Foveated Encoding for Large High-Resolution Displays

Florian Frieß, Matthias Braun, Valentin Bruder, Steffen Frey, Guido Reina, and Thomas Ertl

Abstract— Collaborative exploration of scientific data sets across large high-resolution displays requires both high visual detail as well as low-latency transfer of image data (oftentimes inducing the need to trade one for the other). In this work, we present a system that dynamically adapts the encoding quality in such systems in a way that reduces the required bandwidth without impacting the details perceived by one or more observers. Humans perceive sharp, colourful details, in the small foveal region around the centre of the field of view, while information in the periphery is perceived blurred and colourless. We account for this by tracking the gaze of observers, and respectively adapting the quality parameter of each macroblock used by the H.264 encoder, considering the so-called visual acuity fall-off. This allows to substantially reduce the required bandwidth with barely noticeable changes in visual quality, which is crucial for collaborative analysis across display walls at different locations. We demonstrate the reduced overall required bandwidth and the high quality inside the foveated regions using particle rendering and parallel coordinates.

Index Terms—Large high-resolution displays, Fovetaed Encoding, Remote Visualisation

1 Introduction

Streaming and conferencing technologies for working remotely are becoming increasingly important. Many of the widely adopted solutions support resolutions of 4k, but have to make compromises with respect to video quality under heavy load. Aiming primarily at desktop or mobile device usage, they do not offer support for large tiled high-resolution displays and image resolutions beyond 4k. While there are specialized solutions that support sharing the output of these types of systems based on hardware-accelerated video encoding, these also make compromises between quality and bandwidth. They either deliver a high quality image and therefore induce bandwidth requirements that cannot generally be met, or they uniformly decrease the quality to maintain adequate frame rates. However, in visualisation in particular, details are crucial in areas that are currently under investigation. In contrast, lower quality is sufficient in other areas that are mostly required for context. With techniques that uniformly adjust quality, some of the limited bandwidth is wasted on image areas outside of the user's region of interest.

To address this, we adapt the concept behind foveated rendering to dynamically adjust compression settings for streaming. This means that we track the gaze of users to locally adapt the quality of the encoding, in order to improve the image quality in the region of interest while keeping the overall required bandwidth as low as possible. This is achieved by lowering the encoding quality in the periphery, which is barely noticeable due to characteristics of the human visual system. A challenge in this context is that large high-resolution display setups are oftentimes unique research prototypes, which means that any technique designed to share content between such systems needs to be able to deal with different systems. For instance, there are systems that use as many GPUs as possible in a single machine so that applications can run just like on a desktop computer, while others rely on a GPU cluster in order to visualise large and complex scientific data sets, such as particle data or 3D flow fields. Although their individual hardware setup, and in addition the software that runs on them, might be different, these systems still share one property: they have a large frame buffer that is potentially distributed. With the recent wide availability of fast, low-latency hardware video encoders and decoders, we choose to use a combination of such hardware and frame buffer streaming as the basis of our solution for interactively sharing visualisations between such systems. These hardware video encoder and decoder chips represent an ideal solution since they are available on almost all newer GPUs and use a separate pipeline, thus not interfering with other GPU tasks, like rendering and computation. This allows sharing the content of these displays in real time, but considering the trade-off between quality and bandwidth is still required.

Combining the hardware encoders with the foveated encoding we created an approach that provides multiple users with a large high-resolution image shared from another large display, preserving high quality locally, while the rest of the image is compressed heavily in order to reduce the required bandwidth. Our approach makes high quality screen capture sessions between two large high-resolution displays possible over commonly available Ethernet connections, e. g. 100 to 400 Mb/s. Every captured frame, on each of the display nodes that render a part of the (distributed) frame buffer, is optionally converted, rotated and downscaled or divided to fit the colour format and maximum size used by the hardware encoder. By tracking the gaze of the observers on the client side, we determine foveated regions around their centre of vision. Every time new foveated regions are received by the server, all encoding macroblocks are checked for intersection with those regions. For blocks inside a region, the distance to the centre of the region is computed. Based on this distance, the quality parameter of the macroblock is changed, so that close to the centre of the region the quality is better. Following the visual acuity fall-off model, the quality decreases towards the border of the foveated region until it reaches the lowest value, which is also used for all macroblocks outside of the regions. The encoded frames are forwarded to the clients that decode and display them.

In the following, Sect. 2 first gives an overview of related work. Next, Sect. 3 covers the model for the visual acuity fall-off we used to compute the foveated regions. Sect. 4 describes our method conceptually, followed by a detailed description of the implementation in Sect. 5. Finally, Sect. 6 presents and discusses the results of latency and throughput tests.

The major contributions of this work are our approach for foveated encoding with hardware encoders supporting multiple users, and our implementation of foveated encoding for large high-resolution displays.

2 Related Work

There are several approaches and systems for remote visualisation that have been proposed in recent years [7, 31]. Although SAGE [29] and SAGE 2 [28] by Renambot et al. mainly focus on collaboration they also offer remote visualisation capabilities. SAGE uses pixel streaming

• *Florian Frieß is with University of Stuttgart. E-mail: florian.friess@visus.uni-stuttgart.de.*
• *Matthias Braun is with University of Stuttgart. E-mail: matthias.braun@visus.uni-stuttgart.de.*
• *Valentin Bruder is with University of Stuttgart. E-mail: valentin.bruder@visus.uni-stuttgart.de.*
• *Steffen Frey is with University of Groningen. E-mail: s.d.frey@rug.nl.*
• *Guido Reina is with University of Stuttgart. E-mail: guido.reina@visus.uni-stuttgart.de.*
• *Thomas Ertl is with University of Stuttgart. E-mail: thomas.ertl@vis.uni-stuttgart.de.*

Manuscript received 30 Apr. 2020; revised 31 July 2020; accepted 14 Aug. 2020. Date of publication 14 Oct. 2020; date of current version 15 Jan. 2021. Digital Object Identifier no. 10.1109/TVCG.2020.3030445

but it requires code changes to applications in order to generate the streams. SAGE 2 employs a browser-centric application allowing it to run a large variety of systems and therefore moved away from pixel streaming. There are also systems that have been developed specifically with the use case of interactive high-resolution streaming of visualisation content in mind. Biedert et al. [1] developed a streaming solution using hardware video compression on the GPU to achieve high frame rates. They use one GPU to encode each part of the tiled display and synchronise the resulting video streams on the server and client side to ensure a smooth playback. While the system offers a high frame rate, it does not allow to change the settings of the encoder on-the-fly, meaning that it provides a constant image quality and bandwidth requirements. Marrinan et al. [19] showed a system that did not use any compression but is still able to stream high-resolution content in real-time. They make use of multiple TCP socket servers, launching a single TCP server for each node that renders a part of the final image, allowing parallel processing of rendering and transmitting a frame to one or more clients. Each client, i. e. the nodes at the display side, will connect to one (or more) servers and will receive partial frames, which have to be redistributed to match the layout of the tiled display. Their system is able to reach high frame rates, however it requires a connection with an enormous bandwidth to do so, which is not commonly available.

A large amount of work addressing bandwidth limitations has been done so far. Levoy [18] and Bolin and Meyer [2] proposed adaptive sampling techniques, while Koller et al. [16] and Herzog et al. [14] used image compression techniques. Pajak et al. [26] use augmented video information to efficiently compress and stream images of dynamic 3D models. Moreland et al. [22] present an approach using level-of-detail techniques providing an interactive rendering regardless of the network performance. A technique presented by Frey et al. [11] is targeted towards scientific visualisation in a remote setup. They integrate sampling and compression techniques to balance visualisation and transfer to optimise image quality. Frieß et al. [12] follow a similar approach and try to get the best possible image quality for a limited bandwidth. They split the image into smaller tiles and use a convolutional neural network to predict the quality and size of the image for different encoder settings. Based on the predictions and an optimizer they assign each tile a different encoding setting to preserve the quality in regions with fine structures while reducing the image quality in more homogeneous areas. While this approach also reduces the required bandwidth, it is unaware of the current user focus and therefore potentially spends significant bandwidth on regions that nobody is currently looking at it.

Foveated video compression, achieved by changing the implementation of the video encoder, has been explored previously. Lee and Bovik [30] improved the efficiency of video processing by constructing several foveated video processing algorithms: foveation filtering (local bandwidth reduction), motion estimation, motion compensation, video rate control, and video postprocessing. Their approach led to a better computational efficiency by using a protocol between the encoder and the decoder. Chen and Guillemot [5] adapted the macroblock quantization adjustment in the H.264/advanced video coding by using a foveated model. This model enhances the spatial and temporal just-noticeable-distortion models in order to account for the relationship between visibility and eccentricity. For each macroblock the quantization parameter is optimized based on this model. Illahi et al. [15] adapted the video encoder of a cloud-gaming application so that it changes the quality of the encoding based on the gaze direction of the player. They adjust the quality parameter of each macroblock based on the current gaze position, increasing the quality in macroblocks the player looks at and decreasing the quality in the remaining ones. Zare et al. [35] proposed to use tiled based encoding in order to transmit wide-angle and high-resolution spherical panoramic video content to head-mounted displays. They store the video content in two different resolutions, divided into multiple tiles using the High Efficiency Video Coding (HEVC) standard. Based on the user's current viewport tiles are selected. For these tiles the highest captured resolution is transmitted while the remaining tiles are transmitted from the low-resolution version. In contrast to these techniques, our system is not restricted to a single machine but is able to deal with different hardware set-ups used to build large high-resolution displays.

A variety of parallel rendering frameworks and middlewares have been suggested to display content on large high-resolution displays. The Cross Platform Cluster Graphics Library (CGLX) [8, 27] is an OpenGL-based framework for distributed high-performance visualisations. It allows adapting existing or developing new OpenGL-based applications for tiled displays. It also supports co-located collaboration through multiple multi-touch devices to which the updated scene information is streamed. Eilemann et al. [9] developed Equalizer, a parallel OpenGL-based middleware that supports scalable display environments. It provides a large set of features to support a wide variety of research and industry applications. OmegaLib [10] is based on Equalizer. It provides tools to develop immersive 2D and 3D applications for systems ranging from large display walls to CAVEs. Additionally it supports dynamic reconfiguration of the display environment to interactively allocate 2D and 3D workspaces.

3 Visual Acuity Fall-Off

The ability of a person to recognize and distinguish small details is usually referred to as *visual acuity*. There is a large body of work that attributes the human eye a fall-off in visual acuity towards the periphery. This means humans typically cannot make out many details and colours outside a small foveal region around the fixation point. In contrast, near this point at the centre of the field of view, humans perceive the highest number of details and a broad range of colours. Strasburger et al. [33] give an overview on different works on peripheral vision and pattern recognition. This property has been exploited in numerous works in computer graphics and visualisation that implement foveated rendering methods [4, 13, 32, 34].

The visual acuity fall-off can be modelled as a hyperbolic function. This model matches the density distribution of photoreceptors in the human macula and has been validated with low level vision tasks [3, 33]. By using a cubic function, we approximate the fall-off function to determine a quality factor q, based on the distance d to the centre of vision:

$$q = (1-d)^3, \text{ with } d \in [0,1]. \tag{1}$$

We apply this function in a foveal region that we determine conservatively by assuming a 50° field-of-view. The average human macula size (the region in the retina containing fovea, parafovea, and perifovea) is typically below 20°. We encode with the lowest quality outside this region. Due to the discrete nature of the quality parameter QP, used by the H.264 video codec, that we apply for encoding inside the foveal region, we approximate Equation 1 with a piecewise linear function to determine the quality parameter at a specific position:

$$QP = QP_{\min} - \text{round}(q \cdot (QP_{\min} - QP_{\max})). \tag{2}$$

In this work, we use $QP_{\min} = 51$ and $QP_{\max} = 11$, resulting in 41 different quality parameters. This produces a smooth approximation of the visual acuity fall-off function, and results in barely perceptible visual impact in the periphery.

4 Method

In this section, we describe the full pipeline of our method on both sides of the screen sharing session (cf. Fig. 1). The server side renders and captures the visualisation, and then carries out foveated encoding of the captured frames. For this we change the quality of the macroblocks, used by the H.264 video codec, based on their distance to the foveated regions. The client side provides the respective foveated regions, based on tracking the users, and displays the decoded frames.

The server side uses the following steps to produce the foveated encoding. This is computed in parallel on all nodes that render a part of the (potentially distributed) frame buffer. Firstly the last rendered frame is acquired as a texture. Since the encoder expects a different colour format we use a compute shader to perform the necessary conversion. This shader also handles the optional rotation, in 90° steps, in addition to the, also optional, downscaling or dividing into separate tiles of the texture in order to fulfil the resolution requirements of the encoder. It outputs either the converted, rotated and downscaled texture or multiple

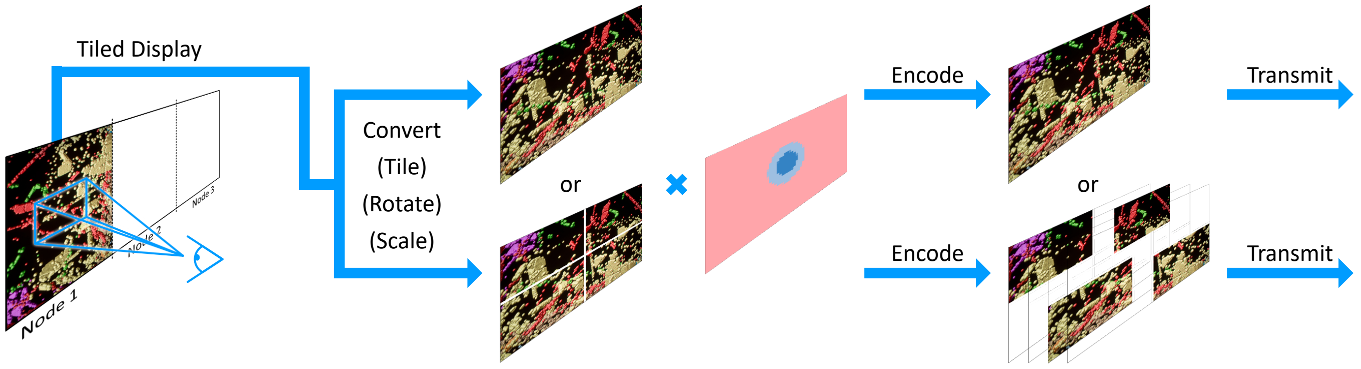


Fig. 1: Overview of the major parts of the pipeline from capturing a frame on the server side to sending it to the client side. Each captured frame is converted, and optionally rotated, downscaled or divided to fit the resolution requirements of the encoder. Based on the received foveated regions the constant quality parameter for each macroblock of the encoder is adapted and the frame or the tiles are encoded. The client side decodes and displays the received frames or tiles.

converted and rotated tiles. Initially all macroblocks are considered to be in the peripheral region. In the next step, the intersection between the last received foveated regions and the macroblocks is computed. If there is an intersection, the new quality parameter for the respective macroblock is computed based on the distance to the centre of the region. Since we want to reduce the required bandwidth, all macroblocks outside the regions always use the lowest possible quality parameter leading to the average colour inside the macroblock. All encoded frames, or tiles, are sliced to fit UDP packages and forwarded to the client.

The client side uses the following steps to display the incoming encoded streams, which are again performed on all nodes, that render a part of the (potentially distributed) frame buffer, in parallel. All received UDP packages are reordered based on the timestamp and the, monotonically increasing, sequence number of the package. Then the (potentially sliced) frames are re-assembled and queued for decoding as well as display once they are complete. Incomplete frames are dropped after a user specified time has passed (we use 50 ms). Additionally a single node on the client side computes the foveated regions by tracking the users and forwards these regions to the server side.

The following paragraphs provide additional details for the different steps of our method.

Tracking data We need to track users to acquire the foveated regions for the encoding. We offer two approaches to do that: the first approach uses a tracking system to track multiple users, while the second approach uses the mouse to compute the foveated region. Both deliver rectangles, representing the foveated region, with 120 Hz to a single node on the client side. These rectangles are determined based on the current position with respect to the display and the view direction of the user in case of the tracking system or on the screen-space position and a user defined width and height in case of the mouse. We use the average acuity values described in Sect. 3, so the (configurable) size of the rectangle spans 50° horizontally and 50° vertically from the user's position. The mouse based tracking was originally implemented for debugging but can also be used as an alternative in case a tracking system is not available.

Encoding and Decoding Encoding of the full frame, or the tiles, is done by dedicated hardware encoders that are separated from other workloads (compute or graphics), so they are not affected by any other load on the GPU and work asynchronously. This makes encoding a fire-and-forget operation: the submitting thread does not need to wait for the encoding to finish, allowing to capture high frame rates. Once the frame is encoded it is downloaded into main memory, sliced to fit UDP packets and sent to the consumer. We use a constant quality parameter and the maximum bitrate of the encoder is not limited, i. e. the encoding will always reach the desired quality, and we insert a key frame in regular intervals. This quality parameter can be set to integral values between 51 and 1, with 51 resulting in the lowest quality and 1 yielding the best quality – basically turning off compression. In order to adapt the quality we use a map that contains an offset for the quality parameter of each macroblock. This allows us to change the quality parameter in the interval $[11, 51]$ for each macroblock individually. According to our experiments, going lower than 11 does not yield any visible difference and only increases the required bandwidth. We scale the quality parameter for macroblocks inside the foveated region between 11 (close to the centre) and 51 (at the border). Decoding, same as encoding, is performed asynchronously on the dedicated hardware units on the GPU. Each decoded frame is copied into the display queue where it remains until it is displayed. In case the frame is divided instead of downscaled we use a separate encoder and decoder for each tile.

Network We support two classes of network technologies: side-local communication between nodes driving a single tiled display (as well as arbitrary other nodes that can provide frames) and internet communication between sides. For communication between sides we use our own UDP-based protocol in order to achieve the best possible saturation of the network and to avoid latency introduced by TCP. On-side communication is implemented using the Message Passing Interface (MPI), which is the de-facto standard for communication in HPC clusters. MPI has the advantage of providing specialised implementations for high-speed network technologies like InfiniBand (IB) that talk almost directly to the hardware, while it also can be used with standard Ethernet connections.

Display We render the decoded and assembled frames directly, therefore all scaling and positioning transformations are performed on the fly on the GPU. In order to ensure that the displayed frame is consistent, even if it is displayed across multiple nodes, we provide two different approaches: The first one uses an MPI barrier that synchronises all nodes just before the buffer swap of the next frame. This is a portable solution, but it does not guarantee that the buffer swap is happening at exactly the same time and it is a comparably expensive operation. The second one is based on NVIDIA's Quadro Sync technology [24], which requires additional hardware support to use it. Quadro Sync uses a proprietary protocol in the graphics driver and a separate Ethernet network to ensure that all displays swap buffers at the same time. This makes it the preferred way to synchronise the output over multiple display nodes as it does not interfere with MPI or any other component of the software.

5 Implementation

Architecture and implementation of our system are designed such that the system can run in different kinds of tiled display environments, while utilising any hardware support available. To achieve this, we use a declarative approach via XML, which allows to configure each node in the system in the same file. The nodes are identified by pertinent

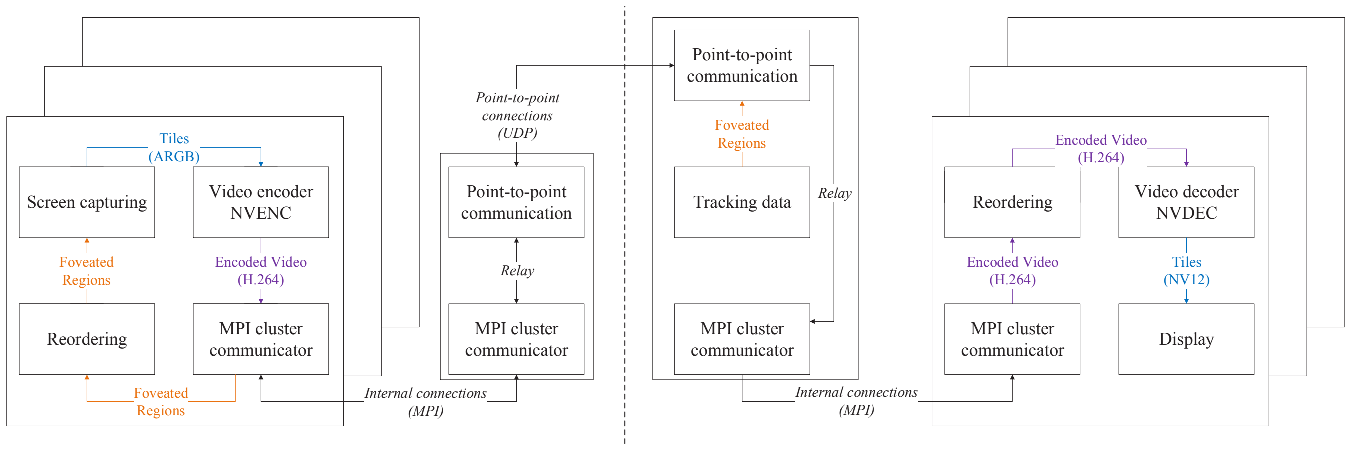


Fig. 2: Overview of the full system. On the left is the server side and on the right the client side, separated by the dotted line. Depicted are the display nodes of the large displays and the streaming nodes, which transmit the data between the sides. For each node we show the components (see Sect. 5) of the system and data flow between them. Teal-coloured connections represent raw images, while purple connections represent encoded images. The tracking data, i. e. the look-at rectangles, are represented by orange connections. Black-coloured connections represent any of the above, i. e. data-agnostic communication links.

strings like the host name, an IP address or one of the MAC addresses of the computer. The settings for encoder and decoder can also be configured in the same file, as well as the number of tiles per node, in case foveated encoding is used.

Our system is written in C++ and relies on Direct3D 11, the Desktop Duplication API [20], the Video Codec SDK (version 8.2.15) from NVIDIA [25] and NVIDIA's NVAPI. The Video Codec SDK contains NVENC that is used to encode video streams on the GPU and NVDEC for decoding them, again on the GPU. NVAPI allows to address the proprietary framelock feature of Quadro GPUs, Quadro Sync. Our aim was to be as efficient as possible in our implementation to be able to process all data in real time. On the CPU we use zero-copy implementations whenever possible and we try to avoid memory allocations by using pools of frequently used objects. Additionally we avoid moving data between the CPU and GPU and reduce the number of GPU-to-GPU copies. In the following, we describe the integration and interplay of these components in the full system (cf. Fig. 2). Below, where applicable, we first describe the normal use and then the changes we made for the foveated encoding.

5.1 Tracking data

In order to acquire the foveated regions for the encoding we can track the viewing direction of multiple users. For this, we use devices equipped with reflective markers. Each device, referred to as *rigid body*, has a unique pattern of reflectors that is used as an identifier in the system. We use a system that consists of two Prime 13 cameras and 22 S250e cameras from NaturalPoint OptiTrack and use their software Motive to stream the current position and orientation of each rigid body to any machine on the same network via UDP. It is vital to calibrate the initial orientation of a rigid-body in reference to the display, this enables computing a correct look-at rectangle for the foveated regions. Therefore, we determine the position of the bottom left corner of the display in the tracking coordinate system. In addition to the position we also set the height and width of the display, which determine the up vector and right vector of the display. To calibrate a rigid body, it needs to be placed inside the tracked area, pointing towards the display. The orientation of the rigid body, as seen by the tracking system, is saved automatically, so each rigid body only needs to be calibrated once.

Next, we describe how to compute the look-at rectangles for a single *rigid body*. For this, we first need to determine the intersection of the view direction with the display plane. A 2D illustration of the setup with two rigid bodies, including the relevant variables used in the calculation, is depicted in Fig. 3.

Intersection

In order to compute the intersection point $\vec{s}$ on the display plane, we first determine the observer's viewing direction $\vec{d}$.

$$\vec{d} = \tilde{Q}_{rb} \cdot Q_{rb}^{-1} \cdot \begin{pmatrix} 0 \\ -\vec{n}_d \end{pmatrix} \tag{3}$$

Here, we use the quaternion $\tilde{Q}_{rb}$ that represents the current orientation of the rigid body, the inverse quaternion Q_{rb}^{-1} of its calibrated neutral orientation and the normal vector $\vec{n}_d \in \mathbb{R}^3$. The normal of the display plane $\vec{n}_d$ points towards the tracking area in front of the display wall. Using the First Intercept Theorem, we then compute the distance δ between the position of the rigid-body and the intersection point on the display:

$$\delta = \frac{\vec{n}_d \cdot (\vec{p}_{rb} - \vec{o}_d)}{\vec{n}_d \cdot (-\hat{\vec{d}})}. \tag{4}$$

Here, $\hat{\vec{d}}$ is the normalised vector $\vec{d}$, $\vec{o}_d \in \mathbb{R}^3$ is the physical origin of the display, for which we use the bottom left corner, and $\vec{p}_{rb} \in \mathbb{R}^3$ is the current position of the rigid body. Using the distance δ, we compute the intersection point $\vec{s}$:

$$\vec{s} = (\vec{p}_{rb} + \delta \cdot \hat{\vec{d}}) - \vec{o}_d \in \mathbb{R}^3. \tag{5}$$

In order to deal with different resolutions at the two locations, we convert the intersection point into relative screen-space coordinates:

$$\hat{\vec{s}}(x,y) \in \mathbb{R}^2 : x = \frac{\left(\hat{\vec{r}}_d \cdot \vec{s}\right)}{w}, y = \frac{\left(\hat{\vec{u}}_d \cdot \vec{s}\right)}{h}, \quad \text{with } x,y \in [0,1]. \tag{6}$$

Here, h and w denote the height and width of the display, $\hat{\vec{u}}_d$ the normalised vector in the up-direction of the display and $\hat{\vec{r}}_d$ the normalised vector in the right-direction. The latter two are determined during the calibration step.

Foveated region

Our foveated region is a rectangle that is spanned by four vectors $\vec{c}_i, i \in \{1,2,3,4\}$, starting from the position of the rigid body. Therefore, we compute the intersections with the display as described in the previous paragraph. We use the horizontal angle α and vertical angle β in order to compute the perspective projection. First, the current up vector $\vec{u}$ and right vector $\vec{r}$ of the rigid body are computed, based on the up- and right-vectors of the display as specified in the calibration step:

$$\vec{u} = \tilde{Q}_{rb} \cdot Q_{rb}^{-1} \cdot \begin{pmatrix} 0 \\ \hat{u}_d \end{pmatrix} \in \mathbb{R}^3$$
$$\vec{r} = \tilde{Q}_{rb} \cdot Q_{rb}^{-1} \cdot \begin{pmatrix} 0 \\ \hat{\vec{r}}_d \end{pmatrix} \in \mathbb{R}^3.$$

Half of the horizontal and vertical extents of the rectangle in normalised viewing directions are computed as follows:

$$\delta_\alpha = \tan\left(\frac{\alpha \cdot \pi}{180}\right)$$
$$\delta_\beta = \tan\left(\frac{\beta \cdot \pi}{180}\right).$$

The vectors $\vec{c}_i$ spanning the rectangle can then be computed with the normalised vectors $\hat{\vec{d}}$, $\hat{\vec{u}}$ and $\hat{\vec{r}}$:

$$\vec{c}_i = \hat{\vec{d}} \pm (\delta_\alpha \cdot \hat{\vec{r}}) \pm (\delta_\beta \cdot \hat{\vec{u}}) \in \mathbb{R}^3. \quad (7)$$

Using Eqs. 4, 5 and 6, the relative intersection coordinates, $\hat{\vec{s}}_i \in \mathbb{R}^2$, of the vectors with the display can be computed. If all points are outside of the interval $[0,1]$, their coordinates will be changed to inf, otherwise points will be clamped to the interval. We use 25° for the angles α and β, in order to get a very conservative estimate of the bounding box surrounding the macula of an average human.

5.2 Screen capturing

In order to gain fast access to the full desktop image as a texture we use the Desktop Duplication API. It provides applications with a BGRA texture whenever part of the displayed desktop is changed. Since NVENC expects RGB input to be in the ARGB colour format we use a compute shader to perform the necessary conversion. This shader also handles the optional rotation of the texture, since the Desktop Duplication API handles display rotations by rotating the content of the texture and not the texture itself. For example a display with a resolution of 4096×1200 in portrait mode outputs a texture with the resolution 4096×1200 and the content rotated by 90°, instead of the expected 1200×4096 texture. Technically, the rotation in the shader is implemented without actually moving pixels, but by rotating the texture coordinates. Since a single desktop image might exceed the maximum resolution of the encoder (4096×4096 for Maxwell GPUs) we also implemented downscaling, using bilinear interpolation, and dividing the texture into equally sized tiles by using a texture array. The shader outputs either a texture using the ARGB colour format containing the full display content after optionally downscaling and rotating, or a texture array using the ARGB colour format containing the tiled display content after an optional rotation. In both cases the resulting texture (array) is passed to the hardware encoder directly.

5.3 Encode

We used the NVIDIA Video Codec SDK (NVENC), which provides access to dedicated hardware encoders. It allows us to asynchronously encode the captured frames without piping data that is already on the GPU through main memory. For GTX, RTX and Titan cards the number of parallel encoder sessions is limited to two, while it is unlimited for Quadro type cards. NVENC uses Direct3D 2D interop textures, which provide CUDA and Direct3D access to the same underlying GPU memory. This allows us to avoid an additional copy of the frame after the colour conversion and optional transformation of the frame (see Sect. 5.2). The encoder works with a ring buffer, each element contains one Direct3D 2D interop texture and an output buffer that will contain the encoded frame. The compute shader, described in Sect. 5.2, directly uses the input texture of the element in the ring buffer as its output. Encoding happens in-place and the encoder signals an event for each element whenever a frame is encoded. We insert a key frame regularly in order to avoid artefacts from intra frames that can occur from sudden movements or new objects appearing. In case the texture

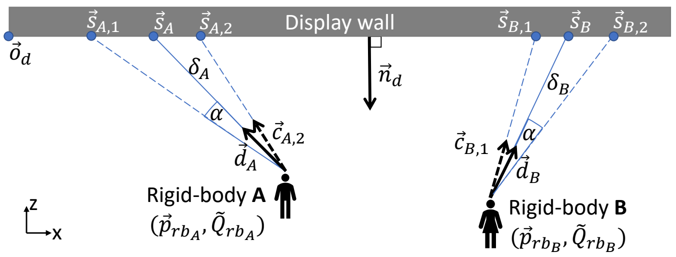


Fig. 3: Schematic illustration of our tracking with two rigid-bodies A and B, depicted in the xz-plane. First, the intersection $\vec{s}$ of the view direction $\vec{d}$ with the display wall is calculated based on position and orientation of the rigid body. In a second step, the corner vectors c_i of the foveated region are determined using the angle α, and used to calculate the intersection points s_i with the display wall.

is divided into smaller tiles we use one encoder session for each tile, if that is possible. When the maximum number of sessions is too limited, a single session can encode multiple tiles, but this increases latency and in addition the required bandwidth, since each frame must be encoded as a key frame. Therefore it is recommended to use downscaling in this case.

For the foveated encoding we change the quality parameter of each macroblock, which has a size of 16×16 pixels, based on its distance from the centre of the foveated regions. This quality parameter can be set to integral values between 51 and 1, with 51 resulting in the lowest quality and 1 in the best quality (cf. Sect. 4). With a quality parameter of 51 the encoder computes the average colour for each macroblock and with a quality parameter of 1 the compression is basically turned off. Note that the impact of this parameter has been investigated closely in previous work (e.g., Kourtis et al. [17] demonstrate the correlation to SSIM). Based on the overall size of the tiled image and the position of each machine, we compute the rectangle of each macroblock and its centre. Then two maps are created, each contains the offset for the quality parameter of every macroblock. This allows us to use double buffering, i. e. updating the parameters for the next frame while we encode the current frame using the current parameters. Both are initialised with the value 0 for every macroblock. Every time foveated regions are received the intersection between the regions and the rectangles of the macroblocks is computed. For all macroblocks inside a region the distance d between the two centres is computed and scaled to the interval $[0,1]$, see Equation 1. Using Equation 2 we compute the desired quality parameter QP for every macroblock. To get the offset O, which is stored inside the map, we compute the difference to the lowest quality parameter: $O = QP - 51$. NVENC uses this map to change the quality parameter for each macroblock from the constant value, 51 in our case, to the desired one.

5.4 Network

We use an I/O completion port (IOCP) [21] to process all send and receive operations of the communication between sides asynchronously. An IOCP is a queue for completion events of asynchronous I/O operations managed by the operating system. To process the completion events, a thread pool is created, which uses the native parallelism of the underlying hardware to determine the number of threads. With all internet communication in one place we can pool the memory in chunks of the largest UDP packet the system can receive, which is 64 kb, in order to avoid costly heap (re-)allocations.

For the side-local communication we assign a single role, or a combination of roles, to each node in the local cluster, based on the user-provided configuration. Supported roles are *provider* and *receiver*. Providers are nodes that capture frames, generate tracking data or relay data. Receivers are nodes that receive encoded frames or tracking data and display it. When the software is started, all local nodes exchange their roles, the data they provide as well as the data they are interested in, resulting in a map that allows for optimising the communication in the way that only the required point-to-point transfers are initiated.

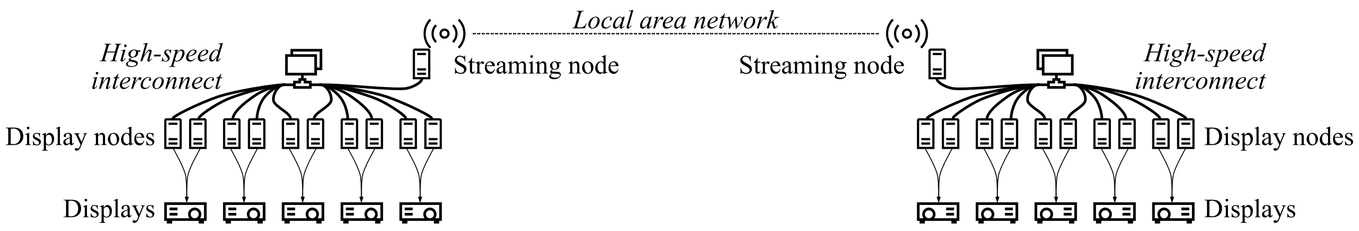


Fig. 4: Hardware setup used in our measurements of throughput and latency of foveated and non-foveated encoding. The two streaming nodes are connected using a 10-Gbit/s-Ethernet network adapter while the display nodes are part of the low-latency InfiniBand network.

All send and receive operations are asynchronous and non-blocking calls. We coalesce all data that requires sending while the previous send operation is not completed and then send it all at once, which has the benefit of increasing the network throughput. Again we pool the memory, in order to avoid heap allocations.

5.5 Decode

The video decoder (NVDEC) works similar to the encoder and is initialised on-the-fly as it needs the settings of the encoder. It uses a separate internal ring buffer on the GPU which contains the encoded frames as well as the decoded frames when decoding finishes. Once the the frames have been fully assembled they are queued for decoding, triggering a chain of callbacks. The last callback signals that the decoding is finished and contains the pointer to the CUDA GPU memory where the frame is located. We copy the frame into a separate ring buffer that holds the frame until it was displayed. This display queue contains Direct3D interop textures again so we can copy directly from the CUDA memory to the texture that is later displayed. In case the texture is divided into smaller tiles we use one decoder for each tile and the final frame is created by copying the tiles to their respective position.

5.6 Display

We use Direct3D for rendering and all scaling and positioning transformations are performed on the fly on the GPU. This also includes clipping the frame to match the segment of the tiled display the respective node is responsible for. As the decoder produces frames in the NV12 colour format we perform colour conversion to RGBA directly in the pixel shader in order to avoid further copies or shader calls. Again, since the decoded frames already reside on the GPU, we avoid transfers between GPU and main memory.

6 Results

We performed a quantitative evaluation of our system by measuring the latency and throughput required to share the content of a tiled display with the resolution of 10800×4096 in a local area network setting, with and without using foveated encoding. The local setting provides a controlled network environment and it allows for visual inspection of latency as it transmits from the right to the left stereo channel of the tiled display. We used the same camera trajectory for visualizing a molecular dynamics simulation, which was shown twice, for all tests discussed in the remainder of this section.

Evaluation focus We particularly address these questions:

- Does the foveated encoding have a negative effect on the encoding and decoding latency?
- Do different foveated intervals have an impact on the latency?
- What is the impact of the optional downscaling or tiling of the input image on the latency ?
- By how much can the required bandwidth be reduced when using foveated encoding with different intervals as well as single and multiple users?

Hardware details The display nodes that drive the tiled display are each equipped with an NVIDIA Quadro M6000, two Xeon E5-2640v3 CPUs and 256 GB of RAM. Each of them is responsible for a tile of 1200×4096 pixels of the overall screen. The two streaming nodes are part of the low-latency InfiniBand network that connects the display nodes and are also connected to each other with a 10 Gb network adapter. Both are equipped with an Intel Core i7-6850K, 64 GB of RAM and a GeForce GTX 1060. All machines, the display nodes and the two streaming nodes, run on Windows Server 2016. Fig. 4 shows a schematic overview of the hardware setup we used for the evaluation. The left half shows the server side, while the right half shows the client side. They are connected to each other via the 10 Gb network adapters of the streaming nodes.

Test scenarios For the non-foveated encoding tests we created three encoder settings: low, medium and high. They have the same settings as for the foveated encoding but a different constant quality parameter. The low setting uses a value of 51, the medium setting a value of 31 and the high setting a value of 11. This corresponds to the upper and lower value of the interval used by the foveated encoding and the medium value is in the middle.

We tested the non-foveated encoding three times, once for each encoder setting applied uniformly to the whole image, and measured the latency of the encoding and decoding operations as well as the throughput. Additionally, we tested the impact of the tiling on the latency and the throughput for the medium setting. The foveated encoding was tested in multiple scenarios. First, we tested the impact on the bandwidth and latency with no look-at rectangle present. Second, we tested the impact on the bandwidth and latency by tracking one user and two users that observed the visualisation simultaneously. Third, we tested the impact on the bandwidth and latency, as well as the visual quality, by using the lower-quality interval $[31, 51]$ for the foveated region instead of the default $[11, 51]$. For all tests we captured 60 fps.

We measured the duration of each individual operation, e. g. encoding, and stored it in a vector. We then periodically, i. e. after every 2000th measurement, computed the minimum, maximum, average, and median duration and the median absolute deviation from the values in the vector and stored them in a file. For all scenarios, we used the algorithm by Cristian [6] to synchronise the clocks of the nodes to an external time source.

Test results Fig. 5 depicts the measured latencies, in milliseconds, for the major steps of the non-foveated and foveated tests. The plotted values depict the worst case measured across all nodes, with the variance between the nodes being about 2 ms throughout. The coloured horizontal lines represent the median values for the different configurations, while the coloured bars represent the median absolute deviation. Minimum and maximum values are indicated by the grey lines. The latency introduced by steps like the reordering of network messages, the assembly of frames and the MPI communication are not shown individually in the graph (their combined impact is well below 1 ms). However, they are included in the *Complete (Source)* and *Complete (Display)* columns. *Complete (Source)* represents the complete duration from capturing a frame to sending it, while *Complete (Display)* represents the complete duration from receiving a frame to displaying it. We also do not show the network latency as it is around 0.1 ms due

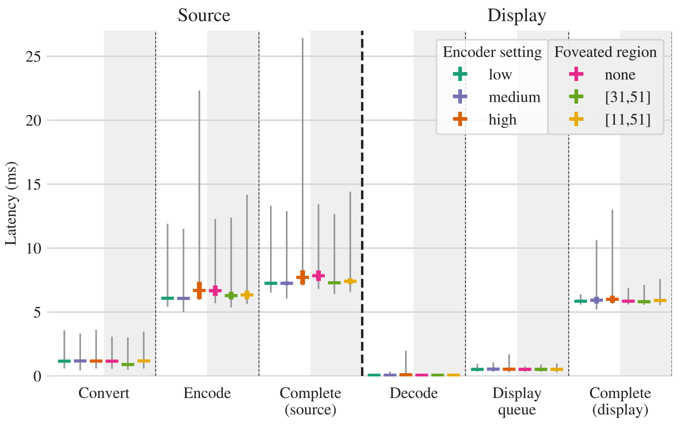


Fig. 5: Overview of measured latencies for major steps of our system. The underlying values are the worst case latencies measured across all nodes, with the variance of other nodes below 2 ms. The plot shows the median values (coloured horizontal lines) and the median absolute deviation indicated as coloured bars. The minimum and maximum latencies are indicated by grey lines. *Complete (Source)* shows the complete duration from capturing the frame to sending it. Likewise, *Complete (Display)* depicts the complete duration from receiving the encoded frame to displaying it. Overall, it shows that there is only very little difference in the encoding and decoding latency between the foveated and the non-foveated case.

to the fact that we use a local area network. The *Display queue* column shows the duration of copying the decoded frame into the display queue as well as the time spent in the queue until the frame is displayed.

The major impact on the overall latency is the encoding with a median of around 6 ms and a maximum of 13 ms for the low and medium settings. For the high settings the median increases slightly while the maximum is 22 ms. The median value corresponds to the encoding latency for intra frames while the maximum is the latency of encoding key frames. The conversion and rotation of the frame takes around 1 ms with the maximum being 3 ms (if we do not perform any downscaling). Downscaling has only a minimal impact, it increases the time to convert the frame by about 1 ms, while scaling it from 1200×4096 to 600×2048. On the display side, the introduced latency is negligible as all operations, i. e. reordering, assembly, decoding, copying into the display queue, and waiting for the frame to be displayed only introduce a latency around or below 1 ms. The only exception is accessing the last decoded frame from NVDEC, which takes 5 ms for the low setting, or more for the medium and high settings, and is responsible for the majority of the introduced latency on the display side.

For the foveated encoding the latencies for one and two users where nearly identical, therefore we only added the latency values for the single user test. For this test, the observer never looked at one spot for long and tried to cover all areas of the screen by walking in a random pattern and also moving closer and further away from the display in order to cover different-sized foveated regions. In case no foveated region is present, the latency values do not change compared to the non-fovated encoding using the low settings. If there is a region present, the latency increases slightly, depending on the size of the region and on the interval. For the $[31,51]$ interval the latencies are lower than for the default $[11,51]$ interval. Overall, it shows that there is only very little difference in the encoding and decoding latency between the foveated and the non-foveated case.

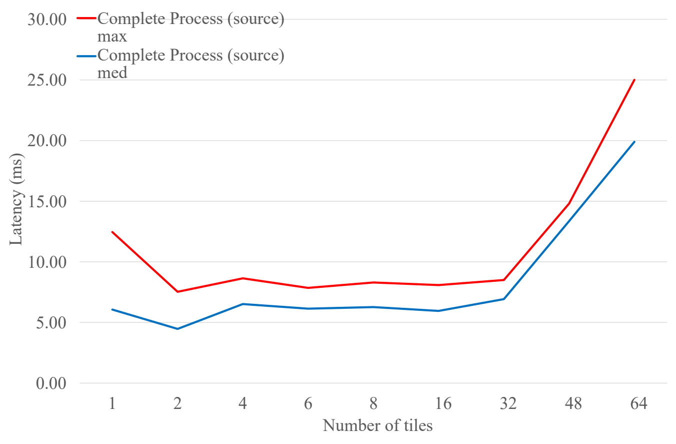
Tiling does not increase the time it takes to convert the frame, but it decreases the latency for the encoding of each tile and, since they are encoded in parallel, the overall encoding latency. This can be seen in Fig. 6 that shows the *Complete (Source)* and *Complete (Display)* latencies for the medium setting and different numbers of tiles. *Complete (Source)* increases if there are more than 32 tiles, because of the number of encoder sessions and the fact that each tile is downloaded separately through PCIe. For 2 to 32 tiles, the maximum latency decreases from 13 ms to about 8 ms because the encoder is able to handle the number of sessions concurrently and because each tile is smaller. *Complete (Display)* is largely unaffected but increases for more than 16 tiles.

In addition to the latency, we also measured the throughput for the three non-foveated tests, as well as multiple foveated tests. The aggregated measured throughput of all nodes for the non-foveated tests can be seen in Fig. 7 together with two foveated tests. The first foveated test uses the interval $[11,51]$ and a single user, while the second uses the interval $[31,51]$. The maximum measured throughput for the non-foveated high encoder settings is about 2 Gb/s, for the medium settings it is 540 Mb/s and for the low settings about 60 Mb/s. We tested the impact of the tiling on the throughput with two configurations. The first uses four tiles and four NVENC sessions, while the second one again uses four tiles but only two NVENC sessions. As expected, the first did not have any impact on the throughput, since the size of the data was roughly the same and the size of the header of the network message is negligible. As discussed in Sect. 5.3 the second configuration only uses key frames and therefore increases the measured throughput from 540 Mb/s to 1.3 Gb/s for the medium setting. The first foveated test yielded roughly the same throughput as the non-foveated low setting, with multiple smaller peaks to 85 Mb/s and a larger peak to 131 Mb/s. We attribute these peaks to the fact that the users stood further away from the screen and therefore the foveated region was larger. Using the interval $[31,51]$ reduced the measured throughput by about 10 Mb/s compared to the $[11,51]$ interval, while covering the same region, although the maximum peak is 150 Mb/s. Again this can be attributed to the fact that the foveated region was larger because the user stood further away from the screen. For two users and the interval $[11,51]$ the measured throughput was slightly higher but on average below 70 Mb/s with the occasional peaks. We repeated the tests three times limiting the bandwidth of the connection between the server and client side to 5, 2.5 and 1 Gb/s. The measured throughput was unaffected and remained the same, except for the high setting which did not work with the bandwidth limited to 1 Gb/s.

7 Discussion

During testing, we noticed that both downscaling and tiling reduce the latency of the encoding. For the downscaling this was expected since the resolution was reduced and therefore the encoding has to be faster. Also the downscaling did not increase the latency for the conversion step tremendously even though it increases the number of texture accesses by four. The reduction for the tiling was not expected, since the encoder still encodes the same amount of pixels and downloads each encoded tile separately. Again the encoding of each tile has to be faster, as they are smaller than the full frame, but we expected the transfer to have more of an impact. As it turned out, increasing the number of tiles to two reduces the latency significantly. Between two and 32 tiles the latency is lower than without using tiling. We assume that at that point the context switch of the encoder and the time to download each tile outweigh the reduced latency of the encoding. For more than 32 tiles the latency increases significantly for each additional tile, reaching over 25 ms for 64 tiles. This latency reduction mainly depends on the resolution. Since our display nodes, with a resolution of 1200×4096, are already close to the maximum of 4096×4096 the impact is higher. The benefit shrinks significantly for lower resolutions and vanishes if the number of NVENC sessions is lower than the number of tiles. In this case the latency increases since each session has to encode multiple tiles and always uses key frames, increasing the latency further. This also negatively impacts the measured throughput, as seen in the test for the non-foveated medium quality setting using four tiles and two NVENC sessions, increasing it by a factor of 2.4. We therefore recommend that the number of tiles never exceed the number of NVENC sessions, as this removes any benefit from using the foveated encoding. We achieved the best latency by using two tiles with 1200×2048 pixels for each of the ten display nodes. This reduced the median overall latency, i. e. the time from capturing to displaying the frame, to 11 ms and the maximum to 19 ms, resulting in the display side being one frame behind the source side.

Latency across all scenarios was hardly noticeable in side-to-side



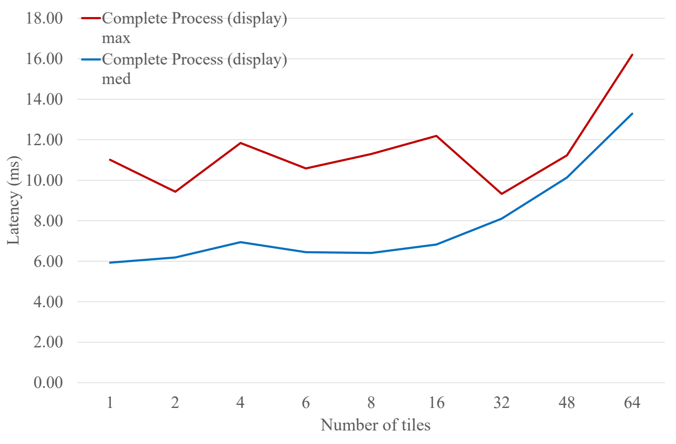
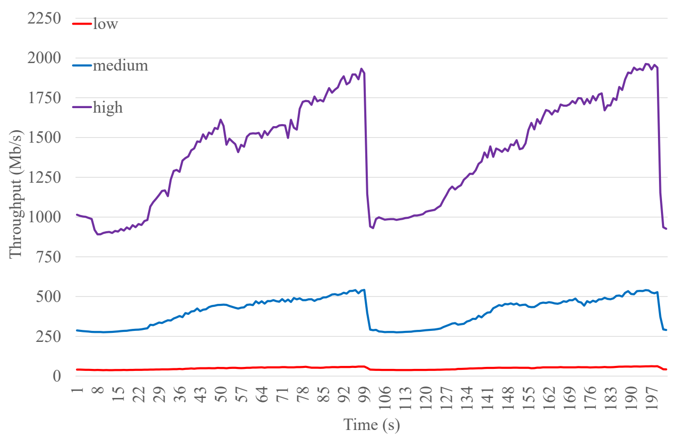


Fig. 6: The impact on the *Complete (Source)* latency by tiling the frame into multiple smaller tiles is shown on the left. For one tile the maximum (max) latency is 13 ms, it shrinks to 8 ms between 2 and 32 tiles and increases to more than 15 ms for more than 48 tiles. The impact on the *Complete (Display)* latency by tiling the frame into multiple smaller tiles is shown on the right. Between 1 and 16 tiles the median (med) latency is largely unaffected, it increases strongly for more than 16 tiles.

comparison, as it was below 30 ms for the whole system, from capturing to displaying the frame. The major contribution to the overall latency, in the non-foveated cases, with up to 22 ms on the source side, comes from the NVENC. For the foveated encoding the latency ranges between the low setting, in case no foveated region is present, and the high setting, in case the region covers the whole display. During our tests we never encountered the latter and latency was mostly around 7 ms and at most 15 ms. Using tiling this latency can be reduced further to at most 8 ms. The latency for the acquisition, transmission and processing of the foveated regions was between 2 and 4 ms, taking the round trip latency to a maximum of 32 ms and 17 ms in the median case.

In our testing the measured throughput for the foveated encoding never exceeded 200 Mb/s, with the highest measured peak at 150 Mb/s. The same can be said for multiple users, they did not increase the measured throughput by much. Therefore the foveated encoding requires on average between 10 and 20 Mb/s more than the low setting, except for situations when users want to get a complete overview over the visualisation, see the peaks in Fig. 7. Similar to the latency the upper and lower limit of the throughput are given by the interval of the quality parameter. The high quality parameter set the upper limit, in our test 2 Gb/s, and the low quality the lower limit, 60 Mb/s. By adapting the interval the upper limit can be reduced further to fit the maximum available bandwidth. During our tests we noticed that there was hardly any visible difference between the medium and the high encoding setting and therefore added the second interval [31, 51] to the tests to show the possible savings. All in all the foveated encoding approach uses, on average, 14 percent of the measured throughput required for the non-foveated medium encoder settings (4 percent compared to high encoder settings), while providing the same quality locally, i. e. where users look at.

As can be seen in Fig. 8 at the example of a parallel coordinates visualisation, the foveated region visibly increases the quality locally. Fine lines that are hardly visible using the lowest encoding quality are clearly defined while using roughly the same throughput. Also we could not visually identify that the rest of the image was encoded with the lowest possible quality since our very conservatively-sized foveated region covered the macula of an average human perfectly. The high precision of the tracking system detected very small movements, which introduced a distracting flickering since the foveated region moved as well, changing the quality parameters for every frame. We introduced a small threshold to filter out little movements and to get rid of the flickering. If the difference between the previous top left corner and the current top left corner of the look-at rectangle is smaller than one centimetre the rectangle is dropped.



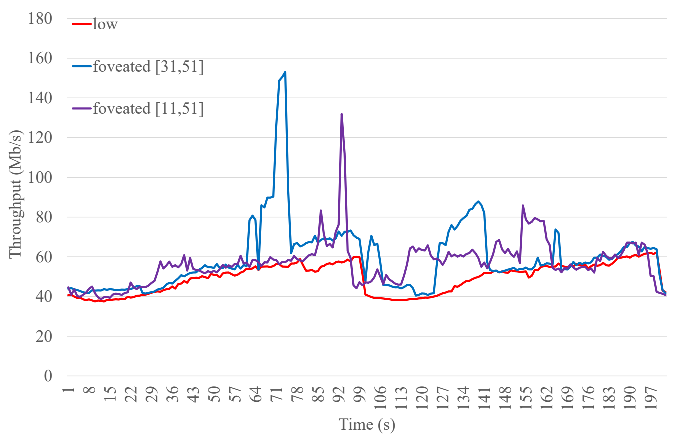


Fig. 7: Aggregated measured throughput of all nodes for the three non-foveated tests are shown on the left side, while the two foveated tests and the context-only test using the low setting are shown on the right. Both foveated tests use a static region and the intervals [31, 51] and [11, 51] show that the first interval requires less bandwidth since the best quality setting is lower. The peaks for both foveated tests can be attributed to the user standing further from the display, leading to a bigger foveated region so there are more macroblocks using a higher quality.

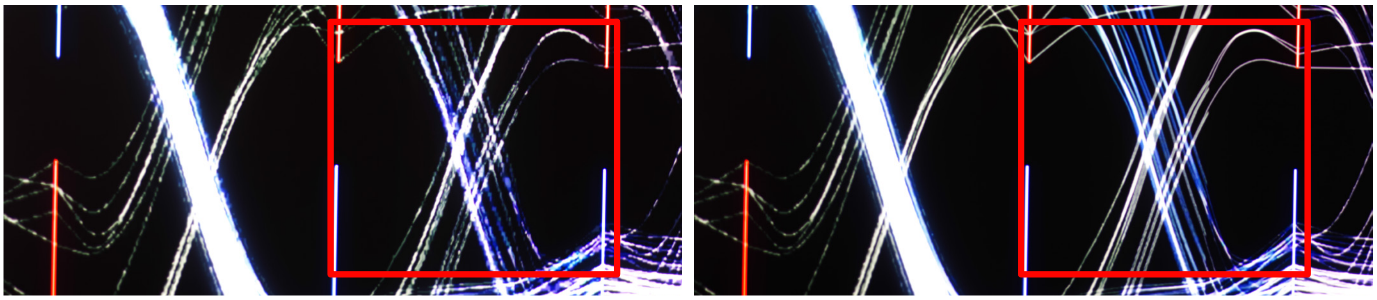


Fig. 8: Two photos that show the difference between no foveated region (left) and the foveated region(right) at the example of parallel coordinates. With the lowest quality parameter (51) the lines on the left photo are not clearly visible, sometimes not at all. The same is true outside of the foveated region, but the lines inside it are clearly visible. The two red rectangles cover part of the foveated region and highlight the differences.

Since the large high-resolution display we developed for runs on Microsoft Windows it was a natural choice to use respective APIs and SDKs. However there are replacement options for the screen capturing and encoding, namely the NVIDIA Capture SDK [23] (now deprecated for Windows), that also work for Linux-based systems. As NVDEC also works for Linux, porting the whole system is a mere implementation effort which only replaces the screen capturing and encoding part.

8 Conclusion and Future Work

We presented a system that reduces the bandwidth needed for remote visualisation on large high-resolution displays. Our approach uses foveated regions, encoding parts of the image inside these regions with a higher quality than the rest of the image. This allows to reduce the required bandwidth significantly while preserving high quality locally in regions that (multiple) users actually focus on. As a result, our approach is viable even if the available bandwidth is well below one Gigabit.

In the future, we plan to add progressive encoding to the system to improve image quality while its original content remains the same. For this, we would gradually increase the quality of the encoding for all macroblocks starting with the foveated regions. We also want to investigate the quality parameter interval used for the foveated encoding, in order to reduce the required bandwidth as much as possible. For this, the quality parameter for the high quality could be reduced gradually until a difference in quality to the original image becomes visible. Additionally, the high quality constant parameter of the interval could be increased based on the distance of the user to the display. While this would result in reduced quality in the foveal region, the perceived result might not be impacted.

Acknowledgments

This work was partially funded by Deutsche Forschungsgemeinschaft (DFG) as part of SFB-TRR 161 (Project ID 251654672) as well as part of Software Sustainability for the Open-Source Particle Visualization Framework MegaMol (Project ID 391302154).

References

[1] T. Biedert, P. Messmer, T. Fogal, and C. Garth. Hardware-accelerated multi-tile streaming for realtime remote visualization. In *Proc. EGPGV*, pp. 33–43, 2018.

[2] M. R. Bolin and G. W. Meyer. A frequency based ray tracer. In *Proc. ACM SIGGRAPH*, pp. 409–418, 1995.

[3] A. Bringmann, S. Syrbe, K. Görner, J. Kacza, M. Francke, P. Wiedemann, and A. Reichenbach. The primate fovea: Structure, function and development. *Prog. Retin. Eye Res.*, 2018.

[4] V. Bruder, C. Schulz, R. Bauer, S. Frey, D. Weiskopf, and T. Ertl. Voronoi-based foveated volume rendering. In *EuroVis (Short Papers)*, pp. 67–71, 2019.

[5] Z. Chen and C. Guillemot. Perceptually-friendly h.264/avc video coding based on foveated just-noticeable-distortion model. *IEEE Trans. Circuits Syst. Video Technol.*, pp. 806–819, 2010.

[6] F. Cristian. Probabilistic clock synchronization. *Distrib. Comput.*, p. 146–158, 1989.

[7] J. Diepstraten, M. Gorke, and T. Ertl. Remote line rendering for mobile devices. In *Comput. Graph. Int.*, pp. 454–461, 2004.

[8] K. U. Doerr and F. Kuester. CGLX: A scalable, high-performance visualization framework for networked display environments. *IEEE Trans. Visual Comput. Graphics*, pp. 320–332, 2011.

[9] S. Eilemann, D. Steiner, and R. Pajarola. Equalizer 2.0 – convergence of a parallel rendering framework. *IEEE Trans. Visual Comput. Graphics*, pp. 1292–1307, 2018.

[10] A. Febretti, A. Nishimoto, V. Mateevitsi, L. Renambot, A. Johnson, and J. Leigh. Omegalib: A multi-view application framework for hybrid reality display environments. In *Proc. IEEE VR*, pp. 9–14, 2014.

[11] S. Frey, F. Sadlo, and T. Ertl. Balanced sampling and compression for remote visualization. In *ACM SIGGRAPH Asia*, p. 1, 2015.

[12] F. Frieß, M. Landwehr, V. Bruder, S. Frey, and T. Ertl. Adaptive encoder settings for interactive remote visualisation on high-resolution displays. In *Proc. IEEE LDAV*, pp. 87–91, 2018.

[13] B. Guenter, M. Finch, S. Drucker, D. Tan, and J. Snyder. Foveated 3D graphics. *ACM Trans. Graph.*, p. 164, 2012.

[14] R. Herzog, S. Kinuwaki, K. Myszkowski, and H.-P. Seidel. Render2mpeg: A perception-based framework towards integrating rendering and video compression. In *Comput. Graph. Forum*, pp. 183–192, 2008.

[15] G. Illahi, M. Siekkinen, and E. Masala. Foveated video streaming for cloud gaming. In *IEEE Int. Workshop on Multimedia Signal Proc.*, pp. 1–6, 2017.

[16] D. Koller, M. Turitzin, M. Levoy, M. Tarini, G. Croccia, P. Cignoni, and R. Scopigno. Protected interactive 3d graphics via remote rendering. In *ACM Trans. Graph.*, pp. 695–703, 2004.

[17] M.-A. Kourtis, H. G. Koumaras, and F. Liberal. Reduced-reference video quality assessment using a static video pattern. *J. Electron. Imaging*, p. 043011, 2016.

[18] M. Levoy. Volume rendering by adaptive refinement. *Vis. Comput.*, pp. 2–7, 1990.

[19] T. Marrinan, S. Rizzi, J. A. Insley, L. Long, L. Renambot, and M. E. Papka. Pxstream: Remote visualization for distributed rendering frameworks. In *Proc. IEEE LDAV*, pp. 37–41, 2019.

[20] Microsoft. Desktop Duplication API. Online, last accessed 27/08/2020, 2018. https://docs.microsoft.com/en-us/windows/desktop/direct3ddxgi/desktop-dup-api.

[21] Microsoft. I/O Completion Ports. Online, last accessed 27/08/2020, 2018. https://docs.microsoft.com/en-us/windows/desktop/fileio/i-o-completion-ports.

[22] K. Moreland, D. Lepage, D. Koller, and G. Humphreys. Remote rendering for ultrascale data. In *J. Phys. Conf. Ser.*, p. 012096, 2008.

[23] NVIDIA. NVIDIA Capture SDK. Online, last accessed 27/08/2020. https://developer.nvidia.com/capture-sdk.

[24] NVIDIA. NVIDIA Quadro Sync. Online, last accessed 27/08/2020. https://www.nvidia.com/en-us/design-visualization/solutions/quadro-sync.

[25] NVIDIA. NVIDIA Video Codec SDK. Online, last accessed 27/08/2020. https://developer.nvidia.com/nvidia-video-codec-sdk.

[26] D. Pajak, R. Herzog, E. Eisemann, K. Myszkowski, and H.-P. Seidel. Scalable remote rendering with depth and motion-flow augmented streaming. In *Comput. Graph. Forum*, pp. 415–424, 2011.

[27] K. Ponto, K. Doerr, T. Wypych, J. Kooker, and F. Kuester. CGLXTouch: A multi-user multi-touch approach for ultra-high-resolution collaborative

workspaces. *Future Gener. Comput. Syst.*, pp. 649–656, 2011.

[28] L. Renambot, T. Marrinan, J. Aurisano, A. Nishimoto, V. Mateevitsi, K. Bharadwaj, L. Long, A. Johnson, M. Brown, and J. Leigh. SAGE2: A collaboration portal for scalable resolution displays. *Future Gener. Comput. Syst.*, pp. 296–305, 2016.

[29] L. Renambot, A. Rao, R. Singh, B. Jeong, N. Krishnaprasad, V. Vishwanath, V. Chandrasekhar, N. Schwarz, A. Spale, C. Zhang, G. Goldman, J. Leigh, and A. Johnson. SAGE: the Scalable Adaptive Graphics Environment. *Proc. WACE*, pp. 2004–2009, 2004.

[30] Sanghoon Lee and A. C. Bovik. Fast algorithms for foveated video processing. *IEEE Trans. Circuits Syst. Video Technol.*, pp. 149–162, 2003.

[31] S. Shi and C.-H. Hsu. A survey of interactive remote rendering systems. *ACM Comput. Surv.*, p. 57, 2015.

[32] M. Stengel, S. Grogorick, M. Eisemann, and M. Magnor. Adaptive image-space sampling for gaze-contingent real-time rendering. In *Comput. Graph. Forum*, pp. 129–139, 2016.

[33] H. Strasburger, I. Rentschler, and M. Jüttner. Peripheral vision and pattern recognition: A review. *J. Vis.*, pp. 13–13, 2011.

[34] K. Vaidyanathan, M. Salvi, R. Toth, T. Foley, T. Akenine-Möller, J. Nilsson, J. Munkberg, J. Hasselgren, M. Sugihara, P. Clarberg, et al. Coarse pixel shading. In *Proc. HPG*, pp. 9–18, 2014.

[35] A. Zare, A. Aminlou, M. M. Hannuksela, and M. Gabbouj. Hevc-compliant tile-based streaming of panoramic video for virtual reality applications. In *Proc. ACM Int. Conf. Multimed.*, p. 601–605, 2016.