRAPHICS DIGITAL LIBRARY
www.eg.org
diglib.eg.org

1. Introduction

Immersive virtual environments with head tracking such as CAVEs [CNSD93], constantly update the display according to the viewer's changing head position. So they provide more intuitive ways for specifying the location of regions of interest, cutting planes, seed points for particle traces, or reference points for isosurface extraction than desktop-based systems. This makes them a powerful tool for gaining insight into complex 3-dimensional problems. These immersive visualization environments require high frame rates and low reaction latencies to achieve a high sensation of presence and to avoid motion sickness [UAW*99] as well as to enable motion parallax for discovering 3D object relations.

Complexity grows with problem size, hence visualization in virtual environments becomes even more useful for large-scale data and simulations. At the same time, interactive visualization of these data faces even more challenges. Very often, only large parallel systems can cope with the amounts of data to be analyzed. Remote rendering (the blue pathway in the distributed visualization pipeline of figure 2) is an established method for visualizing data residing on such remote computing resources that cannot be made available locally. However, it comes with the cost of increased latency from user input to display output. This is a significant obstacle to navigation and interaction with the visualization.

Combining the blue and black pathway in the distributed pipeline was presented as a solution to these challenges by Wagner et al. [WFC*12]. Typically, simulation results are rendered on the remote system while user interface elements and static context geometry is rendered locally. On the local display system, this requires compositing of remotely and locally rendered images. They call the process of compositing locally rendered images with warped RGB-D maps from remote systems "hybrid remote rendering" (HRR).

In this work, we describe the integration of HRR into our parallel system Vistle [Aum15] as well as its extension to a fully distributed visualization system. This enables the creation of arbitrary hybrid object- and image-based remote visualizations by building workflows employing any combination of paths along the arrows in figure 2: without restriction, the modules of a visualization workflow can be assigned to different compute clusters, and pipeline objects as well as image streams can be routed freely between those clusters. The rates of local and remote rendering are decoupled, and the resulting images composited. This empowers the user to build a workflow which optimally employs local and remote resources for a high-fidelity low-latency visualization. We call this hybrid remote visualization.

Our main contributions in this work are

- a fully distributed data-parallel visualization system with the ability to create arbitrary hybrids of local and image- and object-based remote visualization,
- the adaption of HRR to output systems with non-flat display surfaces driven by a cluster,
- as well as a high-bandwidth compression algorithm for depth images.

In the next section 2 we describe the main components of our system together with the additions for this work. In section 3 follows a description of our implementation of HRR. We evaluate and discuss our system in section 4. We put our system into context of other related research in section 5, before we conclude with an outlook on future work.

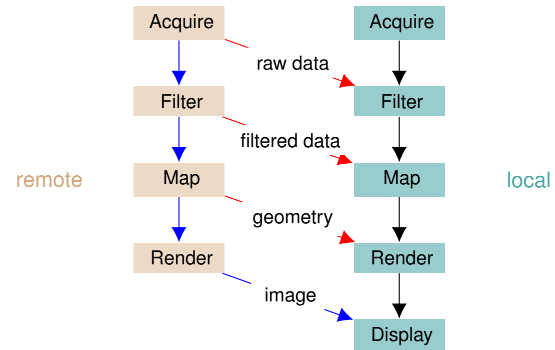

Figure 2: *Data flow in a hybrid remote visualization pipeline: for different parts of the data set, the jump from the remote (beige) to the local (turquoise) system can occur after any stage, and any combination of pathways is possible. The user is presented with a composite of remotely and locally rendered images. The black path is local rendering, the blue arrows show image based remote rendering, and the red arrows enable object-based remote visualization.*

2. System Description

In this section we describe the main components of our system. We start with an overview of the architecture of the parallel visualization system Vistle. Then we describe two important modules: the newly developed hybrid sort-last/sort-first parallel ray caster DisCOVERay, as well as the virtual environment renderer OpenCOVER, which has been adapted to Vistle and extended for hybrid remote rendering with a HRR client plug-in.

2.1. Vistle

Vistle [Aum15] is a scalable distributed implementation of the visualization pipeline. Workflows can be configured with a graphical user interface as shown in figure 3. A hybrid image-/object-based remote visualization is created by placing DisCOVERay (cf. subsection 2.2) and other modules on a remote cluster (colored in beige) and connect them to the renderer COVER (cf. subsection 2.3) running on the local system (colored in turquoise). Already Vistle's predecessor COVISE [WLR94] allowed to create pipelines spanning several systems. This ability has been added to Vistle for this work. Just as COVISE, Vistle targets especially interactive visualization in immersive virtual environments. But the tight integration of remote rendering is a new addition.

The second key-benefit over COVISE is the exploitation of data-parallelism. Modules are realized as MPI processes on a cluster operating on data partitions distributed across nodes. Within a node, multiple threads are used as an additional level of parallelism when processing a single partition. Within compute nodes, different modules communicate via shared memory. As an alternative,

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

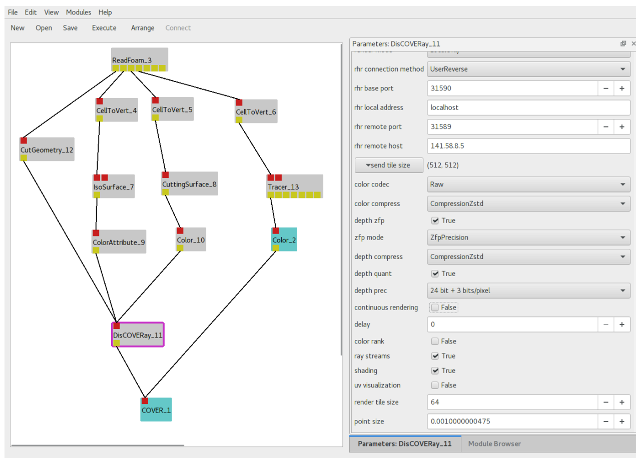

Figure 3: *User interface for configuring a distributed Vistle workflow: modules running on the local system are shown in turquoise, those on the remote cluster in beige. Data flow is from top to bottom.*

Vistle can be compiled for a single-process environment, where each module runs in a separate thread inside a single MPI process. However, this requires a multi-threaded MPI implementation (`MPI_THREAD_MULTIPLE`). TCP is used for communication between clusters. As operation policies of HPC systems often require that communication be funneled through a single node, Vistle only supports that mode for inter-cluster communication to date.

Arbitrary connections in the visualization pipeline can cross the boundary between clusters: for this work, we implemented streaming of both pipeline data objects as well as images. Message payloads transporting those streams can be compressed with entropy based algorithms, either LZ4 or Zstandard. Both of them achieve high compression bandwidth, but LZ4's bandwidth is a little higher at the cost of lower compression. In addition, lossy data reduction of floating point values is possible with the zfp [Lin14] floating point compressor. As zfp exploits spatial coherency, this works best with data on 2- or 3-dimensional structured grids.

Vistle and the other components described here are portable to Linux, Windows and macOS and are available on GitHub.

2.2. DisCOVERay

DisCOVERay is a CPU based data-parallel ray casting render module for Vistle. It builds on the ray tracing framework Embree by Wald et al. [WWB*14], which makes use of the SIMD units of CPUs to reach interactive frame rates. It has to be examined, whether OSPray [WJA*16], an additional layer on top of Embree focussing on scientific visualization, can simplify implementing DisCOVERay's functionality. As each node in the cluster only possesses its own partition of the data set, sort-last [MCEF94] is used for parallelizing the render process across nodes. As it is sufficient to generate images of the same quality as the local renderer COVER (cf. subsection 2.3), only primary rays have to be considered. It would also be costly to trace secondary rays in the distributed scene database. Within a node, the render load is distributed among the local CPU cores by partitioning image space into tiles. Simple load balancing is achieved by partitioning into more tiles than CPU cores.

The Equalizer framework by Eilemann et al. [EMP09] was considered for compositing a complete color and depth image (2.5D) after the parallel rendering. However, the IceT compositor framework [MKPH11] was preferred, because of its ability to use MPI as a communication layer and its promise to integrate more easily with the existing system. As IceT only allows for one image being composited at a time, the rendering of multiple views (e. g. for stereoscopic 3D or for the different sides of CAVEs) is serialized. As shown in [LMJC16], it might be worthwhile to explore other configurations, e. g. combining all views into one image which is composited in a single operation. However, this has to be examined further since, in our context, additional overlap is achieved, as previous views are being encoded and transferred while IceT is busy compositing consecutive views. On the master node of the remote system, IceT's result is divided into tiles, and color and depth components are compressed independently and transferred to the head-node of the local system as soon as ready. After all tiles for all views have been sent, a final mark is transmitted to indicate the completion of a frame. The primary motivation for building the ray caster was that it does not depend on GPU support, such that it allows to scale with the simulation even when there are no GPUs in the compute nodes. Further, rendering on the CPU has the advantage that no image transfer from GPU to CPU is necessary before compositing on the remote cluster.

The purpose of this render module is to provide the hybrid remote rendering service. Because of this, a rather light-weight implementation was possible, as most of the application logic resides in the HRR client. Only support for cycling through timesteps and switching semantic parts of the geometry is integrated into the remote renderer. The framework for providing the remote rendering service is organized into a library. Building on this, there is also a GPU renderer implemented with OpenSceneGraph. The library also makes it easy to use existing image generation software as a simple HRR server: it is sufficient to provide a combined RGB/depth image as a response to a request with a viewport and model/view and projection matrices.

2.3. OpenCOVER

OpenCOVER [RFL*98] was built as the render module of COVISE [WLR94] for immersive virtual environments. It is based on the OpenSceneGraph [WQ10] scene graph library, which employs OpenGL for rendering. It supports desktop, head-mounted displays (HTC Vive and Oculus Rift), as well as projection-based immersive virtual environments. Projection-based environments consisting of several output surfaces, such as CAVEs, nowadays are usually driven by a small cluster, where each node possesses one or a few GPUs connected to the display devices. If OpenCOVER is used in such a setting, one node is configured as the master node, which receives all user input and distributes it to the slave nodes. The data to be rendered is replicated across all nodes. As all nodes see the same input, the application can be mirrored, running in lock-step on all cluster nodes.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

For the work presented here, OpenCOVER has been refactored so that the connection to visualization systems is established with a plug-in, allowing for also developing a plug-in for Vistle. All preexisting interaction methods for controlling e. g. cutting surfaces, seed points for particle tracing or isovalues from virtual environments can be used together with Vistle. Relying on the established OpenCOVER VR framework allows for combining all available plug-ins with data from Vistle. One important use case is to provide context for large-scale simulation results with interactive 3D environments, e. g. based on VRML, or point-clouds from 3D laser scans.

2.4. HRR Client

The client application for HRR is implemented as another plug-in to OpenCOVER, which establishes a dedicated TCP connection. This TCP connection is initiated between OpenCOVER's head node to DisCOVERay, when the output of DisCOVERay is connected to the input of OpenCOVER from the workflow editor. This means that image tiles received from remote have to be redistributed within the local CAVE cluster. Together with uploading the received images to the GPU this redistribution is the major source of frame rate jitter during hybrid remote rendering. In the current implementation, the HRR client is implemented with OpenSceneGraph and a couple of shaders for reprojection. This takes care of compositing the remote image with local content. As OpenSceneGraph does not provide a mechanism for asynchronous texture upload from CPU to GPU memory, this problem cannot be easily overcome.

3. Hybrid Remote Rendering in Vistle

3.1. Overview

To improve frame rate and reaction times, we try to decouple interaction from network latencies as far as possible, as recommended by Taylor et al. [TJV*10], but still without requiring transferring huge data to the client. Only features extracted from simulation results are rendered either directly on the simulation host or on a remote visualization cluster. But interaction cues for the parameters controlling the visualization algorithms applied on the visualization cluster, 3D interaction menus and possibly "context information" such as essentially static geometry, as e. g. turbine shapes, are rendered locally, at a rate independent of the remote rendering. As both remotely and locally rendered images are composited on the display system, we call this technique "hybrid remote rendering". Just as in sort-last [MCEF94] parallel rendering, this compositing usually takes pixel depth into account, but it might also use opacity information.

The left part of figure 1 illustrates HRR with a visualization of the simulated water flow in a Francis pump turbine. 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 an isosurface of the pressure as well as a plane colorized according to turbulence (top left). The casing, blades and vanes are also rendered on the remote system, as they have been extracted from the simulation data. The local system renders the menu and interaction elements, e. g. for moving the cutting plane or positioning the seed points for the streamlines. But also the streamlines are rendered locally, as only surface geometry is well suited for reprojection (bottom left).

All the interactive features of the visualization system are available even though parts of the rendering are delegated to a remote system. For instance, 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.

3.2. HRR Protocol

The HRR protocol was originally layered on top of VNC by means of the library LibVNCServer, as we wanted to take advantage of its event distribution and image compression features. But as HRR requires synchronization between RGB and depth images, all the machinery for encoding images had to be implemented within the HRR framework. So it was easy to remove the dependency on VNC, which allowed for employing Vistle's native message format together with built-in compression facilities. The HRR protocol supports the following functionality:

- Transmission of depth data (z-buffer) from server to client for enabling compositing with image contributions rendered on the client
- Transmission of color data synchronized with depth image
- Reception of model/view and projection matrices sent by client
- Reception of configuration of light sources in the scene
- Controlling animated sequences and switching of semantic parts of the data set

3.3. Compression

3.3.1. RGB Image Compression

Interactive rendering requires high throughput. Depending on image sizes and network connectivity, different compression methods are useful. Image transmission can take advantage of the available lossless message compression methods. Alternatively, we use the TurboJPEG library for compression of color images. Just as when used in VirtualGL for sending 3D rendered images, we also subdivide the image into tiles of 256×256 pixels, which allows to use several threads to encode the image in parallel.

3.3.2. Compression of Depth Buffer Data

High-throughput compression of depth buffer data is not yet solved in a satisfactory manner. Most color image codecs are not viable for depth image compression, as most of these handle only channels with 8 or 10 bit precision. It has been tried to adapt color image codecs to depth compression though, albeit with limited success [PKW11]. Implementations of algorithms dedicated to depth image compression do not seem to be widely available. Hence we implemented one based on two orthogonal components: a lossy compression step ("Quant") followed by a lossless entropy encoding.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Lossy Compression: Depth Quantization Similar to DirectX/S3TC texture compression [INH03], the lossy step of the algorithm operates independently on image patches consisting of 4×4 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 fixed number of bits as a weight for interpolating between the two stored depth values. This quantizes the depth data within a patch to just a few possible values between minimum and maximum. 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 of the patch is stored first, the highest interpolation weight is interpreted as background depth, thereby decreasing the number of representable discrete values between minimum and maximum by one. This is illustrated in figure 5 for a depth buffer with 8-bit precision and just 2 bits interpolation weight per pixel: the source patch, the compressed data, and the restored patch are shown from left to right. Hence, the algorithm is able to resolve background and two depth planes within a single patch. An important property of the algorithm is that pixels on the far plane are always encoded without error. The regular data pattern with a fixed number of output bits per input patch allows for an easy and efficient parallel implementation on GPUs. The lowered data rate reduces the transfer overhead from GPU to CPU.

Figure 4: *Left: Reprojection of a sphere showing "comet tail" artifacts caused by rubber banding to pixels on the far plane, which have not been correctly eliminated from the image. Middle/right: Depth map used for evaluating lossy depth compression and corresponding color image of a jet break-up in a non-Newtonian fluid in air (courtesy of ITLR, University of Stuttgart).*

Compression ratio and quality depend on the precision of the original image and the precisions of the stored depth values and interpolation weights. Initially we evaluated two different configurations: storing minimum and maximum depths with 24 bits and 3 bits interpolation weights/pixel, or 16 bits for extrema together with 4 bit weights. But as already first visual tests showed that the 16 bit variant was inferior for our use case, it was dropped. Both configurations use 96 bits per 4×4 pixel patch. This reduces the data size to 25%.

Lossless Entropy Compression Orthogonal to the lossy depth compression step, entropy based compression is employed for lossless depth compression on the CPU. The implementation of the lossy depth compression tries to ensure good entropy compression ratios by emitting a uniform pattern for patches consisting of only the background depth: all bits are set in the compressed data for such a patch.

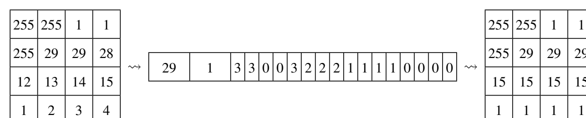

Figure 5: *Lossy depth compression. Left: 4×4 pixel patch with 8 bit precision with far-depth (255). Middle: encoded as maximum depth (except far) first, minimum depth, and then 2 bits/pixel used as interpolation weight, highest value reserved for far. Right: decoded depth image.*

3.3.3. Evaluation of Lossy Depth Compression

We compare the performance of our algorithm against zfp in different settings on a Sandy Bridge running at 2.6GHz. zfp was operated in three different modes: "precision" was set to 16, "accuracy" to $1/1024$, zfp's "fixed" rate was set to 6, so that it achieves almost the same compression ratio as our algorithm "Quant". Except for when operating at fixed rate, zfp achieves somewhat better peak-signal to noise ratio (PSNR) as Quant, but with better compression. However, our algorithm is faster at decompression and significantly so at compression. Our measurements (see the the columns marked with +LZ4 or +Zstd in table 1) show that both algorithms, zfp and our own, work well when chained together, entropy compression after lossy compression. In practical use, the compressed depth map requires about as much bandwidth as the JPEG compressed color image.

3.4. Hiding Latency with Reprojection of 2.5D Images

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 [PHE*11] of 2.5D images, i. e. the on-screen position of a pixel rendered for a previous viewer position is adapted for the current viewer position.

3.4.1. Reprojection Artifacts

Whenever a surface segment covers more pixels from the new vantage point, gaps between reprojected pixels will open (see figure 8 at top left). These small black lines can be closed by simply drawing points covering more than one screen pixel. In order to retain sharp edges, the point size has to be adaptive. While [WFC*12] seems to adapt the size of reprojected pixels solely based on their viewer distance and their movement (cf. figure 8 top center), we take the distance to reprojected neighbor pixels into account: up to a limit for the point size of 3 pixels, we reliably fill gaps between reprojected pixels (cf. figure 8 bottom middle).

However, this is not possible when large areas which are hidden in the original view become visible, such as parts of the isosurface or areas previously obstructed by the isosurface. One common way of avoiding this is to render a mesh of quads with the original pixels as vertices (cf. figure 8 top right). However, this will fill areas where there is simply no information, e. g. when surface segments become visible that have been obstructed before. As we like to convey that

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Codec	PSNR (dB)	Comp. rate (MPix/s)	Decomp. rate (MPix/s)	Rel. size (%)	+LZ4, rel. size (%)	+Zstd, rel. size (%)
raw/memcpy		39.0	1726.3	100.0	102.3	99.4
zfp prec	87.7	27.2	74.4	12.6	7.0	6.2
zfp accu	88.7	27.0	74.0	12.7	7.1	6.2
zfp fixed	71.0	17.7	45.3	25.0	17.9	14.8
Quant	86.9	72.8	101.0	25.0	17.8	16.7

Table 1: *Results for compression of depth map as shown in the middle of figure 4 with different settings for zfp and our algorithm "Quant": peak signal-to-noise ratio, bandwidth for compression and decompression in mega-pixels/second, relative size of compressed data of resp. lossy codec alone or when its output is compressed by LZ4 or Zstandard.*

there is no information for a screen area, we also allow for discarding fragments where triangles become too distorted (cf. figure 8 bottom center). For future improvements, we consider integrating more sophisticated reprojection algorithms such as described in [SSB*17].

The holes appearing in reprojected images are the most obvious artifacts. In addition, shading is not exact as reflectance calculations have been done for the previous viewer position. This last problem could be mitigated by an approach similar to "deferred shading", where color and normal are sent to the display system and shading is computed locally. This could even be extended so that classification is handled locally on the display system by sending raw data together with an index into an array of transfer functions which should be applied.

3.4.2. Amplified Compression Errors

The achieved PSNRs of more than 80dB for the depth codecs (cf. table 1) are relatively high compared to codecs for color images. In the case of HRR we still have to deal with two kinds of errors caused by this. On the one hand, a qualitative visual error can occur, if these values are used for depth compositing of local and remote images: based on the depth value of a pixel, its color value is chosen 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 as the PSNR suggests. Figure 6 illustrates such artifacts. On the other hand, together with reprojection quantitative errors can occur, when the shape of an object observed from a different vantage point becomes distorted as the pixel position has been wrongly estimated. One especially disturbing effect are "comet tail" artifacts exhibited by zfp as shown in the left of figure 4: at the edges between the far plane and the object, some pixels from the far plane are moved slightly closer to the viewer and thus show as very distorted triangles when a mesh is used for reprojection. This can be partially healed with the mesh shader that discards these distorted triangles. But our compression algorithm does not show this behavior.

Together with reprojection, JPEG chroma subsampling (e. g. when using the YUV420 color space) is not advisable: strong z variations, especially where neighboring pixels correspond to different objects will place neighbor pixels from the source picture with averaged colors at very different screen positions, and hence the averaging error will become much more obvious, see figure 7.

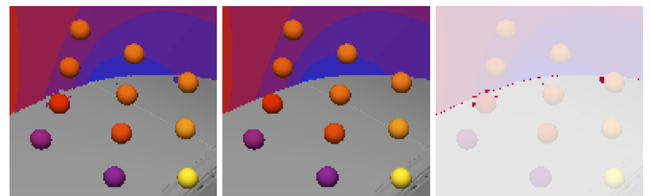

Figure 6: *Artifacts due to lossy depth compression: left with lossy compression, correct rendering in the middle and with highlighted differences at the right.*

However, we avoid the especially problematic case where pixels on the far clipping plane receive a wrong color by clipping those pixels.

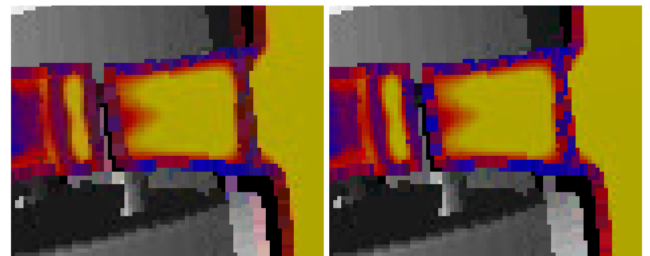

Figure 7: *Reprojection of JPEG compressed color images with chroma-subsampling (YUV411, left) and without (YUV444, right).*

3.5. Multi-Screen Projection Systems

The naïve approach to handling multi-screen systems is to generate remote images with 1-to-1 pixel mapping. In this case, disocclusion artifacts also appear at the the edges of these display surfaces. An easy solution is to fill in holes with pixels from neighboring displays. In a system like a CAVE, this requires a lot of pixels to be distributed to all display nodes, and for stereoscopic displays this has to be done for both eyes. Instead, we chose to render a collection of six images arranged on the faces of a cube centered at the point in the middle of the viewer's eyes, similar to a cube map, and reproject those. As the reprojection is done from the same source image for both eyes, this halves the number of pixels that the remote ray caster has to render, composite and send to the CAVE

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

cluster head node. Optionally, we do not render the cube face behind the viewer ("Cube map 5") and we render all but the front face at reduced resolution: the number of pixels to render can be reduced significantly with only little loss in image quality. For instance, halving the resolution of the side images in each dimension means that the total size of the images is only two (instead of five) times the size of the front image ("Cube map 1+$\frac{4}{4}$").

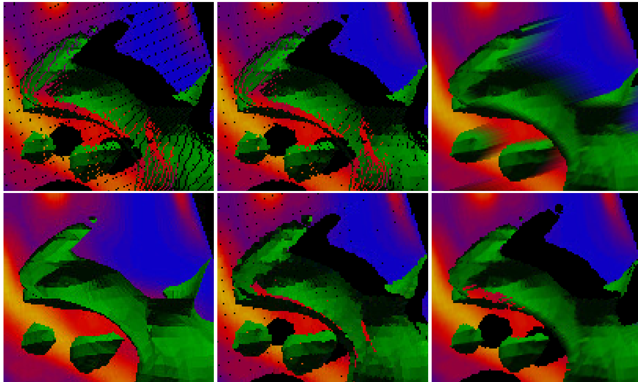

Figure 8: *Reprojection artifacts (in reading order): reprojection with constant point size, with point size adapted to depth of current pixel, as a mesh, direct rendering with correct viewing parameters, with point size taking reprojected screen position of neighbor pixels into account (our method), as a mesh with holes instead of distorted triangles.*

4. Evaluation and Practical Experience

For evaluating the system, we used the 5-side CAVE at HLRS: for each side, there is a projector with two video inputs. Every input has its dedicated Sandy Bridge (2×4 cores at 3.3 GHz) node with an NVidia Quadro P6000 GPU rendering 1600×1600 pixels. There is an equal head node with an output window of 1600×1000 pixels. This adds up to the 27.2 million pixels in table 2. The 11 nodes are connected with InfiniBand QDR. The remote end of the setup was part of the HLRS Vulcan cluster consisting of 32 Skylake nodes (2×20 cores each) running at 2.0 GHz and InfiniBand EDR interconnect. There was a shared Gigabit Ethernet connection from rank 0 to the head node of the CAVE cluster.

The data set used for testing was an OpenFOAM simulation of a pump turbine. The simulation was conducted by IHS (Institute of Fluid Mechanics and Hydraulic Machinery at the University of Stuttgart) on 128 cores and hence consisted of 128 partitions. Of the available 273 timesteps, every 5$^{\text{th}}$ was used. Each timestep has 5.8 million unstructured cells. In the example workflow (also cf. figure 3), the boundary of the domain was cut open and added as a spatial reference. Per-cell mapped data fields were interpolated to a vertex-based representation. A pressure isosurface was extracted as well as a colored cut through the turbulence measure *nuSgs*. These data were rendered remotely with DisCOVERay (see 2.2). In addition, stream lines were computed. But they were sent as geometry together with the mapped velocity to the CAVE cluster, where a transfer function was applied and the data was distributed to all the nodes. This work distribution was chosen, as warping 2.5D images of lines does not produce satisfying results.

Navigation within the visualization was made easy because of quick position updates with reprojection. Also precise placement of cutting planes or reference points for isosurfaces was possible. Interaction with the visualization modules might become even quicker in such a remote setting: DisCOVERay does not need to replicate the local geometry to the other nodes because of the sort-last compositing step, while COVER has to broadcast the local geometry to all other nodes. Measurements were taken (cf. table 2) while the timesteps were advanced without any interaction going on. The turbine as shown in figure 1 was centered on the front screen. This means that a new rendering is only requested after an animation frame was received and rendered, so that also the time to upload the new frame data to the GPU limits the achievable remote frame rate.

We tested different configurations for the proxy geometry, for which remote images are rendered. The first part of table 2 labelled "10+1 screens" shows the data for the naïve approach: for each of the 10 local screens in the CAVE and for the cluster head node, a remote image with exactly matching pixel dimensions is rendered. In this configuration, the master node receives 10 equally sized images from the remote renderer, and every display node is only sent the image corresponding to its own image area. This means that each node only receives the data for a single RGB-D image. But this also means that rendering does not have to take into account a lot of geometry, so that local frame rates of 60Hz are achievable. For this proxy geometry, different combinations of codecs for depth and RGB images have been tested. In table 2 we show the local frame rate, the bandwidth used for transmitting the image tiles from remote master to local master node, the latency which is added to the system by remote rendering, the rate at which remote images are rendered, together with the relative sizes of the compressed images.

In both "Cube map" configurations, the proxy geometry covers five faces of a cube, with the front side orientated towards the front wall of the CAVE, and every node of the CAVE cluster renders the same proxy geometry, but with different projections. This means that a larger number of pixels has to be broadcast to every cluster node and has to be copied from CPU to GPU, which leads to lower local frame rates. On the other hand, the additional latency incurred by remote rendering is halved for both "Cube map" configurations compared to rendering "10+1 screens". For a "Cube map 1+$\frac{4}{4}$" configuration, both local and remote frame rates are much higher, and as the most import screen of a 5-side CAVE is the front, image quality is not severely affected. Because of this, we generally recommend the "Cube map 1+$\frac{4}{4}$" setting and to occasionally engage COVER's high-quality mode initially introduced for volume rendering [SDWWL01] by clicking a button on the 3D input device while holding it above one's head: until another mouse button is clicked, pixel-exact remote images will be streamed.

The effect of compression on frame rate and latency is limited: as long as there is sufficient bandwidth, the additional latency is 0.2s for the "10+1 screens", and only when the Gigabit link becomes saturated, remote frame rate drops to 0.5Hz. The entropy

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

Proxy geometry	Depth codec	RGB codec	Total image size (MPix)	Local frame rate (Hz)	Bandwidth (MB/s)	Added latency (s)	Remote frame rate (Hz)	Relative depth size (%)	Relative RGB size (%)
	raw	raw			110	2	0.5	100.0	100.0
	LZ4	LZ4			42	0.2	4.0	8.0	3.0
	Zstd	Zstd			32	0.18	4.3	6.4	1.6
	Zstd	JPG			35	0.18	4.3	6.4	2.2
$10+1$ screens	zfp	Zstd	27.20	60	40	0.17	4.3	8.1	1.5
	zfp/Zstd	Zstd			6.7	0.16	4.3	0.3	1.5
	Quant	Zstd			65	0.24	3.3	18.8	1.5
	Quant/LZ4	LZ4			16	0.16	4.3	1.3	3.0
	Quant/Zstd	Zstd			10	0.16	4.3	1.0	1.5
Cube map 5	Quant/Zstd	Zstd	12.80	30	2.9	0.09	4.6	0.5	0.9
Cube map $1+\frac{4}{4}$	Quant/Zstd	Zstd	5.12	50	4.3	0.07	7.7	1.2	2.3

Table 2: *Performance of different color and depth image codec combinations and of different proxy geometry configurations.*

compressors LZ4 and Zstd both achieve a significant reduction in bandwidth, both for color and depth data. Probably due to the larger amount of empty pixels than in the data for table 1, the reduction is more prominent. As expected, Zstd achieves better compression ratios than LZ4, but observed performance is very similar. The zfp compressor with a configured "precision" of 16 achieves better compression than our depth codec (see 3.3.2, "Quant"). The output of both zfp and our codec is still a good target for subsequent entropy compression, as the depth codec combinations "zfp/Zstd", "Quant/LZ4" and "Quant/Zstd" show. Those codec combinations are especially advantageous in low-bandwidth settings, e. g. over a DSL line or Wi-Fi links. But if bandwidth is not severely limited, we recommend LZ4 for both depth and RGB data.

If more frequent updates are requested from the ray caster while the viewer position changes or the user is navigating, frame rates increase up to 20Hz for "Cube map $1+\frac{4}{4}$", but they are still limited by the performance of the remote ray caster and compositor.

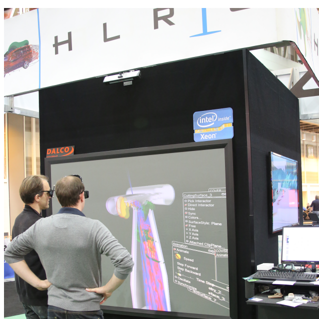

Figure 9: *Pump turbine at HLRS booth at SC '14 in New Orleans with stereo 3D hybrid remote rendering in Stuttgart.*

An early version of the system was demonstrated successfully at the HLRS booth (see figure 9) at the Supercomputing conference in New Orleans in 2014: 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 megapixels 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 with a bandwidth of about 10 MB/s was used. With full compression, display updates occurred at rates of about 10 frames/s. The system is also usable across a broadband Internet connection (50 Mbit/s DSL) on single screen systems, e. g. a laptop computer. These use cases show that remote hybrid rendering and visualization is a promising approach: it is applicable to long-distance links, display systems with high pixel counts and multiple surfaces, and low-bandwidth connections.

5. Related Work

In the remainder of this section we give a concise overview of important parallel visualization frameworks, discuss other uses of hybrid remote rendering, study techniques for 3D image reprojection, and explore compression of depth maps.

Parallel Remote Visualization Remote visualization is enabled by several parallel visualization systems. They allow for distributing the visualization pipeline [Mor13] over several clusters: VisIt [ACB*11] and ParaView [Aya16] take the algorithms for most of their modules from VTK [SML06] and build their own pipeline execution model around it, while EnSight [FK12] relies on independent implementations. Common to those systems is a client-server architecture. Distributed processing is restricted to data objects travelling from one remote cluster server to a local display client system, but they cannot be routed between remote servers in an arbitrary order. Interactivity can be improved by streaming objects at reduced resolution [AWD*09]. Even though there are generic solutions, which enable remote execution of a wide variety of applications [SME02], streaming of RGB images from the server to the display application is an integral part of the systems. Rendering typically takes place either local or remote [CGM*06]. So far,

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

hybrid approaches do not seem to be available in these common systems.

Hybrid Remote Rendering Hybrid remote rendering is a combination of local and remote image generation. It is employed in a variety of ways. Most often, the aim is to balance the use of local and remote resources as well as network bandwidth for optimal performance, while taking data availability into account. In the volume rendering system described by Engel et al., a low-resolution model that can be rendered locally is shown during interaction until high-fidelity still images from a more powerful remote system are available [EHT*00]. Tamm et al. [TK] describe a volume rendering system where rendering work is scheduled dynamically either on the remote system or the local client, depending on data availability and changes in available bandwidth and system performance. The Semotus Visum system by Luke et al. [LH02] used a combination of geometry and image streaming to facilitate warped re-rendering of remotely generated images. Wagner et al. [WFC*12] describe a visualization system based on the Vista framework that provides hybrid remote rendering: local and warped remote images are composited. In contrast to our work, there do not seem to be any provisions for object-based remote visualization.

3D Image Warping, Re-Rendering and Reprojection Warping of 3D images for adjusting to different projections or viewer positions is, as summarized by Mark [Mar99], a well-established technique. Common use cases are hiding of latency for adjusting projections for HMDs to head rotations [PLW13], adjusting to new viewer or object positions [SvLBF10], generating a second view from a monoscopic rendering for a stereoscopic projection [SSB*17], or aligning camera images with the real world in AR applications [SOS*17].

Planar 2.5D images are most common, but Doellner et al. [DHK12] use a collection of six images surrounding the viewer as a cube in so called “G-buffer cube maps”, where each of them contains several layers such as viewer distance, normals and colors as input for renderings from new vantage points. This allows them to re-render image streams on mobile devices with correct lighting. We follow a similar approach, but do not rely on G-buffers, saving the transmission of normals in addition to RGB-D images.

There are different approaches to handle areas disoccluded in the target projection. In our system we implement basic ideas: adapting the size of reprojected points – but at the cost of not filling all holes, and rerendering the depth image as a height field – which leads to “rubber banding” in the filled areas. Didyk et al. [DRE*10] propose to warp a grid adapted to the depth map and reuse information from previous frames. Schollmeyer et al. [SSB*17] present a high-quality solution by inserting blurred background images – the farther into the hole, the more blurred. These approaches should work well in our context, and could be implemented in the future. More difficult to apply is the proposition of Pajak et al. [PHE*11] to fill holes based on motion vectors.

Compression and Transmission of RGB-D Images Transferring depth images with 24 bits per channel as needed by hybrid remote rendering requires efficient – both in terms of compression ratio and rate – codecs for high-precision images. Common image codecs such as JPEG only work on channels with 8 or 10 bit precision, which is not sufficient for encoding depth maps used for compositing. Because of this, there are efforts e. g. by Pece et al. [PKW11] to encode the depth data as RGB, so that regular video codecs can be reused for depth images. But due to the lossy color channel compression and the subsampling of chroma components they only reach rather low precision. As depth maps can be represented as floating point images, dedicated codecs like sz as described by Di et al. [DC16] or zfp as described by Lindstrom [Lin14] can be used. But e. g. zfp is not optimal for the use case of depth images, where a lot of background depth occurs, as there is significant error at the edges between background and the data set. Lossless video codecs as employed by Leaf et al. [LMM] for storing floating point volume data are more promising.

There are also algorithms dedicated to compressing depth images, such as by Cappellari et al. [CCRCK09] and Schiopu et al. [ST12], but they do not provide the achieved compression bandwidth. Mehrotra et al. [MZC*11] propose an algorithm that is adapted to the varying accuracy of their capturing device and, while only operating on 16 bit values, they achieve high compression ratios at a rate comparable to our algorithm. Our own algorithm achieves higher precision at a lower compression rate, and with its regular data pattern, it lends itself to an implementation on the GPU, although up to now this has only been done for the compression part.

Another approach is to compress a geometric representation of the depth map as e. g. in the Semotus Visum system [LH02] or based on BSP trees as by Sarkis et al. [SZD10]. Often this has the added benefit that warping can reuse the data structure that has been employed for transmission.

6. Conclusions and Future Work

Overall, our results show that with Vistle and HRR it is efficiently possible to explore remote data sets from immersive virtual environments in high quality, and there is flexibility in order to adapt to available network and compute resources. While we did not yet examine the scaling properties of the system for a large number of nodes and cores, we expect to be able to make use of a large computing systems, as the employed algorithms promise logarithmic or better scaling behavior.

In the future, we plan to improve reprojection quality by incorporating more sophisticated methods including deferred shading. Also, with the advent of IPv6, we hope to benefit from direct connections between render nodes and display nodes, instead of routing all network traffic through head nodes.

7. Acknowledgments

This work has been supported in part by the CRESTA project that has received funding from the European Community’s Seventh Framework Programme (ICT-2011.9.13) and in part by the bwVisu and bwVisu2 projects funded by the Ministry of Science, Research and the Arts of the country of Baden-Württemberg.

We thank the institutes IHS and ITLR at the University of Stuttgart for providing the data used in the evaluation. We also thank the anonymous reviewers for their valuable suggestions.

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

References

[ACB*11] AHERN S., CHILDS H., BRUGGER E., WHITLOCK B., MEREDITH J.: VisIt: An end-user tool for visualizing and analyzing very large data. *Proc SciDAC* (2011). 8

[Aum15] AUMÜLLER M.: The Architecture of Vistle, a Scalable Distributed Visualization System. In *Solving Software Challenges for Exascale*. Springer International Publishing, Cham, Feb. 2015, pp. 141–147. 2

[AWD*09] AHRENS J. P., WOODRING J., DEMARLE D. E., PATCHETT J., MALTRUD M.: Interactive remote large-scale data visualization via prioritized multi-resolution streaming. In *UltraVis '09: Proceedings of the 2009 Workshop on Ultrascale Visualization* (New York, USA, 2009), ACM, pp. 1–10. 8

[Aya16] AYACHIT U.: *The Paraview Guide*, updated for ParaView version 5.0 ed. Community Edition. Kitware, Inc., Nov. 2016. 8

[CCRCK09] CAPPELLARI L., CRUZ-REYES C., CALVAGNO G., KARR J.: Lossy to Lossless Spatially Scalable Depth Map Coding with Cellular Automata. *2009 Data Compression Conference* (2009), 332–341. 9

[CGM*06] CEDILNIK A., GEVECI B., MORELAND K., AHRENS J. P., FAVRE J. M.: Remote Large Data Visualization in the ParaView Framework. In *EG PGV'06: Proceedings of the 6th Eurographics conference on Parallel Graphics and Visualization* (2006), pp. 163–170. 8

[CNSD93] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A.: 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* (1993), 142. 2

[DC16] DI S., CAPPELLO F.: Fast Error-Bounded Lossy HPC Data Compression with SZ. *IPDPS* (2016). 9

[DHK12] DOELLNER J., HAGEDORN B., KLIMKE J.: Server-based rendering of large 3D scenes for mobile devices using G-buffer cube maps. In *Proceedings of the 17th International Conference on 3D Web* (New York, New York, USA, Aug. 2012), ACM, pp. 97–100. 9

[DRE*10] DIDYK P., RITSCHEL T., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Adaptive Image-space Stereo View Synthesis. *VMV* (2010). 9

[EHT*00] ENGEL K., HASTREITER P., TOMANDL B., EBERHARDT K., ERTL T.: Combining local and remote visualization techniques for interactive volume rendering in medical applications. *IEEE Visualization* (2000). 9

[EMP09] EILEMANN S., MAKHINYA M., PAJAROLA R.: Equalizer - A Scalable Parallel Rendering Framework. *IEEE Trans. Vis. Comput. Graph. ()* (2009). 3

[FK12] FRANK R., KROGH M. F.: The EnSight Visualization Application. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, Bethel E. W., Childs H., Hansen C., (Eds.). Oct 2012, pp. 429–442. 8

[INH03] IOURCHA K. I., NAYAK K. S., HONG Z.: Fixed-rate block-based image compression with inferred pixel values. 5

[LH02] LUKE E., HANSEN C. D.: Semotus Visum - A Flexible Remote Visualization Framework. *IEEE Visualization* (2002), 61–68. 9

[Lin14] LINDSTROM P.: Fixed-Rate Compressed Floating-Point Arrays. *Visualization and Computer Graphics, IEEE Transactions on 20*, 12 (2014), 2674–2683. 3, 9

[LMJC16] LARSEN M., MORELAND K., JOHNSON C. R., CHILDS H.: Optimizing multi-image sort-last parallel rendering. *LDAV* (2016), 37–46. 3

[LMM] LEAF N., MILLER B., MA K.-L.: In situ video encoding of floating-point volume data using special-purpose hardware for a posteriori rendering and analysis. In *2017 IEEE 7th Symposium on Large Data Analysis and Visualization (LDAV)*, IEEE, pp. 64–73. 9

[Mar99] MARK W. R.: *Post-rendering 3D image warping: Visibility, reconstruction, and performance for depth-image warping*. PhD thesis, University of North Carolina at Chapel Hill, 1999. 9

[MCEF94] MOLNAR S., COX M., ELLSWORTH D., FUCHS H.: A sorting classification of parallel rendering. *Computer Graphics and Applications, IEEE 14*, 4 (1994), 23–32. 3, 4

[MKPH11] MORELAND K., KENDALL W., PETERKA T., HUANG J.: An image compositing solution at scale. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for* (2011), pp. 1–10. 3

[Mor13] MORELAND K.: A Survey of Visualization Pipelines. *Visualization and Computer Graphics, IEEE Transactions on 19*, 3 (2013), 367–378. 8

[MZC*11] MEHROTRA S., ZHANG Z., CAI Q., ZHANG C., CHOU P. A.: Low-Complexity. Near-Lossless Coding of Depth Maps from Kinect-Like Depth Cameras. *2011 IEEE 13th International Workshop on Multimedia Signal Processing* (2011), 1 6. 9

[PHE*11] PAJAK D., HERZOG R., EISEMANN E., MYSZKOWSKI K., SEIDEL H.-P.: Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming. *Computer Graphics Forum 30*, 2 (2011), 415–424. 5, 9

[PKW11] PECE F., KAUTZ J., WEYRICH T.: Adapting Standard Video Codecs for Depth Streaming. In *Joint Virtual Reality Conference of EuroVR 2011* (2011), Blach R., Coquillart S., DCruz M., Steed A., Welch G., (Eds.). 4, 9

[PLW13] PEEK E. M., LUTTEROTH C., WÜNSCHE B.: More for less - Fast image warping for improving the appearance of head tracking on HMDs. *IVCNZ* (2013), 41–46. 9

[RFL*98] RANTZAU D., FRANK K., LANG U., RAINER D., WOESSNER U.: COVISE in the CUBE: An Environment for Analyzing Large and Complex Simulation Data. *2nd Workshop on Immersive Projection Technology* (1998). 3

[SDWWL01] SCHULZE-DÖBOLD J., WÖSSNER U., WALZ S. P., LANG U.: Volume rendering in a virtual environment. In *Immersive Projection Technology and Virtual Environments 2001* (Vienna, 2001), Fröhlich B., Deisinger J., Bullinger H.-J., (Eds.), Springer Vienna, pp. 187–198. 7

[SME02] STEGMAIER S., MAGALLÓN M., ERTL T.: A generic solution for hardware-accelerated remote visualization. In *Proceedings of the Symposium on Data Visualisation 2002* (Aire-la-Ville, Switzerland, Switzerland, 2002), VISSYM '02, Eurographics Association, pp. 87–ff. 8

[SML06] SCHROEDER W., MARTIN K., LORENSEN B.: *The Visualization Toolkit*. An Object-Oriented Approach to 3D Graphics. Kitware, Inc., Dec. 2006. 8

[SOS*17] SCHOPS T., OSWALD M. R., SPECIALE P., YANG S., POLLEFEYS M.: Real-Time View Correction for Mobile Devices. *Visualization and Computer Graphics, IEEE Transactions on 23*, 11 (Nov. 2017), 2455–2462. 9

[SSB*17] SCHOLLMEYER A., SCHNEEGANS S., BECK S., STEED A., FROEHLICH B.: Efficient Hybrid Image Warping for High Frame-Rate Stereoscopic Rendering. *Visualization and Computer Graphics, IEEE Transactions on 23*, 4 (2017), 1332–1341. 6, 9

[ST12] SCHIOPU I., TABUS I.: Depth image lossless compression using mixtures of local predictors inside variability constrained regions. *ISCCSP* (2012), 1 4. 9

[SvLBF10] SMIT F. A., VAN LIERE R., BECK S., FRÖHLICH B.: A shared-scene-graph image-warping architecture for VR: Low latency versus image quality. *Computers and Graphics 34*, 1 (Feb. 2010), 3–16. 9

[SZD10] SARKIS M., ZIA W., DIEPOLD K.: Fast Depth Map Compression and Meshing with Compressed Tritree. In *Solving Software Challenges for Exascale*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 44–55. 9

[TJV*10] TAYLOR, II R. M., JERALD J., VANDERKNYFF C., WENDT J., BORLAND D., MARSHBURN D., SHERMAN W. R., WHITTON M. C.: Lessons about Virtual Environment Software Systems from 20 Years of VE Building. *Presence: Teleoperators and Virtual Environments 19* (2010), 162 178. 4

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.

[TK] TAMM G., KRÜGER J.: Hybrid Rendering with Scheduling under Uncertainty. *Visualization and Computer Graphics, IEEE Transactions on 20*, 5, 767–780. 9

[UAW*99] USOH M., ARTHUR K., WHITTON M., BASTOS R., STEED A., SLATER M., BROOKS F.: Walking > walking-in-place > flying, in virtual environments. *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (July 1999). 2

[WFC*12] WAGNER C., FLATKEN M., CHEN F., GERNDT A., HANSEN C. D., HAGEN H.: Interactive Hybrid Remote Rendering for Multi-pipe Powerwall Systems . In *Virtuelle und Erweiterte Realität - 9. Workshop der GI-Fachgruppe VR/AR*, Geiger C., Herder J., Vierjahn T., (Eds.). Shaker Verlag, Aachen, Aug. 2012, pp. 155–166. 2, 5, 9

[WJA*16] WALD I., JOHNSON G., AMSTUTZ J., BROWNLEE C., KNOLL A., JEFFERS J., GUNTHER J., NAVRATIL P.: OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *Visualization and Computer Graphics, IEEE Transactions on PP*, 99 (2016), 1–1. 3

[WLR94] WIERSE A., LANG U., RÜHLE R.: A system architecture for data-oriented visualization. In *Database Issues for Data Visualization, LNCS, vol. 871*, Lee J. P., Grinstein G. G., (Eds.). Springer-Verlag, Berlin/Heidelberg, 1994, pp. 148–159. 2, 3

[WQ10] WANG R., QIAN X.: *OpenSceneGraph 3.0: Beginner's Guide*. Packt Publishing, Dec. 2010. 3

[WWB*14] WALD I., WOOP S., BENTHIN C., JOHNSON G. S., ERNST M.: Embree: a kernel framework for efficient CPU ray tracing. *ACM Transactions on Graphics (TOG) 33*, 4 (July 2014), 143–8. 3

© 2019 The Author(s)
Eurographics Proceedings © 2019 The Eurographics Association.