Shading Atlas Streaming

JOERG H. MUELLER, Graz University of Technology
PHILIP VOGLREITER, Graz University of Technology
MARK DOKTER, Graz University of Technology
THOMAS NEFF, Graz University of Technology
MINA MAKAR, Qualcomm Technologies Inc.
MARKUS STEINBERGER, Graz University of Technology
DIETER SCHMALSTIEG, Graz University of Technology and Qualcomm Technologies Inc.

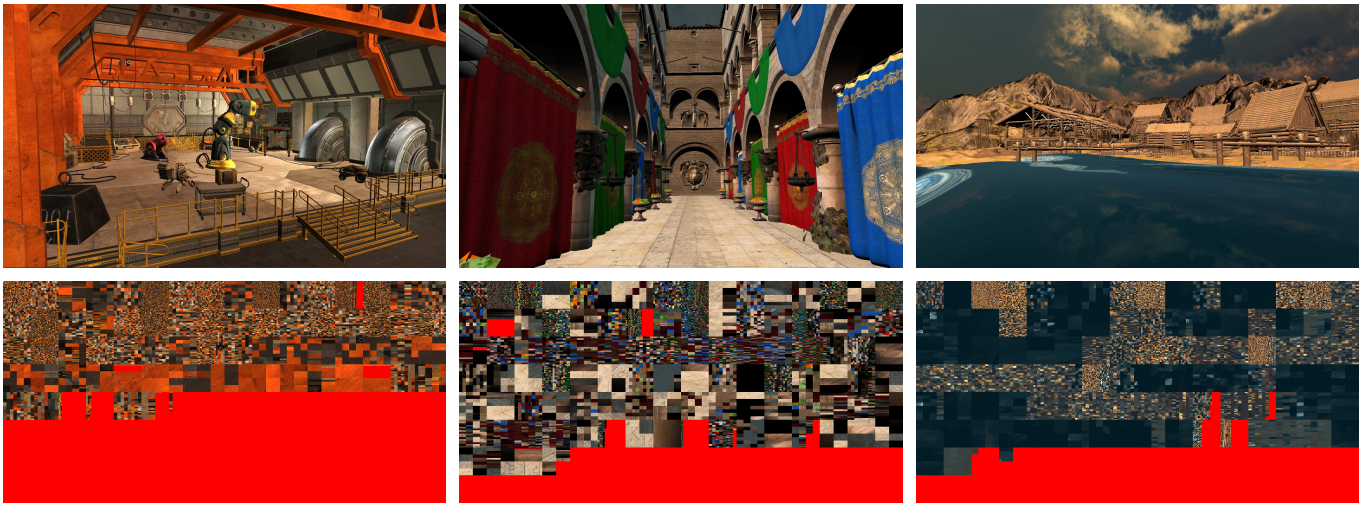


Fig. 1. Game scenes (top row) with corresponding shading atlas (bottom row). The shading atlas contains all the shading information of the visible surfaces corresponding to the rendered scenes. The object-space parametrization is created fully dynamically. From the shading atlas, novel views at close viewpoints can be rendered for framerate upsampling and warping. The shading atlas is temporally coherent and lends itself to efficient MPEG compression and streaming.

Streaming high quality rendering for virtual reality applications requires minimizing perceived latency. We introduce Shading Atlas Streaming (SAS), a novel object-space rendering framework suitable for streaming virtual reality content. SAS decouples server-side shading from client-side rendering, allowing the client to perform framerate upsampling and latency compensation autonomously for short periods of time. The shading information created by the server in object space is temporally coherent and can be efficiently compressed using standard MPEG encoding. Our results show that SAS compares favorably to previous methods for remote image-based rendering in terms of image quality and network bandwidth efficiency. SAS allows highly efficient parallel allocation in a virtualized-texture-like memory hierarchy, solving a common efficiency problem of object-space shading. With SAS, untethered virtual reality headsets can benefit from high quality rendering without paying in increased latency.

CCS Concepts: • **Computing methodologies** → **Rendering**; **Virtual reality**;

Additional Key Words and Phrases: Streaming, Shading, Texture atlas, Object-space shading, Virtual reality

ACM Reference format:
Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading Atlas Streaming. 1, 1, Article 1 (May 2018), 16 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

J. Mueller, P. Voglreiter, M. Dokter and T. Neff are affiliated with the Christian Doppler Laboratory of Semantic 3D Vision established at Graz University of Technology.
2018. XXXX-XXXX/2018/5-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The current state of the art to deliver virtual reality (VR) to a consumer audience is to combine a high-end PC with a tethered VR headset. Obviously, both the need for an expensive PC and the tether are undesirable. Untethering the headset gives the user freedom to roam the virtual environment. All-in-one VR headsets, such as Oculus Go, are untethered by design, and some (e.g., VIVE Focus or HoloLens) offer superb inside-out tracking. However, an embedded GPU in an all-in-one-device will not be able to deliver the high-end graphics required for VR in the foreseeable future, since it is limited by power budget, thermal restrictions and production cost.

As an alternative to standalone operation, VR games can be delivered as a streaming service from cloud or edge computing directly to the VR headset, eliminating the need for owning and maintaining an expensive gaming PC. Cloud game services over wide-area networks are commercially offered by vendors such as Sony or NVidia. However, bandwidth and latency of wireless networks are major concerns. Unlike video-on-demand services, which can hide latency and bandwidth variations by heavy buffering, VR games must be instantly responsive [Chen et al. 2011; Manzano et al. 2012]. Even edge servers [Choy et al. 2012] or wireless local area networks [Lee et al. 2015] barely meet the latency and bandwidth requirements of game streaming.

One opportunity for improving the perceived latency in such a streaming scenario is to better utilize the GPU of the all-in-one device, which is only relatively lightly loaded with tasks such as video decoding and displaying a pixel stream. We leverage this opportunity by proposing a *remote rendering* approach, where the graphics pipeline is subdivided across a client-server system such that perceived latency is minimized [Shi and Hsu 2015].

In the search for a good remote rendering solution, it is important to identify natural splits in the rendering pipeline. Current VR systems (including tethered ones, such as Oculus Rift or Sony PSVR) use a two-stage graphics pipeline, where a framebuffer is generated by a rendering stage and *warped* by a display stage. Warping jointly addresses viewpoint extrapolation, spatial upsampling, temporal upsampling, and distortion compensation for headset optics.

However, an intrinsic problem of *image-based rendering* (IBR) methods, such as warping, is that they index by view (i.e., in image space) and therefore lack object-space information needed to detect visibility changes. Hence, viewpoint extrapolation suffers from disocclusion artifacts.

Rendering caches that operate in *object space* [Ragan-Kelley et al. 2011] represent an interesting alternative to warping in image space. Object-space shading is employed in some game engines to amortize shading or texture synthesis cost [Baker 2016; Chen 2015]. However, for large scenes, a global object-space map becomes expensive to maintain [Liktor and Dachsbacher 2012] and prohibitive to stream. A scalable solution for streaming VR must ensure that all relevant aspects - memory consumption, shading load and transmission bandwidth - are output-sensitive, i.e., proportional to the final image display and not to the scene size.

In this paper, we introduce a novel approach for remote rendering, called *shading atlas streaming* (SAS). In SAS, the server GPU fills an atlas with shaded pixels corresponding to just the visible triangles. The client GPU performs a final geometry pass, but samples the shading information from the atlas rather than invoking expensive fragment shaders. SAS makes the following contributions:

- *Potentially visible set approximation.* We combine a visibility pass with motion prediction to approximate a potentially visible set [Airey et al. 1990], or PVS. The atlas is filled with shading information for triangles that have recently been visible or will likely be visible according to the motion prediction. In practice, this corresponds to an online sampling of the PVS, sufficient for preventing almost all disocclusion events, provided the motion prediction is reliable[1].
- *Virtual memory for shading information.* The atlas acts as a cache. This cache is filled incrementally with shaded pixels corresponding to the currently visible triangles. Coupling virtual memory allocation to visibility makes physical memory requirements proportional to the target resolution.
- *Implicit mipmapping.* Allocation sizes in the SAS are derived from the footprint of the triangle in image space, thereby delivering the equivalent of anisotropic mipmapping.
- *Graceful degradation.* Under high memory pressure (for example, during phases of rapid view rotation), the memory stress induced by the enlarged PVS can be alleviated by biasing the mipmap level selection, trading temporarily reduced image quality for disruptive disocclusions. This behavior is akin to dynamic resolution scaling in contemporary games, with the important difference that ours works in object space and can easily support advanced behaviors such as per-object priorities and foveation.
- *Efficient transmission.* Changes to the atlas can be efficiently encoded as incremental video frames, compatible with hardware-accelerated MPEG encoding and transmission. No proprietary transmission layer is necessary.

We illustrate our work with examples drawn from a prototype of an end-to-end system, targeting an all-in-one headset prototype. We show that our approach outperforms conventional video streaming in terms of image quality at the same network bitrate.

2 RELATED WORK

The SAS architecture is related to image-based rendering, remote rendering, object-space shading and virtualized textures. In this section, we discuss previous work in these areas and link them to our research.

2.1 Image-based rendering

IBR synthesizes an image from a novel viewpoint by warping a previous image. Probably the most widely known warping is asynchronous time warping (ATW) popularized by the Oculus Rift [OculusVR 2018]. ATW compensates for the perceived latency during rapid rotational head motion by re-adjusting the displayed viewport inside a slightly overscanned framebuffer. This adjustment happens just prior to display scan-out, based on the most recent tracked (or predicted) head motion. An important limitation of ATW is that it does not support full 3D camera motion.

Full 3D motion can be handled by more advanced warping methods, which use a variety of intermediary representations. Some avoid polygonal geometry and use alternatives such as point splatting [Chen and Williams 1993], layered depth images [Shade et al. 1998] or unstructured lumigraphs [Buehler et al. 2001]. However, the most widely used strategy for real-time warping is to create a geometric proxy from the depth buffer and associate the shading content with the proxy via projective texture mapping [Mark et al. 1997]. Geometric simplification can be used to obtain a more lightweight proxy [Bowles et al. 2012; Didyk et al. 2010; Lee et al. 2015].

[1] A problem which we consider out of scope for our purposes

However, IBR methods relying on the depth buffer as a representation of scene geometry can suffer from the limited resolution of the depth buffer.

2.2 Remote rendering

A good way to classify remote rendering systems is by the characteristics of the transmitted data [Shi and Hsu 2015]. We can coarsely distinguish approaches that transmit only video, systems that transmit precomputed IBR information for static scenes, systems that transmit dynamically computed IBR views with the original scene geometry, and systems that transmit color+depth buffers.

The simplest system category does not send any geometry to the client. While such a minimal system is limited to pure ATW on the client side, the server still can use rendering information for more efficient MPEG encoding [Noimark and Cohen-Or 2003] or proprietary color+depth compression [Pajak et al. 2011].

If a static scene can be assumed, complex scenes can be converted to an IBR database in precomputation. The proxy geometry used to organize the IBR can take on a variety of forms, such as sparse [Teler and Lischinski 2001] or dense [Boos et al. 2016] impostors, view-dependent texture maps [Cohen-Or et al. 1999], or geometry images [Sheng et al. 2011].

For dynamic scenes, we can take advantage of the fact that, in most games, only the appearance changes dynamically, while most of the geometry is static. The static geometry can be used directly for perspective texture mapping, possibly with simplification [Reinert et al. 2016]. Rendering the same model at server and client can reduce transmission to *residual images* remaining after considering disocclusion [Mann and Cohen-Or 1997], low-frequency illumination effects [Levoy 1995], or factoring out expensive indirect illumination effects [Crassin et al. 2015].

Deriving the proxy geometry from the rendered depth buffer makes IBR completely independent of the scene complexity. Data can be transmitted as color+depth images and rendered via splatting or texture-mapping to a meshed depth buffer [Chang and Ger 2002; Shi et al. 2012]. This strategy can be combined with residual image transmission [Bao and Gourlay 2004; Cuervo et al. 2015; Yoon and Neumann 2000] or speculative rendering [Lee et al. 2015].

One implicit assumption of most, if not all, IBR approaches is that real-time geometric rendering is inherently expensive and must be avoided as much as possible. This may have been true 20 years ago, but today even a mobile GPU has a fairly high geometry and rasterization throughput. The real cost of today's rendering workloads lies in complex surface shaders. For scenes of reasonable complexity, the most interesting use of IBR lies in amortizing the shading results. We explore this design point in our work.

2.3 Object-space shading

Shading in object space has recently become a viable alternative to shading in image space [Baker 2016]. While traditional IBR primarily uses image space representations to exploit temporal coherence, object-space shading [Cook et al. 1987] can exploit spatio-temporal coherence in real time [Ragan-Kelley et al. 2011]. Unfortunately, many sophisticated object-space shading ideas are implemented in software rendering [Andersson et al. 2014; Burns et al. 2010; Clarberg et al. 2014, 2013] and would require future GPU extensions for real-time use in VR.

However, some variants of object-space shading can be implemented with shaders using multiple passes. A *visibility stage* determines visible geometry, a *shading stage* updates shading information in the texture, and a *display stage* generates the image to be displayed from the texture. The explicit mapping into object-space makes it straight forward to take advantage of spatio-temporal coherence, for example, by adjusting shading rates. However, memory consumption and shader load are directly dependent on the effectiveness of the visibility pass.

In scenes with limited overdraw, such as in games with a zoomed-out viewpoint, it can be sufficient to allocate a pre-charted texture per object on demand (for each object that is not culled) to store the shading information [Baker 2016]. In scenes with high overdraw, visibility can be determined per triangle using a depth buffer. For example, texel shading [Hillesland and Yang 2016] shades only the visible portion of the scene, but organizes the shaded pixels using pre-charted, mipmapped textures per object. Large portions of the allocated memory remain empty, making texel shading unsuitable for video streaming. Another example, the compact G-buffer [Liktor and Dachsbacher 2012], stores only the shaded pixels, but organizes them using a hash map. Hashing is convenient for stochastic rasterization, but not so suitable for video streaming, since it scrambles spatial coherence.

In contrast to previous object-space methods, SAS has both a compact memory footprint and temporal coherence. We fill our object-space texture with shaded information for individual triangles at appropriate sampling rates. The per-triangle sampling rates of our approach are determined similarly to vertex color textures [Yuksel 2017], but the triangle-to-texture mapping is determined using an approach inspired by virtualized textures, as described below.

2.4 Virtualized textures

A memory-efficient representation of shading information usually requires a *texture atlas*, i.e., a one-to-one mapping from all object surfaces into a single texture space [Carr and Hart 2002]. When the amount of surface space to be mapped is too large, it may be necessary to use a virtualized texture (VT). The texture is split into square tiles (pages), of which only a working set is loaded [Chadjas et al. 2010]. For terrain rendering, this working set is implicitly given by the viewing frustum [Tanner et al. 1998]. For general polygonal objects, an *indirection texture* (pagetable) is required [Kraus and Ertl 2002]. Dynamic updates to the working set additionally require a *tile fault* component to determine missing tiles and a *tile fetch* component that loads these tiles [Lefebvre et al. 2004].

Today's GPU architectures expose hardware-accelerated paging for VT [Cebenoyan 2014], albeit only with a single level of indirection and a fixed tile size. Therefore, VT has been mostly used with paging in software, for example, in the well-known id tech 5 engine [van Waveren 2009].

Terrain rendering lends itself to a hierarchical VT implementation, since the terrain can easily be broken up into square chunks. Visibility and level of detail for each chunk can be implicitly determined [Tanner et al. 1998], so dynamic packing of visible terrain

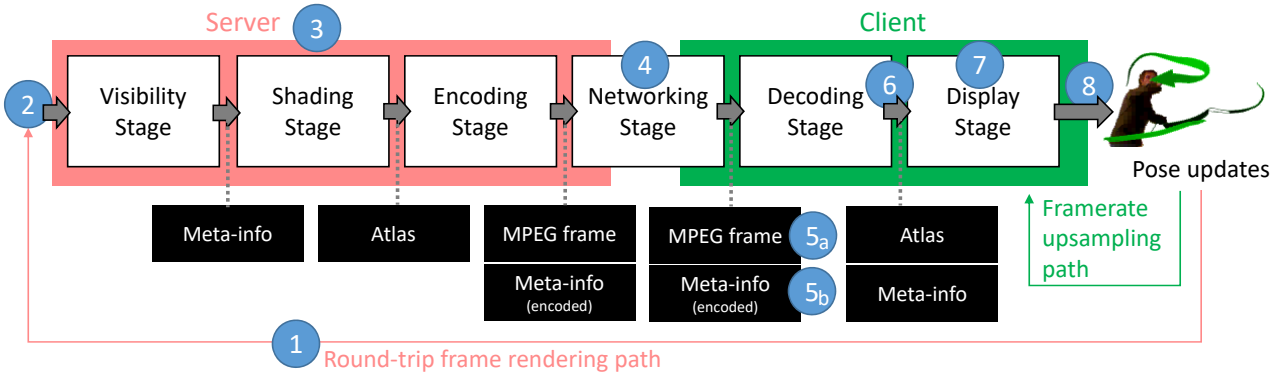


Fig. 2. Our pipeline is split across a server and a client part, with the shading atlas as the central data structure connecting the two. Camera pose updates generated by the user are sent upstream, on a slow (networked) path to the server for rendering new shading into the atlas, and on a fast (direct) path to the client to render new images to the display.

chunks given at multiple resolutions into a single "stitch map" can be computed on the CPU [Baker 2016; Chen 2015; McAnlis 2009].

In contrast, SAS requires a dynamic packing of shading information associated with individual triangles, rather than large chunks or objects. Computing the packing of a large number of entities given at different levels of detail on the CPU would be too slow. We will show how to efficiently allocate a hierarchical VT on the GPU at full frame rate.

3 SYSTEM OVERVIEW

To understand the design constraints of a remote rendering system using object-space shading, we begin with an overview of the server-client pipeline (Figure 2). The end-to-end latency for a round trip from client to server and back can be broken down as follows:

(1) The client sends the current view matrix to the server. This duration is owed to network latency. It can be changed with faster networking technology.

Fig. 3. (top) Image-based rendering methods typically suffer from artifacts caused by disocclusions (red arrow), when the viewpoint is translated. (bottom) SAS can avoid artifacts by predicting future viewpoints and rendering all geometry in the corresponding potentially visible set.

(2) The message waits at the server until a new frame starts rendering. This duration depends on how frequent the client sends pose updates, since the most recently received pose is used for the next frame.
(3) The server renders the atlas frame. This duration corresponds to the processing times for visibility, shading and encoding stages. It depends on server CPU and GPU performance.
(4) Atlas frame and corresponding meta-information are sent to the client. This duration is owed to network latency.
(5) The client decodes (5a) the atlas frame and (5b) the meta-information. This duration corresponds to the maximum of client's processing time for the two decoding tasks, since they are carried out by different hardware units. The duration depends on client CPU and GPU performance.
(6) The received data waits at the client until a new frame starts rendering. This duration depends on the client frame rate and varies between 0 and the client frame time.
(7) The client renders the final frame. This duration corresponds to the client's processing times for the display stage. It depends on the client's CPU and GPU performance.
(8) The final frame is displayed. The most recently received server frame is used to render additional frames at the client (*framerate upsampling*). Thus, this duration increases with every upsampled frame, until a new atlas frame arrives.

The end-to-end latency of a streaming system not only depends on the network latency, but also on the server's frame rate for sending data and the client's frame rate for displaying new images. If the server has a lower framerate than the client, this framerate upsampling induces additional variable latency on top of the network latency. However, lowering the server frame rate has several advantages. First, the server can render higher quality results, as it has more time per frame. Second, a lower server frame rate can be encoded at a lower bitrate. Third, the client has to decode fewer frames, benefiting from faster processing and power savings. We have found that these advantages outweigh the overhead of shading all polygons in a PVS, even when the PVS becomes large.

The system design must balance server and client framerates to deliver optimal quality for a given bitrate. In an optimal pipeline, every stage must be both fast and avoid any loss of image quality. In the remainder of this section, we discuss the design decisions of the various pipeline stages, before returning to our central enabling technology, the shading atlas, in section 4. We report on a system evaluation in section 5.

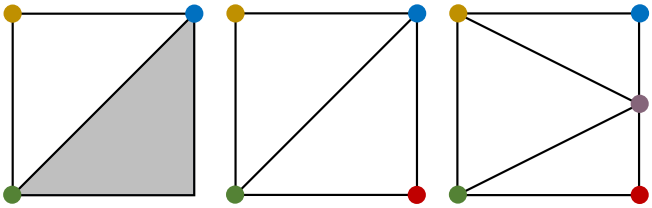
3.1 Visibility stage

The target of the visibility stage is to determine which geometry is visible and needs to be shaded. It addresses the main problem IBR methods have with framerate upsampling – disocclusions. While disocclusions provoked by camera rotation can be avoided with an extended field of view (FOV), disocclusions provoked by camera translation would need additional views of the scene. Unlike most IBR methods, SAS only has disocclusion problems if the disoccluded geometry is not part of the PVS (Figure 3, top). We compute the PVS by predicting the future viewpoints that will be used at the client and include the associated visible geometry into the PVS (Figure 3, bottom).

The unit for determining visibility is a *patch*, i.e., a group of two or three adjacent triangles (Figure 4), which are suitable for dense atlas packing [Carr and Hart 2002]. In section 5.2, we demonstrate that this fine-grained visibility determination is necessary to keep the atlas size small. Assignment of triangles to patches is determined during preprocessing with a heuristic optimization; solo triangles are used only when unavoidable.

The visibility stage renders a patch id buffer, with depth buffer enabled. A subsequent compute pass marks the patches in the id buffer as visible, effectively computing an exact visible set (EVS). To extend the EVS into a PVS covering the geometry that may become visible as the camera moves, we combine two measures (Figure 5):

- We enlarge the FOV at the borders by increasing the resolution of the id pass and adapting the projection matrix to keep the original FOV and resolution in the center of the id buffer. This procedure is typically applied in IBR methods to ensure visibility for small rotations of the view.
- Based on head motion prediction, we extrapolate motion for a short period into the future and sample an EVS for multiple viewpoint locations, starting from the current viewpoint location. The PVS is determined as the union of all patches contained in at least one EVS sample. We heuristically sample the EVS several times, using either prediction based on the visual-inertial tracking system of the client device or simple linear extrapolation (for deterministic testing). We also consider the motion of animated objects corresponding to the predicted time for which an EVS is created. Note that this only applies to deterministic object motion or object motion that can be reliably predicted. We do not consider speculative rendering [Lee et al. 2015].



(a) One-triangle patch (b) Two-triangle patch (c) Three-triangle patch

Fig. 4. Our system supports three different patch types with one, two or three triangles. The preferred patch type is the two-triangle patch.

The success of a sampling approach to PVS depends on the quality of the motion prediction. For short look-ahead periods, we found our simple approach to deliver perceptually correct results. If the PVS computation misses a triangle that would otherwise be visible, the client cannot render this triangle, and a hole may appear. We report in section 5.1 on the correctness of the PVS computation and demonstrate that missed triangles are rare.

Atlas memory management—allocating and deallocating space for patches—is directly derived from the PVS. Newly visible patches are allocated in the atlas; patches that have become invisible are removed from the atlas. Patch sizes are determined based on the projected area of a patch on the screen as detailed in Section 4.3. Details on the atlas memory management are given in Section 4.2. The atlas memory management also writes a vertex buffer containing the visible geometry including corresponding texture coordinates for the shading atlas.

3.2 Shading stage

In the shading stage, visible patches are shaded into the atlas. The number of shader invocations should be proportional to screen size with some overhead factor. Since a vertex buffer containing the visible geometry was created, the shading stage can be implemented as a standard geometry rendering pass with the following characteristics:

- No depth buffer is needed, since triangles do not overlap.
- Instead of screen space positions, the vertex shader needs to output the shading atlas texture coordinates as position.

We designed the shading stage to be able to use a standard geometry pass, because it has important advantages. First, there are no special limitations in terms of materials. Unlike many deferred rendering approaches, no uber-shader is required. Second, object-space rendering methods which perform shading in a compute shader [Hillesland and Yang 2016] must interpolate all attributes in software instead of using hardware interpolation units, and they do not benefit from the free derivatives delivered by hardware quad shading.

3.3 Encoding stage

In the encoding stage, the updates to atlas and meta-information that must be sent to the client are prepared for network transmission. We collect changes to the PVS data structures directly on the GPU during the visibility stage, so that a single readback to the CPU in the encoding stage suffices.

The network load is dominated by the atlas, while the meta-information consumes comparatively little bandwidth. However, the atlas content is temporally coherent, as both the visible triangles and their shading changes only slowly with a moving viewpoint. Therefore, the atlas can be efficiently encoded as an MPEG stream.

Hardware-accelerated MPEG encoding on the GPU is very efficient and offers sufficient throughput for our use case. Therefore, we designed our atlas in an MPEG friendly way, as a single large texture filled with temporally coherent shading loads. MPEG encoders maintain internal state, which makes it expensive to run multiple encoders in parallel. Therefore, storing the shading information as a collection of variable-sized per-object textures [Baker 2016] would be difficult to fit to MPEG encoding.

We optimize our video coding configuration for low complexity and low latency. We encode the texture atlas using the H.264 video coding standard. The compressed frames are small enough so that the readback time to the CPU is negligible. Moreover, readback (and subsequent network transmission) can run in parallel with rendering of the next frame.

The client has no need to know of the memory management associated with the atlas. The client only requires visible triangles with corresponding vertices and texture coordinates pointing into the atlas. Thus, we encode the vertices, triangles, and texture coordinates for the shading atlas as meta-information. Vertices and triangles are only transmitted once, when they first become visible. Texture coordinates are transmitted periodically, whenever patches are re-located inside the atlas. Rigid body animations are supported by tracking per-object transformation matrices. Animations computed in the vertex shader must be transmitted in every frame, provided the vertex is visible. Note that we implemented meta-information transmission to demonstrate an operational end-to-end system. However, meta-information transmission is not yet optimized and not explicitly considered in the system evaluation.

3.4 Networking stage

The network transmission must consider two channels with different characteristics. The atlas is transmitted as an MPEG stream over a potentially lossy channel optimized for throughput. For this channel, we rely on RTP over UDP. Since the UDP connection is prone to packet losses, the video encoder inserts I-frames at regular intervals (typically every five frames) to stop error propagation in the decoded atlas due to a packet loss.

The meta-information (geometry, patches, and animation updates) use a comparatively small bandwidth, but must be transmitted over a reliable channel to ensure that the scene structure at the client is kept intact. We considered using a custom sub-channel of the MPEG stream [Collet et al. 2015] or APP messages of RTCP for this purpose, but ultimately decided for a simple TCP connection, because it trivially ensures reliable transmission.

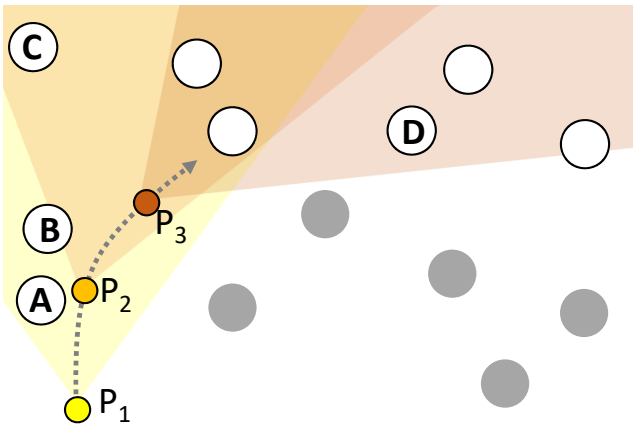


Fig. 5. From the current viewpoint P_1, two future viewpoints, P_2 and P_3 are predicted, with the corresponding FOV shown in color. An EVS is computed for each of the three viewpoints, and the PVS is determined as the union of all objects visible in any EVS (visible objects are marked as white circles). In the example, object A is only visible from P_1, but not from P_2 or P_3. Conversely, object D is only visible from P_3. Object C is jointly occluded by objects A and B from P_1, but becomes visible from P_2.

3.5 Decoding stage

The client decodes received messages immediately upon arrival and updates its internal structures. MPEG decoding on the mobile device is handled by a dedicated hardware unit. The meta-information is decoded on the GPU.

Since the client prototype receives network updates over two separate connections, it must ensure that both streams are synchronized after each frame is received. Decoding and display stages run in multiple threads at different update rates and communicate via triple buffering to avoid any latencies from locked buffers.

3.6 Display stage

The display stage at the client relies on a very simple forward rendering pass. The client issues a single draw call on a vertex buffer that has been generated by processing the incoming server data. This vertex buffer already contains the correct texture coordinates into the atlas, requiring no indirect lookup. Rendering is decoupled from the remaining pipeline; if an update has been received from the server and is ready to be used, the rendering thread simply switches buffers and continues rendering at full speed.

Since we have a direct uv-mapping from vertices to shading atlas, no projective or indirect texture mapping is required, as necessary in other IBR methods. The variable shading resolution baked into the atlas even relieves the client from having to perform mipmapping. The remaining requirements of rendering to a VR headset are taken care of in two sub-stages:

(1) Framerate upsampling uses the latest pose from the head tracker for rendering, while the shading information in the atlas is generated based on an earlier pose. Moreover, a stereo image pair is generated by rendering twice, with a statically calibrated offset for left and right eye. The results of the first sub-stage are stored in a framebuffer object.
(2) We compensate for the barrel distortion induced by the magnifying lenses in the VR headset by applying radial corrections for every pixel on the screen. The distortion is considered separately in each RGB color channel to compensate for the color abberations of the lens system.

4 SHADING ATLAS

An atlas suitable for streaming must meet several requirements:

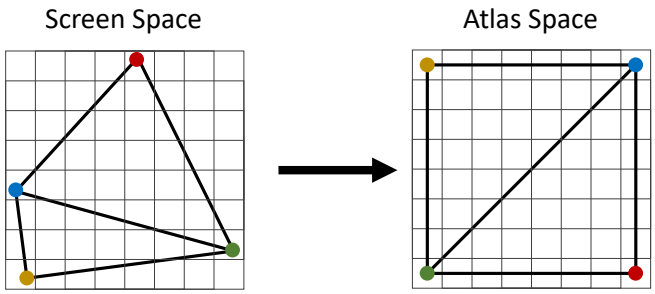


Fig. 6. A patch consisting of two neighboring triangles is mapped from screen space to a block in atlas space. The vertices are inset by half a pixel from the block edges in order to allow bilinear interpolation in the atlas without results being influenced by neighboring blocks.

- *Small footprint*: The total number of pixels must be small enough so a hardware-accelerated encoder can deal with the atlas in real time.
- *Temporal coherence*: The location of shading information corresponding to a particular triangle must change as little as possible from frame to frame, so that efficient MPEG encoding is possible [Collet et al. 2015].
- *Compactness*: Memory fragmentation, i.e., interleaving of occupied and empty areas, makes it more difficult to find places for new memory allocation, especially bigger areas. Moreover, fragmentation reduces video encoding efficiency.
- *Dynamic memory management*: Since we do not know in advance what parts of the scene will become visible and how large a visible triangle will appear on the screen, we cannot precompute any atlas layout or uv-coordinates.
- *Parallel allocation*: Fine-grained memory allocations of variable-size memory chunks for a large number of triangle patches cannot be performed on the CPU at full frame rate. Allocation must happen on the GPU in a highly parallel manner, with as little need for thread synchronization as possible.

We approach this challenging combination of requirements with an atlas based on VT with a three-layer memory hierarchy and a variable tile size (see supplementary material for implementation details). We begin by introducing the data structure and atlas memory hierarchy. Then, we discuss parallel memory allocation. Finally, we explain how to handle situations of high memory pressure. We demonstrate the resulting minimal memory fragmentation in Section 5.4.

4.1 Patches and blocks

We pack one, two or three adjacent triangles into a *patch*, as shown in Figure 4. All atlas operations, such as allocation, deallocation and shading, operate on whole patches. The shading information corresponding to a patch fits into a *block*, as shown in Figure 6. Given the index of a triangle, we can look up the corresponding information in the atlas using a two-step indirection: For each triangle, we store a patch id; each patch contains a record on a block in the atlas.

Blocks have a rectangular, landscape format with each side length corresponding to a power of two. We call the block size a *level* because of its conceptual similarity to a mipmap level. The power-of-two dimensions enable subdivision without leftover space and allow for compact integer addressing using the binary logarithm of the dimension. They also support the video encoder in matching MPEG macroblocks. The disadvantage of this decision is that only a discrete set of block sizes are available and can cause occasional seams between neighboring patches, as they transition between different resolutions. This problem can be minimized by careful construction of the patches.

A half-pixel wide reserve at the border of the block is sufficient when sampling with bilinear filtering. No conservative rasterization [Akenine-Möller and Aila 2005] per triangle, as in other object-space methods [Hillesland and Yang 2016], is required. Our block layout is motivated by similar goals as in the vertex color texture work [Yuksel 2017]. A larger border could be used to support anisotropic sampling [Mittring 2008]. However, our atlas already contains pre-shaded, pre-filtered content. Avoiding anisotropic sampling during the final display stage benefits the client, which has only modest GPU performance.

We manage the atlas memory over multiple frames, with the goal of optimizing temporal coherence. Already allocated patches, which remain visible in subsequent frames, should remain unchanged, while new allocations should strive to fill gaps between previous blocks, so that fragmentation is minimized. This ensures a small memory footprint, but, more importantly, temporal coherence: Blocks remain in the same place over multiple frames. The shading stored inside a block is updated in every frame. However, the shading typically changes gradually, so the MPEG encoder needs to transmit only local image differences. The blocks are organized in a three-layer hierarchy (Figure 7):

(1) The atlas is regularly subdivided into square *superblocks* of the same size.
(2) Each superblock is subdivided into *columns* of equal width.
(3) Each column is divided into *blocks* of the same width, but variable height.

The three-layer hierarchy allows a tight packing of blocks of variable sizes without having to resort to a global optimization scheme. The effort for insertions and deletions is linear in the number of block changes, without being affected by block sizes. This is made possible by managing one block dimension (width) *across* superblocks, while the other block dimension (height) is managed *inside* each superblock. See section 4.2 for a detailed example.

4.2 Parallel memory allocation

On a massively multi-threaded GPU, dynamic memory allocation is known to be a hard problem. However, the required fine-grained allocation is too computationally demanding to run it on the CPU on just 1-2 cores, as it would be too slow (see section 5.3 for details). On the GPU, conventional solutions for memory management, such as global free-space lists, quickly become infeasible due to excessive locking requirements when many threads compete for access [Steinberger et al. 2012]. Instead, we address this problem by combining two strategies, a GPU-friendly allocation strategy and a global garbage collection step.

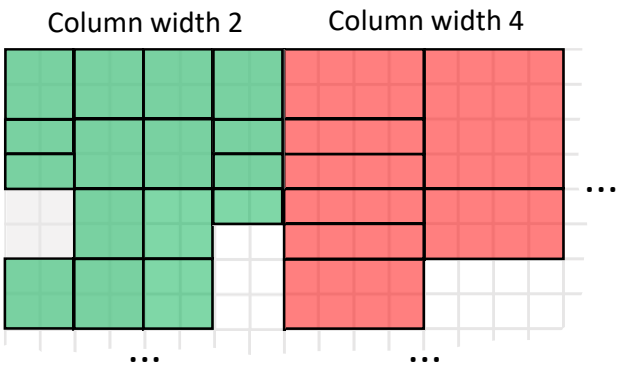


Fig. 7. The atlas is divided into square superblocks (8x8 in the figure). The green superblock is subdivided into columns of width 2, and the red superblock is subdivided into columns of width 4. Columns are packed with blocks of variable height.

We use lock-free allocation, which is not affected by race conditions. Every thread performing an allocation step is guaranteed to finish within bounded time. Each block is serviced by an independent thread. To keep fragmentation minimal without requiring locking, we manage free blocks in *block stacks* and let the allocation run in three distinct phases: a request phase, a provisioning phase and an assignment phase. In each phase, either allocation *or* deallocation occurs, but not both. This approach lets us operate each stack by atomically increasing or decreasing a stack counter without any need for locks.

Request phase. This phase runs in parallel for all patches. Blocks of patches that have become invisible are inserted into a block stack corresponding to their level. Patches that have become visible increase an atomic request counter corresponding to their level. Patches that are reallocated, e.g., because their desired block size changed, execute both operations. Upon conclusion of the phase, we have determined the total number of blocks required to be allocated for each level, and each patch allocating a block stores its *slot*, i.e., the value it has drawn from the request counter.

Provisioning phase. This phase operates in parallel with one thread per column width. It determines how many requests from patches can be served from the block stacks. For the remaining patches, it draws free superblocks from a superblock stack. It starts in decreasing order of block height and sums up the total memory needed, storing offsets and number of required blocks to be allocated from superblocks per block height. We also memorize the fill-rate of the last superblock for each level and top up the superblock in the next frames before requesting a new superblock for that level.

An example of this procedure is shown in Figure 8. In this example, the provisioning phase runs for a column width of 4 pixels with 8×8 superblocks. The job description is given in Table 1, where each column represents a block height. The table lists, per height, the requested blocks, the blocks that are free on the block stack, the required blocks, i.e. requested blocks remaining after using up the block stack, and the offset (in height units) from the first superblock.

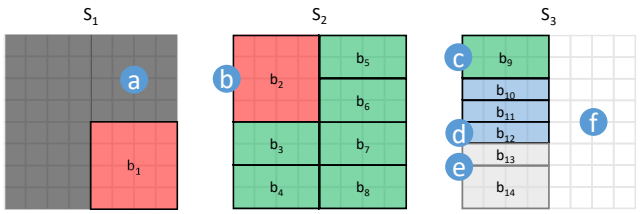


Fig. 8. Example of the provisioning phase of the parallel memory allocation. (a) The current superblock S_1 is partially filled. (b) For the allocation of the 4×4 blocks, a second superblock S_2 has to be allocated. (c) For the 4×2 blocks, another superblock S_3 has to be allocated. (d) The 4×1 blocks fit into the remaining space of S_3. (e) Remaining rectangular blocks are put on block stacks. (f) The remaining space in S_3 will be used in the provisioning phase of the next frame.

Table 1. Job description for the provisioning example. The data corresponds to the example shown in Figure 8. It shows the state before (top rows), values computed during (center rows) and the state after (bottom rows) the provisioning phase.

Height of block	4px	2px	1px
Requested blocks	7	16	10
Free on block stack (before)	5	7	7
Required blocks	2	9	3
Offset (in height units)	3	10	34
Free on block stack (after)	5	8	8
Total available blocks	7	17	11

We start with the largest height 4 and first consider superblock S_1. (a) It already contains three blocks of height 4, which gives us a starting offset of 3. (b) Since we need to allocate space for two blocks b_1, b_2, but only b_1 fits into S_1, we allocate a second superblock S_2 with space for b_2. We proceed to a height of 2 and compute the offset from the starting point, S_1, as $(3 + 2) \cdot 2 = 10$ blocks of height 2. (c) We need space for seven blocks $b_3 \cdots b_9$, but S_2 has only enough space for six, so a third superblock S_3 is allocated, with space for b_9. (d) For height 1, we start with an offset of $(10 + 7) \cdot 2 = 34$ blocks and determine that we can fit all remaining blocks $b_{10} \cdots b_{12}$ into S_3. (e) Since S_3 is not completely filled, we put the remaining rectangular blocks b_{13} on the block stack for level $(4, 1)$ and b_{14} on the block stack for level $(4, 2)$. This ensures that we can start with a block of height 4 again in the next frame. (f) We memorize that two blocks of height 4 are left in S_3 for the next frame.

Assignment phase. This phase runs in parallel for all allocating patches. Based on its slot and the required blocks from the last phase, each patch determines whether its request is served by the allocated superblocks or the block stack. If the slot is within the number of required blocks, we directly compute the position within the provisioned superblock with the slot number and stored offset. Otherwise, the patch directly draws a block from the block stack.

This strategy is not only lock-free, it also deals with provisioning by superblock, avoiding tedious bookkeeping during bulk allocations. Recycling happens on the level of individual blocks, but uses efficient atomically operated stacks to manage the recycled blocks.

Remaining fragmentation is taken care of by global garbage collection, which is aligned with the creation of a new MPEG I-frame. Unlike a P-frame, an I-frame does not rely on temporal coherence, so the garbage collection is free to re-arrange all blocks arbitrarily in order to minimize fragmentation, without hurting MPEG bitrate. We call this process an *atlas reset*. The atlas reset is invoked adaptively: If the memory load exceeds a high water mark, an atlas reset is performed in sync with an I-frame. Clearing of unused blocks in the atlas happens for every I-frame to avoid compression of outdated shading information.

In section 5.4, we compare our parallel memory management strategy to simpler alternatives. We show that only our method can keep memory fragmentation low enough to handle large scenes.

4.3 Level selection

Level selection determines the block size to which a visible patch is mapped. We encode the rectangular block size as 2D vector of binary logarithmic values, the level $\vec{l}$. The goal of the level selection is to assign each patch an appropriate share of the texture atlas. Under memory pressure or if other objectives, such as foveated rendering, should be addressed, a *bias* can be applied to this level.

Shading information is stored in atlas space, but applied in screen space. Therefore, the aim of the level selection should be to preserve the shading rates of the triangle in screen space. Choosing the level based on screen-space edge lengths and area fulfills this requirement. We compute the edge length s of an edge as

$$s = \frac{r}{f} \cos^{-1}\left(\frac{\vec{v}_1 \cdot \vec{v}_2}{v_1 v_2}\right), \tag{1}$$

where r is the resolution in pixels, f is the field of view, and $\vec{v}_1$ and $\vec{v}_2$ are the vectors from the camera to the vertices of the edge. This approximation has the advantage of being independent of the view direction, as it assumes a curved image plane.

Given the edge lengths, we can compute area (Heron's formula) and aspect ratio of the block. The aspect ratio is computed from the maximum edge lengths of those edges that are axis-aligned in the patch layout. To choose one of the discrete block sizes, the binary logarithms of the area, A, and of the aspect ratio, a, are computed, rounded and clamped; a is rounded so that it has the same parity as A, and we arrive at integer values. With A and a, the binary logarithmic width w and height h are computed as

$$\vec{l} = \begin{bmatrix} w \\ h \end{bmatrix} = \begin{bmatrix} \frac{A+a}{2} \\ \frac{A-a}{2} \end{bmatrix}. \tag{2}$$

The chosen levels must respect the anticipated memory pressure, to avoid running out of memory if too many patches must be allocated. A bias b is derived from the ideal area I (the sum of all triangle areas, clamped to the smallest and largest block area), and the total atlas space S. We correct the computation by removing the number of 1×1 blocks d that cannot be further scaled down.

$$b = \log_2\left(\frac{I-d}{S-d}\right) \tag{3}$$

Fig. 9. Untethered smartphone-based headset prototype based on the Qualcomm Snapdragon™ 835 Mobile Platform, with a resolution of 2560×1440, displaying a stereo view of the Robot Lab scene.

On this log-scale, a bias of 0 means that the ideal level is accepted, a bias of 1 means the ideal area is halved and so on. The bias is subtracted from the logarithmic area before rounding and clamping.

5 RESULTS

Our software prototype has been implemented using the Vulkan graphics API and runs under Windows, Linux and Android. Tests were conducted on a desktop PC (NVidia GeForce GTX 1080 Ti GPU, Intel i7-5820K CPU 3.3 GHz, 24 GB RAM) and a headset prototype based on the Qualcomm Snapdragon™ 835 Mobile Platform (Figure 9). The PC was used for frame-by-frame tests (sections 5.1-5.8) and as a server for end-to-end timings with the headset as the client (section 5.9).

In our tests, we used a rendering target for the client with a resolution of 1920×1080 at a horizontal FOV of 90°. All our tests run with realistic game scenes. We use the Unity3D sample scenes "Viking Village" (outdoor scene, 4.7M triangles, 1215 objects, 31 textures) and "Robot Lab" (indoor scene, 0.5M triangles, 645 objects, 49 textures), and the Crytek scene "Sponza" (indoor scene, 0.3M triangles, 392 objects, 53 textures). For each scene, we recorded a camera path with a total length of 10 seconds (1200 frames).

To compute the PVS, we enlarge the FOV by 20%. Moreover, the PVS uses linear extrapolation of the camera path based on the two most recent view matrices. Unless otherwise noted, prediction is done in steps of 33.3 ms. The number of prediction steps varies based on the overall maximum latency, which depends on an assumed static network latency and the server frame rate. For example, at a server frame rate of 30 fps (33.3 ms per frame) with a static latency of 100 ms, the maximum latency is 133.3 ms. In this case, prediction intervals of 33.3 ms, 66.6 ms, 100 ms and 133.3 ms are used.

5.1 PVS correctness

We begin by demonstrating that our sampling strategy to determine a PVS is effective, which is a prerequisite for correct operation of the system. Since we want to avoid disocclusion coming from PVS underestimation, the main interest lies on the false negatives, i.e., the patches that are not classified as part of the PVS, yet become visible. We also record the false positives, i.e., the patches that are reported as part of the PVS, although they never become visible.

We compare the following modes: *Linear prediction* uses a linear extrapolation of motion in five steps, with the EVS computed at 120 Hz (0 ms, 8.3 ms, 16.7 ms, 25 ms, 33 ms) and added to the PVS.

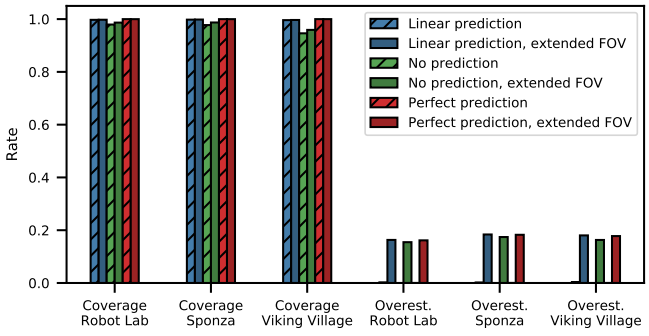


Fig. 10. For PVS computation, we report coverage rate (how many visible patches were not overlooked) and overestimation rate (how many patches were needlessly classified as visible).

Reference prediction works like linear prediction, but using the actual (recorded) camera path instead of a predicted one. *No prediction* uses the EVS sampled at 0 ms as the PVS, establishing a baseline.

Let v denote the total number of patches visible over the five EVS sample positions, computed by reference prediction and weighted by the projected area of the patches' triangles in screen space, as a measure of visual importance. Similarly, let f_n denote the number of area-weighted false negative patches, and, f_p denote the number of area-weighted false positive patches. We measure the coverage rate $r_c = 1 - f_n/v$ and the overestimation rate $r_o = f_p/v$, both for the regular FOV and the extended FOV.

Figure 10 shows averages for our test scenes. We see that even simple linear prediction achieves an average coverage rate >99%. In contrast, no prediction has an average coverage rate of 96%. We expect that the result of no prediction would be even lower for a faster moving camera. We conclude that our PVS algorithm is suitable for framerate upsampling in the considered interval.

The main reason why linear prediction misses a small percentage of the triangles is geometric aliasing from rasterizing tiny, sub-pixel sized triangles in the test scenes. Such tiny triangles can occasionally produce isolated flickering. They can be avoided with better level of detail management (see section 6.4). A similar problem with numerical limitations of rasterization precision causes reference prediction to miss the 100% mark for the extended FOV.

As expected, the overestimation of the regular FOV is negligible, but the overestimation of the extended FOV compared is around 10%. It could be reduced by using a more modest setting for the extended FOV, provided a better prediction model is used.

5.2 Visibility algorithm vs memory requirements

We were interested in how the visibility computation strategy influences the memory requirements of the shading atlas. In conventional game engines, per-object visibility is often determined on the CPU during view-frustum culling and avoids a potentially expensive GPU visibility pass. However, without occlusion culling, the number of pixels that need to be represented in the atlas is dependent on the depth complexity, and thus not necessarily proportional to the screen resolution. Since streaming performance is sensitive to pixel count, we must strive for a small atlas size.

Therefore, we compare memory requirements (without bias) of per-patch visibility and per-object view-frustum culling. In all our test sequences, culling required a large multiple of allocated pixels compared to per-patch visibility: Robot Lab 16.42×, Sponza 11.25×, Viking Village 95.96×. Since even a factor of 2× would already imply using twice the network bandwidth, we conclude that coarse-grained visibility based on culling is not sufficient for SAS.

Table 2. Comparison of memory management speed CPU/GPU.

Scene	CPU	GPU	Speedup
Robot Lab	44.66	0.19	235×
Sponza	37.21	0.14	266×
Viking Village	377.20	0.24	1715×

5.3 Memory management speed

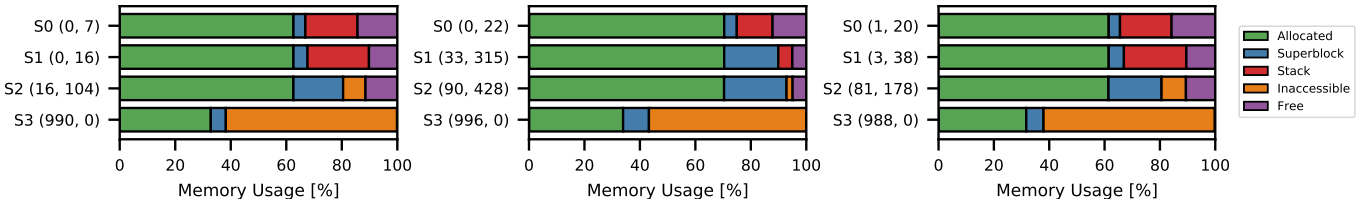
Since fine-grained multi-layer memory allocation has been shown to be fast enough on the CPU for terrain VT [Mittring 2008], we investigated if our memory allocation could be run on the CPU as well, avoiding a more complex GPU-side memory management. For comparison, we implemented a single-threaded CPU memory allocation strategy, which uses a quadtree-like layout to allocate blocks and achieves fragmentation that is always better than or equal to our GPU allocation strategy. We compare the performance of CPU and GPU allocation. We report only the runtime of the memory allocation itself, assuming that all other stages of the pipeline are identical. In favor of the CPU variant, we do not consider the additional transfer times between GPU and CPU. Table 2 shows the measured times.

We see that, on average, the CPU variant takes at least 30 ms even for the smallest scene, while the time required by the GPU version is <0.25 ms. At a targeted server framerate of 30 Hz (33 ms per frame), adding an additional 30 ms for memory management would lower the framerate to <15 Hz. We therefore rule out the possibility of memory allocation on the CPU.

5.4 Fragmentation vs allocation strategy

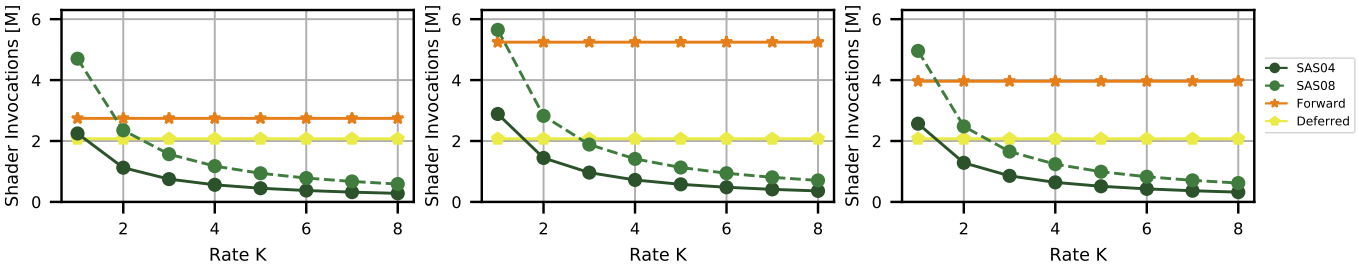
When the atlas size is limited, fragmentation results in excessive memory pressure and reduced image quality. We demonstrate the difference between our recommended strategy, denoted by $S0$, to simpler strategies $S1$ (no columns), $S2$ (no columns, no block stack), and $S3$ (no columns, no block stack, no atlas reset). We compare how memory is used by each method, for an atlas size of 4 MPix. New memory can be either allocated in free superblocks, at the end of partially filled superblocks, or by claiming free areas in the middle of superblocks. We consider the latter two cases as fragmented memory. Free memory in the middle of superblocks is either available on the block stacks ($S0, S1$) or temporarily inaccessible until the whole superblock is freed ($S2, S3$).

As can be seen in Figure 11, the recommended strategy $S0$ shows the least memory fragmentation and requires the least number of atlas resets. It only runs out of memory once across all tests, when



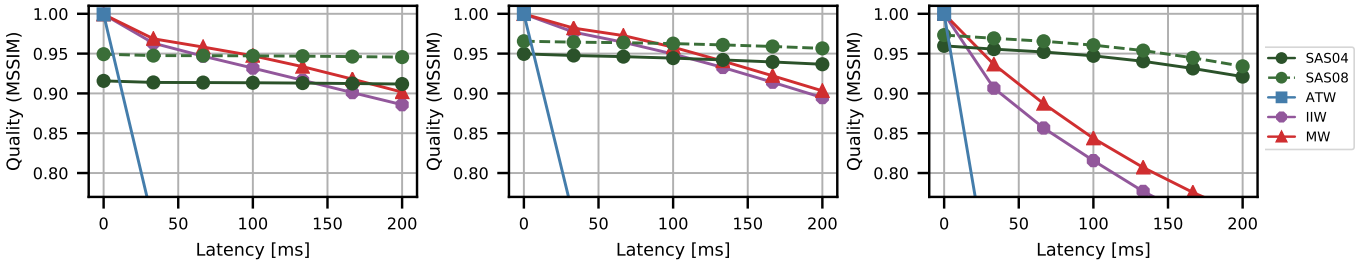
(a) Robot Lab (b) Sponza (c) Viking Village

Fig. 11. Comparison of the influence of the memory allocation strategy on atlas memory averaged over 1000 frames. Compared are $S0$ (reference strategy), $S1$ (no columns), $S2$ (no columns, no block stack), and $S3$ (no columns, no block stack, no atlas reset). Stated in parenthesis are the number of frames where out of memory occurs and the number of atlas resets. The 4 MPix atlas memory is either allocated (green), at the end of a superblock that is partially filled (blue), within blocks on the block stacks (red), temporarily inaccessible (orange) or within free superblocks (purple). Only $S0$ can reliably maintain a safety margin for all atlas sizes.



(a) Robot Lab (b) Sponza (c) Viking Village

Fig. 12. Comparison of fragment shader invocations (in millions) for forward rendering, deferred rendering and standalone object-space shading with our Shading Atlas. The x-axis displays the upsampling factor K, describing how many frames are generated from a single shading atlas.



(a) Robot Lab (b) Sponza (c) Viking Village

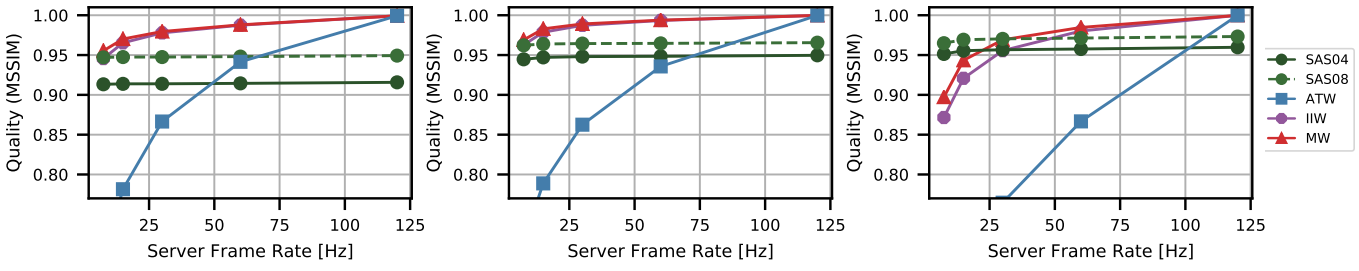
Fig. 13. Image quality (reported as MSSIM) for varying network latency, compared for SAS (using multiple atlas sizes). As latency increases, SAS maintains image quality better than asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).

we set an aggressive high water mark of 96% for Viking Village. The simpler strategies $S1$ and $S2$ require more atlas resets and run out of memory frequently. Strategy $S3$, which can neither suppress fragmentation nor recover on atlas resets, consistently runs out of memory within the first few frames of the test sequence. We conclude that our preferred memory management strategy $S0$ is mandatory for sustainable operation of the shading atlas.

5.5 Rendering amortization from framerate upsampling

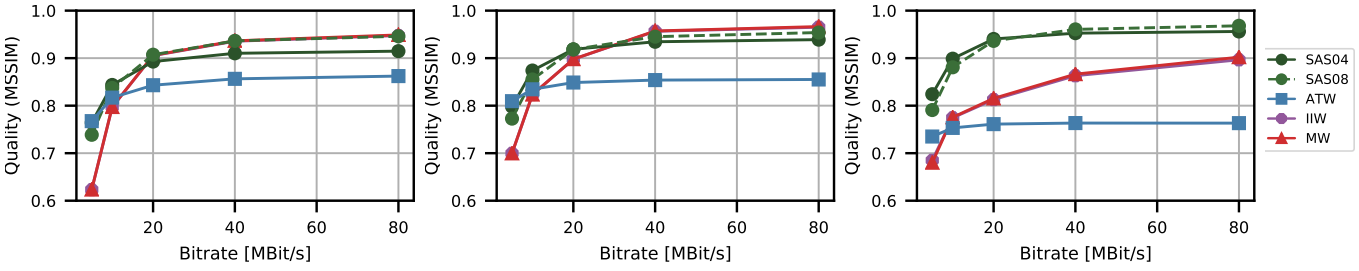
Before we investigate how our object-space rendering performs in a client-server system, we evaluate how well rendered pixels can be amortized during framerate upsampling in a standalone configuration (assuming a tethered headset is connected to a desktop computer). Compared to conventional forward rendering, object-space shading introduces some amount of overhead, which is largely related to the atlas size.

In Figure 12, we compare fragment shader invocations of forward rendering and deferred rendering at 1920×1080 to our object-space rendering with atlas sizes of 4 and 8 MPix. The x-axis varies the upsampling factor K (e.g., $K = 4$ for upsampling from 30 to 120Hz). Object-space rendering becomes more efficient in all cases for $K \geq 3$ and even for $K \geq 2$ in comparison to forward rendering. Note that



(a) Robot Lab (b) Sponza (c) Viking Village

Fig. 14. Image quality (reported as MSSIM) for varying upsampling rate, compared for SAS (using multiple atlas sizes), asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).



(a) Robot Lab (b) Sponza (c) Viking Village

Fig. 15. Rate distortion curves after MPEG compression. We report image quality (MSSIM) as a function of network bitrate for SAS (using multiple atlas sizes), asynchronous time warping (ATW), mesh warping (MW) and iterative image warping (IIW).

4 MPix is already more than adequate in terms of shading quality for 4× upsampling. We conclude that our approach is suitable for operation as a standalone object-space shading system.

5.6 Image quality vs network latency

Conventional streaming combines forward rendering with framerate upsampling via IBR. The key measure for a good streaming solution is the relationship of image quality to pixel rate (i.e., pixels per second that must be transmitted).

We compare SAS against three IBR methods. Asynchronous Time Warping (ATW) is the most simple IBR method which just applies a homography transformation to the source image. Mesh Warping (MW) [Mark et al. 1997] uses a dense vertex grid with one quad per pixel and transforms the grid based on the depth buffer. Iterative Image Warping (IIW) [Yang et al. 2011] reverses the warping flow and searches for an input pixel starting from the output pixel.

The experiment shown in Figure 13 varies the (simulated) network latency, while keeping the client and server frame rate fixed at 120 Hz. Without latency, IBR methods have the advantage of using exactly the same input view as the ground truth, while SAS depends on the atlas size.

To measure image quality, we use the structured similarity image measure (SSIM) [Wang et al. 2004]. We compute the SSIM using a Gaussian filter with a window size of 11 pixels and a standard deviation of $\sigma = 2$, and constants $C_1 = 0.0001$ and $C_2 = 0.0009$. The ground truth for the comparison uses the scene rendered with a forward rendering pass. We compute the mean SSIM (MSSIM) for the whole sequence of 1200 frames.

As can be expected, quality decreases at different rates as network latency increases: While the quality of IBR methods starts to decrease immediately, SAS manages to keep the quality at a high level due to its geometric model. For large scenes, the crossover point is already at 20 ms. This is noteworthy, since end-to-end latency is typically much larger (see also section 5.9).

5.7 Image quality vs upsampling rate

In this experiment, we vary framerate upsampling rather than network latency. The latter is set to zero. The client frame rate is fixed at 120 Hz, while the server frame rate varies (7.5, 15, 30, 60, and 120 Hz). Increasing the framerate upsampling can be seen as introducing variable latency (duration (8) in Figure 2) from zero to a maximum just before a new server frame arrives. The average latency resulting from a server framerate of K Hz is $\frac{1}{2K}$ s. The results in Figure 14 are consistent with this average latency estimate. We conclude that average latency resulting from framerate upsampling can be added to network latency in our system model.

ATW cannot compete with the other methods quality-wise. SAS beats MW and IIW at upsampling rates of 4× (30Hz) or higher for Viking Village, but not for the smaller scenes, Robot Lab and Sponza. However, in practice, the effects of latency from framerate upsampling must be added to network latency, which is at least 60ms (see section 5.9), giving SAS an advantage for all scene sizes.

5.8 Rate distortion

All experiments presented so far did not consider the impact of lossy MPEG compression on image quality. In an end-to-end system, the main concern is network utilization. Consequently, we report a rate distortion curve, relating image quality (measured as SSIM) to bitrate (Mbps after compression).

We compare SAS with atlas sizes of 4 and 8 MPix to ATW, MW and IIW. The experiment uses a server frame rate of 30 fps, while the client target frame rate is 120 fps. Network latency was assumed to be zero. First, the server generates uncompressed texture atlases. Then, we apply H.264 video coding, with CAVLC entropy coding, a single reference frame, and no B-frames. We vary the bit-rate in 2× steps (5, 10, 20, 40, and 80 Mbps). Atlas reset and I-frame interval is set to five frames. The decoded atlases are passed to the client, and are used to render the output frames at the target framerate.

Figure 15 presents the results for our test sequences. Even at a bitrate of 10 Mbps, ATW is already beaten by the other methods. For the smaller scenes, MW and IIS benefit more from higher bitrates, outperforming SAS at 20-40 Mbps. However, this measurement does not account for the need of MW and IIS to encode stereo pairs, so the actual crossover point would be at half this bitrate. Moreover, all methods except ATW have SSIM above 90%, which is already a very high image quality with little room for improvements. For the large scene, Viking Village, SAS always delivers the best image quality, by a large margin. In general, we observe diminishing returns for bitrates above 40 Mbps in the tested configuration, approaching the quality of rendering from an uncompressed atlas.

5.9 End to end performance

Finally, we report on system performance on a mobile client device based on the headset running Android. The headset has a single display (shared by both eyes) with 2560 × 1440 resolution at 60 Hz, driven by an Adreno 540 GPU. Typically, a mobile GPU has at least an order of magnitude less compute power than a high-end desktop GPU, like the GeForce used in our tests. Moreover, the larger scenes, such as Viking Village and Robot Lab, do not even fit in the RAM of the mobile device. Therefore, we cannot report on any native VR rendering results on the headset for comparison to SAS.

We connected the headset to the desktop computer by 802.11ac WiFi at a throughput of about 526 Mbps. The client code is not yet optimized, so overall results have to be interpreted with caution. Figure 16 reports performance figures for the various system stages (Figure 2). The server framerate was locked to 30 Hz and the client framerate was unlocked. We tested the system with an atlas size of 4 MPix and high shading load (400 light sources).

The sum of phases 1-7 (until the first upsampled image appears) leads to a median total round-trip-latency of around 86 ms. Pose transmission time from client to server is about 2 ms; server idle time around 4 ms; server processing takes around 29 ms. MPEG transmission time is a significant factor at ~ 30 ms, but it can be amortized by creating stereo pairs from a single atlas, and by framerate upsampling. This early-generation headset is limited to 60 Hz, but future generations will be able to display 120 Hz, which works in favor of framerate upsampling, as more frames can be amortized within the same prediction period.

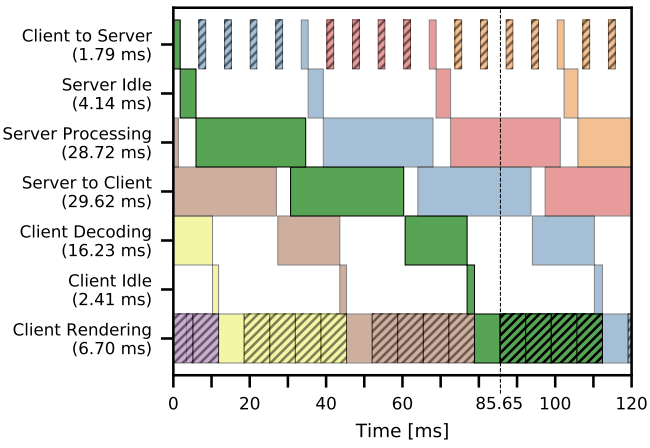


Fig. 16. End to end latency for the Robot Lab scene, measured with the headset as the client device. We report median times for all stages in addition to the median end to end latency (dashed line). The 'Server to Client' and 'Client Decoding' stages show the timing of the longest sub-path.

Meta-information transmission takes a non-negligible amount of time of around 19 ms. We believe this is because of our inefficient implementation. First, we apply only rudimentary compression to this data (Section 6.5). Second, TCP should be replaced with a more throughput-oriented protocol. Occasional packet losses can be rectified during atlas reset.

The client decoding stage is taking around 16 ms total, whereby the atlas and the meta-information are decoded in parallel. The MPEG frame is sent to a dedicated hardware decoder unit, at a rate of 1.7-2.2 Mpix/s (slightly higher for more complex shading). The meta-information is decoded on the GPU. As expected, MPEG decoding is the limiting factor, requiring roughly twice the time of meta-information decoding. While MPEG decoding will become faster with future generations of hardware, the meta-information decoding is currently most affected by scattered memory accesses. This problem can be addressed by a more batch-oriented encoding of meta-information.

The final part of the pipeline consists of waiting for the GPU (2 ms) and rendering the final image (7 ms). The last phase (illustrated using hatched rectangles in Figure 16) renders multiple frames using the same atlas, until new data from the server arrives.

While the throughput of our early prototype obviously needs improvements, we can already make two important observations: First, the fraction of the round trip latency that does not depend on the rendering method (network transmission, encoding, decoding, idle times) already exceeds 50 ms. We have demonstrated in section 5.6 that SAS is a competitive solution at this latency. Second, the display stage at the client is very lightweight. At the observed 6.7 ± 2.6 ms for drawing an average of 32K triangles per frame, we can reliably achieve a frame rate of 120 Hz. Outliers of frame times rarely happen (< 0.6 % of all frames). We conclude that SAS is a valid approach for future generation VR headsets.

6 LIMITATIONS AND EXTENSIONS

In this section, we discuss limitations of our system, and we report some preliminary results on extensions that address them.

6.1 View-dependent rendering

Our current implementation does not consider fast changing view-dependent effects, like crisp highlights. If shading information is re-used by framerate upsampling, fast camera motion can reveal incorrectly extrapolated shading. One obvious method to suppress such problems is by introducing variable-rate shading. Since the shading information belonging to a particular object is explicitly represented in the atlas, each object can be updated at any time. More important objects, or objects with highly view-dependent materials, could be updated more often than others. If an atlas is partially updated in this way, the bitrate required by an MPEG P-frame produced from such a partially updated atlas will be proportional to the amount of changes, and not the overall atlas size. Consequently, bitrate can be allocated based on object priorities without changing the streaming format.

6.2 Transparent geometry

Transparent geometry in the scene can be handled in SAS by adding a traditional (order-dependent) pass for transparent geometry to the client. The transparent geometry must be marked by the server, and rendered by the client after completing the rendering of the opaque geometry, in back to front order, using alpha blending. This requires a second draw call to be executed by the client, but otherwise, the client's rendering pipeline is not significantly complicated. This makes the approach compatible with mobile GPU capabilities, since no costly fragment shaders need to be run.

Only the meta-information must be augmented to distinguish transparent geometry; the shading information of transparent and opaque geometry can be stored together in the atlas in a uniform way. Alpha values that are uniform per triangle can be stored in the triangle meta-information; per-pixel alpha values would have to be stored by extending the atlas from RGB to RGBA channels. If this is undesirable (for example, because hardware MPEG encoders do not support RGBA formats), a 1-bit alpha value can be expressed in a regular RGB atlas by chroma keying.

6.3 Postprocessing effects

Many game engines use postprocessing effects after the main rendering pass, which operate on color, depth and other rendering results stored in a G-buffer. Technically, a postprocessing stage can be added to the shading stage on the server or to the display stage of the client. However, adding general-purpose postprocessing to the client would imply shipping a G-buffer to the client (which would be too costly in terms of network bandwidth) and running proprietary shaders on the client (which would be too costly on a mobile GPU and violate the "thin client" idea).

Therefore, we believe it is more beneficial to add postprocessing to the shading stage of the server. One caveat is that view-dependent postprocessing effects are subject to the same restrictions as view-dependent forward rendering effects. They could be handled with variable shading rates, as discussed in section 6.1. For shipping postprocessing effects to the client, they can be classified according to if/how they interpret the scene geometry:

Fig. 17. To demonstrate how postprocessing effects can be integrated into SAS, we added screen-space ambient occlusion to the shading stage. The screenshots show Phong shaded images without (left) and with (right) SSAO.

Effects that modify surfaces (e.g., depth of field effects or ambient occlusion) can be baked into the shading atlas.

Effects that are purely screen-space aligned can be rendered into a transparent billboard, aligned with the near clipping plane. The billboard is handled in the same way as the transparent geometry described in section 6.2. This would, for example, work for lens flare, etc. For best results, the 2D area affected by the screen-space effects should be subdivided into a quad or triangle mesh, such that moderate block sizes can be assigned to the individual pieces of the screen-space effect.

Effects located in free-space (e.g., particle effects) can be rendered onto strategically placed billboards as well. The billboard is best placed at the average depth of the corresponding 3D object. If the client renders the billboard with depth test enabled, occlusion between the effect pixels and the scene can be approximately resolved, even when the camera is moved during framerate upsampling.

As a proof of concept, screen-space ambient occlusion (SSAO) rendering [Mittring 2007] was added to SAS. Our SSAO implementation uses multiple depth layers [Bavoil and Sainz 2009] via depth peeling on the server to acquire correct results also for occluded triangles. The result is baked into the shading atlas for every polygon in the PVS, including those that only become visible in the future. Figure 17 shows the result rendered at the client from the shading atlas with SSAO enabled and disabled. For better demonstration of the effect, only simple Phong shading with color textures was used.

In a similar way, high dynamic range (HDR) can be added to the server rendering. A post-perspective HDR G-buffer can be replaced by an object-space HDR atlas. Tone mapping on the server converts the HDR atlas to a standard RGB atlas, which is shipped to the client. Like with all other shading effects, the client does not have to be concerned with how the tone-mapped result was created.

6.4 Geometric scene complexity

A fundamental assumption of SAS is that the client is able to handle the geometric complexity of the visible part of the scene. The size of the visible scene will generally be much smaller than the size of the total scene, but the visible part can still exceed the capabilities

of a mobile GPU. We also observe that our multi-sampled PVS puts a high geometry processing load on the server.

However, if either client or server suffer from too large scenes, the geometric complexity can be reduced with standard methods: In the simplest case, a conventional geometric level of detail (LOD) system uses a fixed set of simplifications and chooses to display one of them based on the distance from the viewer. Removed details after simplification can optionally be baked into normal maps.

Geometric LOD is also beneficial to suppress subpixel-sized triangles, which produce geometric aliasing and can be troublesome for PVS estimation. On the one hand, the LOD system chooses a level where triangles in screen-space are large enough to avoid geometric aliasing and reduce the geometric load on the system, especially on the client. On the other hand, the screen-space triangle size should not exceed the size of a superblock, so that the triangle can be shaded at a high enough resolution. Therefore, we favor a combination of simplification and tessellation of overly large triangles to effectively place a band-limit on triangle sizes, such that they fit into the available block sizes.

6.5 Dynamic geometry

Our current system supports dynamic geometry (rigid and non-rigid animations), as can be seen in the "water scene" of the supplementary video. Non-rigid animation or tessellation shaders do not require special handling; changed vertices are transmitted in the same way as newly visible vertices. However, dynamic geometry requires more network bandwidth.

Static scenes typically create a bitrate peak only in the first frame, when many vertices and triangles need to be transmitted at once. This peak can be hidden by delaying the display start of the client until the second frame. In contrast, scenes with dynamic geometry may have a continuous, non-negligible bandwidth requirement for vertex updates. To keep the client simple, we prefer not having to run any shaders on it that are proprietary to the game, such as vertex animation shaders.

Instead, we have added geometry compression as an experimental extension of our system. First, vertices are uniformly quantized; above a certain number (> 30) of visible vertices, we compress the vertices using an octree encoding [Botsch et al. 2002] to about 15 bits per 3D point. We also apply simple lossless compression (difference encoding per attribute followed by exponential Golomb coding) to the triangle and texture coordinate messages, as these messages tend to follow the object scene structure and thus exhibit a lot of coherence. These measures typically reduce the overall bandwidth requirements for dynamic geometry by >3× over a naive uncompressed representation (Table 3). Moreover, compression rates tend to get better with larger amounts of dynamic geometry, since more entropy can be detected.

Table 3. Detailed compression rates achieved on average for selected message types. The overall compression rate is about 3×.

Message Type	Robot Lab	Viking Village
Vertex coordinates	6.68×	5.95×
Vertex id	1.78×	1.70×
Triangles	2.22×	2.20×
Texture coordinates	3.19×	3.09×

6.6 Game engine integration

In general, integrating SAS into a game engine is similar in effort and difficulty to integrating a deferred rendering pipeline or another type of advanced graphics pipeline. Because of the high demands of VR systems, contemporary game engines, such as Unreal Engine [2], give developers a choice of forward rendering with less post-processing to reduce the latency compared to their standard pipeline. Integrating the SAS pipeline into a game engine with reduced post-processing is considerably less challenging.

7 CONCLUSIONS

To our knowledge, we have presented the first work that demonstrates an object-space shading approach for remote rendering with low perceived latency. While high throughput and low latency are essential properties of streaming VR applications, other graphics streaming applications, such as networked games, architectural preview or scientific visualization, can benefit as well.

Compared to other object space shading methods, SAS requires neither custom GPU extensions nor generating unique texture coordinates. Moreover, SAS uses only a moderate amount of memory.

If more than one or two views must be generated, such as on autostereoscopic TV screens, or for future VR headsets with extreme resolutions, such as 8K displays, enormous network bandwidth is required when using conventional video streaming. In contrast, our approach effectively decouples display from image generation for short periods of time, a property that we believe will become increasingly important.

This decoupling makes SAS appealing for non-networked applications as well, where it has the potential to replace existing "split rendering" pipelines that perform framerate upsampling in order to achieve sustained framerates.

We have already discussed the many avenues of future work in the previous section. We are sure that we have only scratched the surface of the possibilities offered by replacing conventional post-perspective video streaming with an alternative in object-space. Our client is designed to be simple and understand only a "fixed function" set of rendering features that we considered essential. An object-space representation lends itself to extensions that will make the client more capable, taking on a larger role in streaming rendering. Finally, user studies will play an important role in future work to better understand perceptual characteristics of split-rendering systems.

ACKNOWLEDGMENTS

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

[2] https://docs.unrealengine.com/en-us/Engine/Performance/ForwardRenderer

REFERENCES

John M. Airey, John H. Rohlf, and Frederick P. Brooks, Jr. 1990. Towards Image Realism with Interactive Update Rates in Complex Virtual Building Environments. In *Proceedings ACM I3D*. 41–50.

Tomas Akenine-Möller and Timo Aila. 2005. Conservative and Tiled Rasterization Using a Modified Triangle Set-Up. *Journal of Graphics Tools* 10, 3 (Jan 2005), 1–8.

Magnus Andersson, Jon Hasselgren, Robert Toth, and Tomas Akenine-Möiler. 2014. Adaptive texture space shading for stochastic rendering. *Computer Graphics Forum* 33, 2 (may 2014), 341–350. https://doi.org/10.1111/cgf.12303

Dan Baker. 2016. Object Space Lighting. Talk at Game Developers Conference. (2016). http://www.cogsci.rpi.edu/~destem/gamearch/gdc16/Object-Space-Lighting-Rev-21.pptx

Paul Bao and Douglas Gourlay. 2004. Remote walkthrough over mobile networks using 3-D image warping and streaming. *IEE Proceedings - Vision, Image and Signal Processing* 151, 4 (Aug 2004), 329–336. https://doi.org/10.1049/ip-vis:20040749

Louis Bavoil and Miguel Sainz. 2009. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH 2009: Talks*. ACM, 45.

Kevin Boos, David Chu, and Eduardo Cuervo. 2016. FlashBack: Immersive Virtual Reality on Mobile Devices via Rendering Memoization. In *Proc. MobiSys*. 291–304.

Mario Botsch, Andreas Wiratanaya, and Leif Kobbelt. 2002. Efficient High Quality Rendering of Point Sampled Geometry. In *Proceedings of the 13th Eurographics Workshop on Rendering*. 53–64.

Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross. 2012. Iterative Image Warping. *Computer Graphics Forum* 31, 2pt1 (2012), 237–246.

Chris Buehler, Michael Bosse, Leonard McMillan, Steven Gortler, and Michael Cohen. 2001. Unstructured Lumigraph Rendering. In *Proceedings SIGGRAPH*. 425–432.

Christopher A. Burns, Kayvon Fatahalian, and William R. Mark. 2010. A Lazy Object-space Shading Architecture with Decoupled Sampling. In *Proceedings HPG*. 19–28.

Nathan A. Carr and John C. Hart. 2002. Meshed atlases for real-time procedural solid texturing. *ACM Transactions on Graphics* 21, 2 (apr 2002), 106–131.

Cem Cebenoyan. 2014. Real Virtual Texturing Taking Advantage of DirectX11.2 Tiled Resources. Game Developer Conference. (2014).

Matthäus Chadjas, Christian Eisenacher, Marc Stamminger, and Sylvain Lefebvre. 2010. Virtual Texture Mapping 101. (2010).

Chun-Fa Chang and Shyh-Haur Ger. 2002. *Enhancing 3D Graphics on Mobile Devices by Image-Based Rendering*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1105–1111.

Ka Chen. 2015. Adaptive Virtual Texture Rendering in Far Cry 4. Talk at Game Developers Conference. (March 2015). http://twvideo01.ubm-us.net/o1/vault/gdc2015/presentations/Chen_Ka_AdaptiveVirtualTexture.pdf

Kuan-Ta Chen, Yu-Chun Chang, Po-Han Tseng, Chun-Ying Huang, and Chin-Laung Lei. 2011. Measuring the Latency of Cloud Gaming Systems. In *Proceedings of the 19th ACM International Conference on Multimedia (MM '11)*. 1269–1272.

Shenchang Eric Chen and Lance Williams. 1993. View interpolation for image synthesis. In *Proceedings SIGGRAPH*. 279–288.

Sharon. Choy, Bernard. Wong, Gwendal. Simon, and Catherine. Rosenberg. 2012. The brewing storm in cloud gaming: A measurement study on cloud to end-user latency. In *Workshop on Network and Systems Support for Games (NetGames)*. 1–6.

Petrik Clarberg, Robert Toth, Jon Hasselgren, Jim Nilsson, and Tomas Akenine-Möller. 2014. AMFS: Adaptive Multi-Frequency Shading for Future Graphics Processors. *ACM Transactions on Graphics* 33, 4 (jul 2014), 1–12.

Petrik Clarberg, Robert Toth, and Jacob Munkberg. 2013. A sort-based deferred shading architecture for decoupled sampling. *ACM Transactions on Graphics* 32, 4 (jul 2013).

Daniel Cohen-Or, Yair Mann, and Shachar Fleishman. 1999. Deep Compression for Streaming Texture Intensive Animations. In *Proceedings SIGGRAPH*. 261–267.

Alvaro Collet, Ming Chuang, Pat Sweeney, Don Gillett, Dennis Evseev, David Calabrese, Hugues Hoppe, Adam Kirk, and Steve Sullivan. 2015. High-quality Streamable Free-viewpoint Video. *ACM Transactions on Graphics* 34, 4 (July 2015).

Robert L. Cook, Loren Carpenter, and Edwin Catmull. 1987. The Reyes Image Rendering Architecture. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*. ACM, New York, NY, USA, 95–102.

Cyril Crassin, David Luebke, Michael Mara, Morgan McGuire, Brent Oster, Peter Shirley, Peter-Pike Sloan, and Chris Wyman. 2015. CloudLight: A System for Amortizing Indirect Lighting in Real-Time Rendering. *Journal of Computer Graphics Techniques (JCGT)* 4, 4 (15 October 2015), 1–27. http://jcgt.org/published/0004/04/01/

Eduardo Cuervo, Alec Wolmany, Landon P. Coxz, Kiron Lebeck, Ali Razeenz, Stefan Saroiuy, and Madanlal Musuvathi. 2015. Kahawai: High-Quality Mobile Gaming Using GPU Offload. In *Proceedings MobiSys*. 121–135.

Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2010. Adaptive Image-space Stereo View Synthesis. In *15th International Workshop on Vision, Modeling and Visualization Workshop*. Siegen, Germany, 299–306.

Karl. E. Hillesland and J. C. Yang. 2016. Texel Shading. In *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. 73–76.

Martin Kraus and Thomas Ertl. 2002. Adaptive Texture Maps. In *Proceedings ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. 7–15.

Kyungmin Lee, David Chu, Eduardo Cuervo, Johannes Kopf, Yury Degtyarev, Sergey Grizan, Alec Wolman, and Jason Flinn. 2015. Outatime - Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*.

Sylvain Lefebvre, Jerome Darbon, and Fabrice Neyret. 2004. Unified Texture Management for Arbitrary Meshes. INRIA Research report 5210. (2004).

Marc Levoy. 1995. Polygon-assisted JPEG and MPEG Compression of Synthetic Images. In *Proceedings SIGGRAPH*. 21–28.

Gábor Liktor and Carsten Dachsbacher. 2012. Decoupled deferred shading for hardware rasterization. In *Proceedings ACM I3D*. 143–150.

Yair Mann and Daniel Cohen-Or. 1997. Selective Pixel Transmission for Navigating in Remote Virtual Environments. *Computer Graphics Forum* 16 (1997), C201–C206.

Marc Manzano, José A. Hernandez, Manuel Uruena, and Eusebi Calle. 2012. An empirical study of Cloud Gaming. In *2012 11th Annual Workshop on Network and Systems Support for Games (NetGames)*. 1–2.

William R. Mark, Leonard McMillan, and Gary Bishop. 1997. Post-rendering 3D warping. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*.

Colt McAnlis. 2009. Halo Wars: The Terrain of Next-Gen. Talk at Game Developers Conference. (2009).

Martin Mittring. 2007. Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 courses*. ACM, 97–121.

Martin Mittring. 2008. Advanced Virtual Texture Topics. In *Advances in Real-Time Rendering in 3D Graphics and Games Course SIGGRAPH 2008*. 23–51. http://portal.acm.org/citation.cfm?id=1404435.1404438

Yuval Noimark and Daniel Cohen-Or. 2003. Streaming Scenes to MPEG-4 Video-Enabled Devices. *IEEE Computer Graphics and Applications* 23 (01 2003), 58–64.

OculusVR. 2018. Rendering to the Oculus Rift. (2018). https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-render/ Visited on March 30, 2018.

Dawid Pajak, Robert Herzog, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. 2011. Scalable Remote Rendering with Depth and Motion-flow Augmented Streaming. *Computer Graphics Forum* 30, 2 (2011), 415–424.

Jonathan Ragan-Kelley, Jaakko Lehtinen, Jiawen Chen, Michael Doggett, and Frédo Durand. 2011. Decoupled sampling for graphics pipelines. *ACM Transactions on Graphics* 30, 3 (may 2011), 1–17. https://doi.org/10.1145/1966394.1966396

Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. 2016. Proxy-guided Image-based Rendering for Mobile Devices. *Computer Graphics Forum* 35, 7 (oct 2016), 353–362. https://doi.org/10.1111/cgf.13032

Jonathan Shade, Steven Gortler, Li-wei He, and Richard Szeliski. 1998. Layered Depth Images. In *Proceedings SIGGRAPH*. 231–242.

Bin Sheng, Wei-Liang Meng, Han-Qiu Sun, and En-Hua Wu. 2011. MCGIM-Based Model Streaming for Realtime Progressive Rendering. *Journal of Computer Science and Technology* 26, 1 (jan 2011), 166–175. https://doi.org/10.1007/s11390-011-9423-8

Shu Shi and Cheng-Hsin Hsu. 2015. A Survey of Interactive Remote Rendering Systems. *ACM Comput. Surv.* 47, 4, Article 57 (May 2015).

Shu Shi, Klara Nahrstedt, and Roy Campbell. 2012. A Real-time Remote Rendering System for Interactive Mobile Graphics. *ACM Trans. Multimedia Comput. Commun. Appl.* 8, 3s, Article 46 (Oct. 2012), 20 pages. https://doi.org/10.1145/2348816.2348825

Markus Steinberger, Michael Kenzel, Bernhard Kainz, and Dieter Schmalstieg. 2012. ScatterAlloc: Massively parallel dynamic memory allocation for the GPU. In *2012 Innovative Parallel Computing (InPar)*. 1–10.

Christopher C. Tanner, Christopher J. Migdal, and Michael T. Jones. 1998. The Clipmap: A Virtual Mipmap. In *Proceedings SIGGRAPH*. 151–158.

Eyal Teler and Dani Lischinski. 2001. Streaming of Complex 3D Scenes for Remote Walkthroughs. *Computer Graphics Forum* 20, 3 (2001), 17–25.

J. M. P. van Waveren. 2009. id Tech 5 Challenges - From Texture Virtualization to Massive Parallelization. In *SIGGRAPH 2009 Course: Beyond Programmable Shading*.

Zhou Wang, Alan Conrad Bovik, Hamid Rahim Sheikh, and Eero P Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing* 13 (apr 2004), 600–612. Issue 4.

Lei Yang, Yu-Chiu Tse, Pedro V. Sander, Jason Lawrence, Diego Nehab, Hugues Hoppe, and Clara L. Wilkins. 2011. Image-based bidirectional scene reprojection. In *Proceedings of the 2011 SIGGRAPH Asia Conference*. Article 150.

Ilmi Yoon and Ulrich Neumann. 2000. Web-Based Remote Rendering with IBRAC (Image-Based Rendering Acceleration and Compression). *Computer Graphics Forum* 19, 3 (2000), 321–330. https://doi.org/10.1111/1467-8659.00424

Cem Yuksel. 2017. Mesh Color Textures. In *Proceedings of High Performance Graphics (HPG '17)*. Article 17, 11 pages.