An Image-Space Split-Rendering Approach to Accelerate Low-Powered Virtual Reality

Ville Cantory*  
University of Minnesota

Nathan Ringo†  
University of Minnesota

ABSTRACT
Current mobile GPUs lack the computing power needed to support high end VR rendering. As an alternative, frames can be rendered on a server and streamed to a thin client, but this approach can incur high end-to-end latency due to processing and network requirements. We propose a networked split-rendering approach using an image-space division of labour between a server and a heavy client to achieve faster end-to-end image presentation rates on the mobile device while preserving image quality.

Index Terms: Computing methodologies—Computer graphics—Graphics systems and interfaces—Virtual reality;

1 INTRODUCTION
High performance virtual reality (VR) experiences require a wired connection from the head mounted display (HMD) system to a high powered desktop. In contrast, standalone systems containing a mobile-class System on a Chip (SoC) can provide wireless VR experiences, but they do not have the compute capabilities to render the highest quality graphics for VR applications.

Some methods split the rendering pipeline across the server and a thin client [2, 4, 6], with the thin client doing much less of the processing than the server. A heavy client, however, will incur lower communication costs with the server, decreasing encoding and network overhead. Screen partitioning methods for parallel rendering have been explored in multi-GPU multi-display workstations to balance the number of vertices rendered by each GPU [1], but these partitions rely on object-space information.

* e-mail: canto063@umn.edu
† e-mail: ringo025@umn.edu

We propose a split-rendering solution that uses an image-space partitioning of the scene coupled with a client-heavy approach to avoid high overhead costs. Our system uses a locally networked server to share the workload with a mobile client in rendering the full scene. The client and server each execute the full rendering pipeline to draw separate portions of the framebuffer. The server then uses lossless compression to encode its rendered subregion and sends it to the client, which decodes the subframe and merges the two together. In this paper, we introduce the proposed split rendering method, showcased in a functioning system, as well as an evaluation on end-to-end latency.

2 SYSTEM
Our system uses a heavy client to mitigate high overheads from server-side processing and network latency that can be incurred by thin-client systems. The client’s workload is reduced by rendering a subset of a frame, with the server rendering the remaining portion. The full server-client system architecture is shown in Figure 2.

To avoid artifacting, lossless compression is used. As this increases the encoding time compared to lossy compression, it is advantageous for the server to render a small region at a high sampling rate, while the client can render the remaining pixels at a reduced sampling rate, akin to Fixed Foveated Rendering (FFR). We opt to have the server render the center (i.e., foveal) pixels while the client renders the peripheral pixels. The viewport is synced across both devices, with the framebuffer being partitioned across them; additionally, the client and server are kept in lockstep, with the server idling until the client displays the final frame and sends updated camera pose information to the server.

At the start of the frame, the server and client have the same camera pose. The server renders its foveal pixels at the full sampling rate, and then undergoes a lossless H.264 encoding stage for each eye. Our implementation uses CPU encoding / decoding. Once each eye has been encoded, it is transmitted over a TCP connection to the client. Meanwhile, the client renders its peripheral pixels at a

Figure 1: Illustration of the proposed client-server workload sharing model. In this example, the client (top) renders only the peripheral pixels into a reduced resolution buffer, while the server (bottom) renders only the center pixels, which are streamed to the client and merged on the client in real-time fashion.
Multiview rendering is implemented with the VK_KHR_multiview Vulkan extension. The server renders the center pixels, encodes them into an H.264 video frame, and sends the video frame to the client. The client receives that video frame asynchronously with its own rendering of the remaining pixels, decodes the video frame, and merges the two together to be displayed in one final image for the user. Often, the decoding stage finishes after the client’s rendering, delaying the merge and display stages.

| Server Dims | Network  | Decode | Merge  | Mbps  |
|-------------|----------|--------|--------|-------|
| 512x360     | 4.99     | 18.64  | 0.96   | 707.83|

Table 1: Client profiling on Sponza for our split-rendering model where the camera pose typically took < 0.6 ms. All times are median values, reported as milliseconds except for Mbps (Megabits per second).

Reduced sampling rate, while another thread receives and decodes the encoded video frame from the server. Once the client’s rendering and decoding has finished, the server’s subframe can be merged with the client’s subframe and displayed to the user. Updated pose information is then sent to the server, completing an entire frame. Multiview rendering is implemented with the VK_KHR_multiview Vulkan extension.

3 Evaluation

To evaluate our approach, we compute the median end-to-end latency (time to complete one frame), showcased in Figure 3. Additional benchmarks for each stage of the server and client’s architecture are also shown in Tables 1 and 2. The full resolution of each frame is 2400x1080 pixels, giving a resolution of 1200x1080 per eye. Each foveal region rendered by the server is 512x360 pixels per eye. A moving camera pass through Crytek’s Sponza (262k triangles, 49 textures) is evaluated over 1000 frames. The client is a OnePlus 8 phone. The full resolution for the phone is 2400x1080, with the server rendering 1024x360 pixels (512x360 per eye) at the full sampling rate.

Table 2: Server profiling when rendering for the OnePlus 8 phone client. The full resolution for the phone is 2400x1080, with the server rendering 1024x360 pixels (512x360 per eye) at the full sampling rate.

4 Conclusion & Future Work

Our experiments have shown that an image-space subdivision of labour wherein a locally networked server renders a portion of the center pixels while the mobile client renders the remaining peripheral pixels at a reduced resolution can lead to large performance improvements over native rendering on the mobile client exclusively. Our results showed a speedup of 23.05% using our image-space split-rendering method.

This work can be extended in several ways: The workloads could be swapped, with the server employing lossy compression to render the peripheral regions while the client renders the fovea, at the cost of visual artifacts appearing in the periphery. Mueller et. al showed that reshading a particular pixel often does not need to occur between frames [5]. Deepfovea showed that with adequate hardware, a small portion of pixels can be sampled to fully reconstruct a scene [3], which could allow the client to be less reliant on receiving fully rendered subframes from the server.

References

[1] Y. Dong and C. Peng. Screen partitioning load balancing for parallel rendering on a multi-gpu multi-display workstation. In EGPW@ EuroVis, pp. 69–78, 2019.
[2] J. Hladky, H.-P. Seidel, and M. Steinberger. Snakebinning: Efficient temporally coherent triangle packing for shading streaming. In Computer Graphics Forum, vol. 40, pp. 475–488. Wiley Online Library, 2021.
[3] A. S. Kaplanyan, A. Sochenov, T. Leimkühler, M. Okunev, T. Goodall, and G. Rufo. Deepfovea: Neural reconstruction for foveated rendering and video compression using learned statistics of natural videos. ACM Transactions on Graphics (TOG), 38(6):1–13, 2019.
[4] D. Li, R. Du, A. Babu, C. D.Brunar, and A. Varshney. A log-rectilinear transformation for foveated 360-degree video streaming. IEEE Transactions on Visualization and Computer Graphics, 27(5):2638–2647, 2021.
[5] J. H. Mueller, T. Neff, P. Vogt, M. Steinberger, and D. Schmalstieg. Temporally adaptive shading reuse for real-time rendering and virtual reality. ACM Transactions on Graphics (TOG), 40(2):1–14, 2021.
[6] J. H. Mueller, P. Vogt, M. Dokter, T. Neff, M. Makar, M. Steinberger, and D. Schmalstieg. Shading atlas streaming. ACM Transactions on Graphics (TOG), 37(6):1–16, 2018.