Remote Rendering for VR and Mobile Devices with Efficient Illumination Streaming

Jaroslav Kravec*
Supervised by: Jiří Bittner†

Department of Computer Graphics and Interaction
Faculty of Electrical Engineering
Czech Technical University
Prague / Czech Republic

Abstract

Wired VR headsets provide high visual quality, but restrain movement of the user by being connected to PC with cable, and untethered headsets have only mobile GPU, which has relatively low performance. We focus on providing smooth VR experience without restriction on movement: high refresh-rate and immediate effect of head movement on the rendering to prevent motion sickness. Video streaming of rendering provides high refresh-rate and quality but can also have high latency. In our approach, the scene is rendered on the server to multiple layers using depth peeling, packed to texture, and with potentially visible set of triangles streamed to the client. This method supports temporal frame up-sampling and provides low latency. Results show that it is a potential alternative to existing image-based methods and atlas streaming approaches.

Keywords: Virtual reality, remote rendering, streaming, low power client, thin client

1 Introduction

The goal of this project is to provide high-quality visualization on mobile devices with relatively low HW performance using the utilization of high-performance remote server for rendering with data streaming over wireless network. We focus on providing smooth VR experience: high refresh-rate (even with low scene update rate) and immediate effect on the rendering of head movement to prevent motion sickness.

2 Related work

The most common technique for remote rendering is video streaming, which provides high refresh-rates and high quality with relatively low requirements for transfer bandwidth. However, its problem is the latency from requesting frame to rendering it on the client, which can be more than 100ms due to network communication.

Methods like frame upsampling and image warping can be used on the client to reduce latency and to provide high framerates. Many solutions require to send additional information e.g., depth or geometry in conjunction with the color information.

Framerate upsampling is a technique to generate additional frames from already rendered frames without the need to re-render the whole scene again.

2.1 Potentially Visible Set

Potentially visible set (PVS) is a term usually referring to occlusion culling algorithms, where candidate set of potentially visible polygons are pre-computed and used to reduce the cost of frame processing by not rendering non-visible parts of the scene.

In the remote rendering context, PVS is a set of triangles, which are visible in the current frame and could be potentially visible in the next frames — to cover camera movement and rotation until the new scene data arrive. PVS is useful for reducing rendering cost on the client and for minimizing the size of transferred data.

PVS can be computed by rasterizing triangle IDs from multiple predicted camera samples [8] or with a more sophisticated method that computes it in the camera offset space [6].

2.2 Image-Based Rendering

Image-based rendering (IBR) are methods that generate and render a 3D model from sets of 2D images. In VR, it can be used to hide latency and as a frame upsampling method. The most common technique is asynchronous time warping (ATW) [9], which warps the viewport inside an overscanned framebuffer. This works well for rotation but does not handle disocclusions when the view position has changed. Advanced warping methods use ad-

*kravec.jaroslav@gmail.com
†bittner@fel.cvut.cz

ditional information e.g., depth buffer. The depth buffer is used as a geometry proxy (grid), which is rendered with perspective texture mapping [7]. That can be costly because a large number of pixels generates a large number of primitives. The grid can be reduced by using a coarse regular grid [3] or by adaptively grouping pixels into coherent blocks [2] [4]. These methods are an approximation of the true warp but provide higher performance. Reinert et al. [10] proposed a method where the server renders dual views with wide-angle non-linear projection, and the client renders it using IBR with simplified preprocessed geometry.

2.3 Object-Space Shading

Object-space shading is an alternative to image-space shading. It is a technique where shading occurs before rasterization and can be packed in some way in textures. Hladky et al. [5] proposed a method that packs shading of triangles in texture using projection-independent methods: L-packing and oversampling implemented with tessellation shader and JPEG for compression. Mueller et al. [8] proposed a method that packs rectangles (pairs of triangles) with temporal coherence between frames, which gives the ability to use MPEG compression but needs mesh preprocessing. Both methods first generate a PVS of triangles.

2.4 Compression

Remote rendering requires some compression, which depends on the rendering technique used and type of data needed to be transferred. Color can be compressed as a video stream (e.g., MPEG [8]) if given technique has space coherency between frames, or as an image (e.g., JPEG [5]).

3 Architecture

We focus on server-client architecture, designed to have a high-performance server and a relatively low-performance client. The server evaluates game logic and pre-renders part of the scene using the latest camera position provided by the client, encodes it, and sends it to the client. The data consists of geometry (triangles) and a texture with packed shading samples. The client consists of two logical loops — one for the rendering and one for the scene updating. The rendering loop repeatedly renders the scene with the latest view (reprojection of the scene) until the new scene data arrive. The update loop receives, decodes, and updates scene data in the background. This way, the scene update rate can be lower than the rendering rate (framerate upsampling). Communication between the client loops is asynchronous, and communication between the client and server is synchronous. The schematic figure of the architecture is shown in Fig. 1.

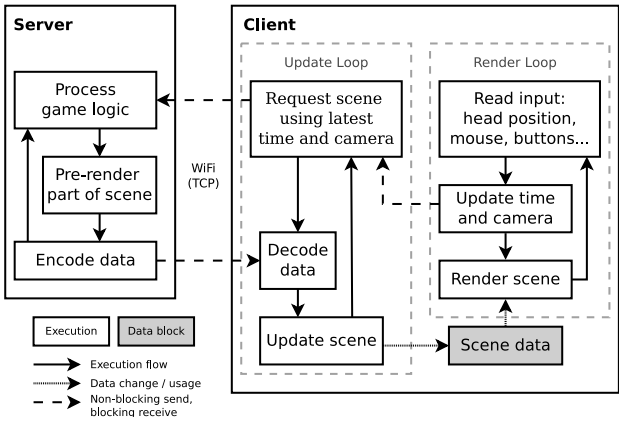

Figure 1: Application architecture. Diagram shows execution loops for the rendering on the client and server, scene updating on the client and the client-server communication.

4 Rendering on the server

The server generates a potentially visible set of triangles (PVS) for the camera view provided by the client. Then the server renders the approximate scene (PVS) using a perspective projection. We need all parts of geometry (fragments) to be stored, and because some parts can be "hidden" behind others, the server generates multiple layers of fragments. Layers are not fully covered and can contain a large number of unused areas. The server divides layers to the fixed-size tiles (e.g., 8x8px), filters out empty tiles, and packs others to one texture. The main purpose of packing is to reduce the transfer size and the client's memory usage. The client needs to receive additional data (block counts) to be able to recover tile positions within the original layers. Fig. 2 contains a schematic overview of the method. We implemented the server using OpenGL with cooperation with CUDA.

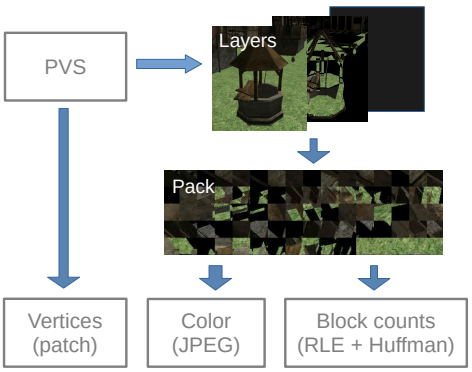

Figure 2: Server-side rendering. The server generates PVS, renders it to the layers, packs to one texture, compresses vertices and the texture, and sends it to the client.

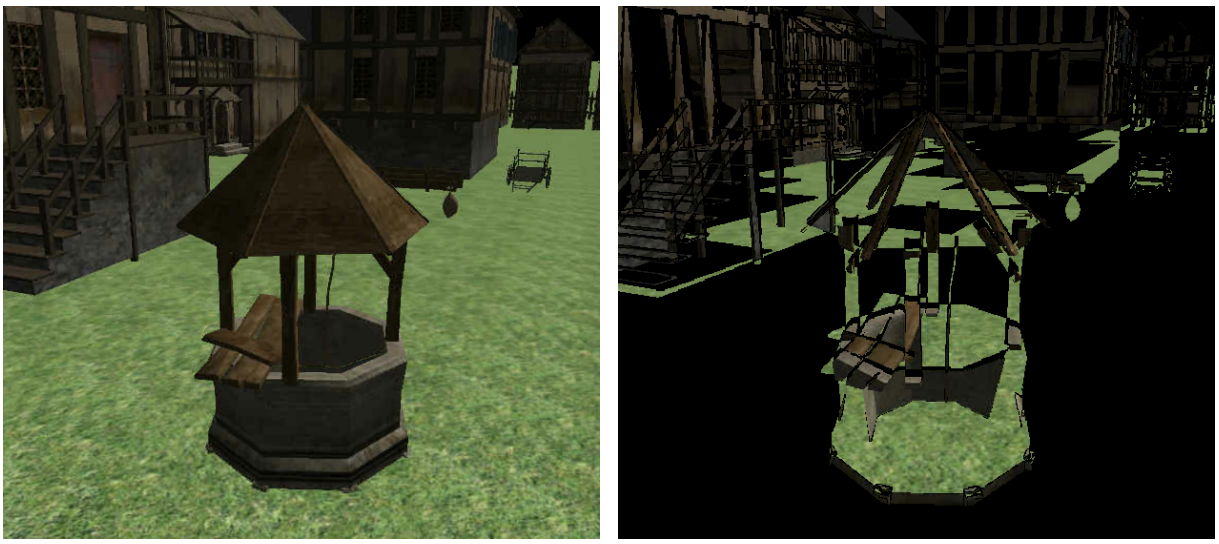

Figure 3: First (left) and second (right) rendered layer.

4.1 PVS

PVS can be created using the algorithm described in Tessellated Shading Streaming [5] or Shading Atlas Streaming [8]. For simplicity of implementation, we choose the second algorithm — rasterizing triangles from multiple predicted camera positions and rotations to cover an area where the client could potentially go within the next few frames. See Algorithm 1. The client requires a PVS in the form of vertices transformed into the global space. To minimize the transport size, we exploit spatial coherency of vertices and send to the client only the difference (patch) of the current PVS to the PVS from the previous scene update. Patch is compressed using Huffman coding.

Data: Scene triangles, current camera view
Result: PVS
Predict camera views: current, 4x corners with constant offset, 2x extrapolate position and rotation;
Mark all triangles as non-visible;
foreach *view* $\in$ *predicted views* **do**
    Render scene using wider FOV with pixel containing the triangle ID;
    Mark triangles as visible based on rendered pixels;
end
Compute new triangle indices using prefix sum;
Collect triangle vertices as a new mesh;

Algorithm 1: Algorithm for creating a PVS for a given camera view using rasterization on the GPU.

4.2 Layers

The server renders the scene (PVS) using perspective projection to several layers. We consider a layer as an image where every pixel contains a fragment closer to the camera than the next layer. The layer size is computed from the client's resolution and an enlarged field of view (FOV). An example of the first two layers is in Fig. 3.

We implemented method introduced by Yang et al. [11], we call it fragment linked lists. The method writes all fragments in one pass to per-pixel linked lists and sorts them in the second pass. Nodes of all lists are stored in a common array inside storage buffer. The head texture contains the index to the first node in their lists for every pixel. We implemented it mostly using OpenGL, with CUDA used only for fragment sorting.

An alternative method we could use is depth peeling. This method is often used for order-independent transparency. It works by rendering the scene multiple times with depth test enabled. Every rendering pass keeps the nearest fragments with a larger distance than fragments stored in the previous pass [1].

4.3 Packing

We divide layers to grids of fixed-size tiles. We use the term block to denote the array of tiles with the same position within a layer, i.e. tiles that are behind each other (see Fig. 4). The server computes the number of non-empty tiles for every block (block counts). Non-empty tiles are contiguously behind each other (starting from the first layer) because empty tile cannot exist between two non-empty tiles.

The server computes a one-dimensional index for every non-empty tile in a deterministic way. We have two alternative orders of tile indexing: block-first and layer-first order. In the block-first order, tiles within the same block

have indices next to each other. In the layer-first order, tiles within the same layer have indices next to each other — every layer is fully processed before going to the next layer. In both cases, blocks are in a row-major order. The server packs tiles to the texture also in a row-major order, with position computed from the tile index. The width of the packed texture can be different from the original layer width. When we ensure that the first layer is fully covered, the server packs it in layer-first order even when other layers use block-first order. The client can skip indirect access to the first layer and compute location directly.

The client needs to receive block counts to be able to recover original layers or to compute indices for indirect addressing. The server compresses block counts using RLE and Huffman coding. We designed the packing method to effectively support JPEG compression, which uses blocks of size 8x8px. We implemented packing using CUDA.

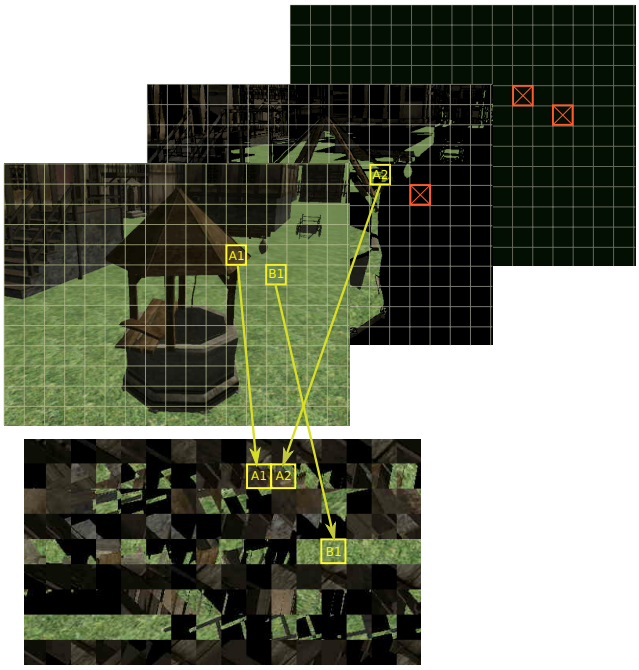

Figure 4: Packing of tiles from all layers to one texture using block-first order.

4.4 Fragment relocation

Continuous geometry (in the sense of distance from the camera) can be divided between different layers. This produces edges that are smoothed using JPEG, which leads to creating artifacts when rendered on the client with different camera position — the outline of the front geometry is visible on the geometry behind on reprojected view (see Fig. 5). Fragments within the same block can be moved to different layers to improve depth continuity, keeping the order of fragments the same. We implemented algorithm in CUDA, see Algorithm 2. The client needs to receive pixel mask for packed texture to be able to skip empty pixels during rendering or to relocate pixels back during the scene updating. The mask is compressed using RLE and Huffman coding.

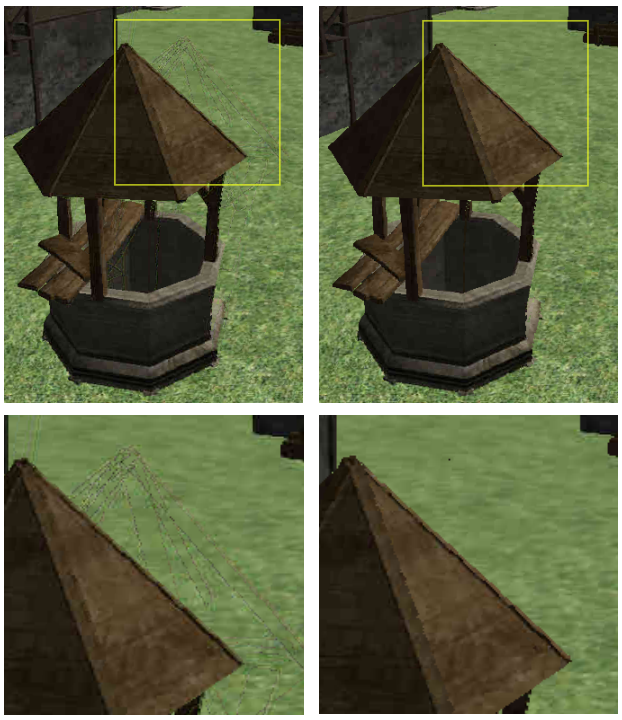

Figure 5: Comparison of the rendered image on the client without (left) and with (right) using fragment relocation. View position on the client is slightly different from the original position the server used.

4.5 Color filling

After packing to tiles, some pixels remain empty. Filling them with the averaged surrounding colors improves JPEG compression quality and reduces the size (less sharp transitions). We implemented color filling as a color dilation algorithm using CUDA. Every tile is processed independently. The algorithm repeatedly fills empty pixels with averaged color from 4-neighbors non-empty pixels until no empty pixels remain. Fig. 6 contains an example of the filling.

5 Rendering on the client

The client uses OpenGL for the rendering and contains two logical loops (see Fig. 1). The render loop corresponds to one thread with OpenGL context, and the update loop consists of two additional threads for communication and decoding.

Data: Block
foreach *layer* ∈ *block* **do**
  do
    collect fragments from the current layer with:
    - depth closer to at least one of 4-neighbors in next layers compared to the depth of neighbors in current layer
    - has space left - number of remaining fragments at the same position is lower than the number of remaining layers;
    move collected fragments to the next layer - this leads to recursive moving all fragments behind them as well;
    replace moved fragments in the current layer with empty fragment with depth computed from neighbors from the next layer
  while *moved at least one fragment*;
end

Algorithm 2: Algorithm for fragment relocation to improve depth continuity for better JPEG compression and reprojection quality on the client.

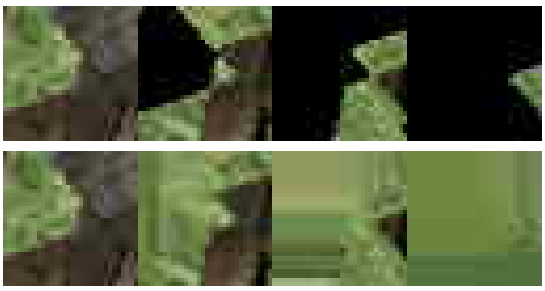

Figure 6: Color filling comparison example: before (top), after (bottom). Tiles are upscaled (original size: 32x32px).

5.1 Scene Updating

The workload of the render thread needs to be minimized to achieve stable framerate. For that, OpenGL buffers are mapped to the client's memory, and the background thread decodes data directly to them. On the render thread, only the remaining portion of updating procedure needs to be computed. Communication thread receives data and passes them using a pipe to the decoding thread. This way, the new data can be processed on the server at the same time as current data are decoding on the client. Rendering on the client requires depth layers, which are recomputed from PVS using depth peeling. The client can process depth peeling in multiple frames by limiting the number of layers that can be processed at once, which stabilizes the framerate but increases the scene update latency. We implemented an optimization, where the client uses on every layer different subset of triangles required to generate depth. The server computes subsets and sends them as a mask of all triangles compressed using RLE and Huffman coding. The client can also use fragment linked lists method for depth layer generation, but early tests show worse performance to depth peeling even without optimization. See Algorithm 3.

Data: Compressed data
Result: Scene data
`// On decoding thread`
Decompress packed texture using JPEG;
Decompress using RLE and Huffman coding:
- block counts
- empty pixel mask
- triangle subset mask for every layer;
Update vertices — apply difference patch;
Apply pixel mask to texture;
Compute triangle indices from subsets for layers;
Compute block indices from block counts — different for block-first and layer-first order;
Copy data to GPU buffers (mapped to the client memory);
`// On render thread`
Generate depth layers using the depth peeling;
Update textures from pixel buffers (packed texture, block indices);
Relocate fragments back to their original layers;

Algorithm 3: The scene updating process on the client.

5.2 Rendering

The client renders geometry applying layers as perspective projected textures. Fragment shader iterates through depth layers comparing with original depth to find the correct layer number. Depth layers are precomputed from PVS in the scene update phase. If pixels are not relocated back during the scene update, the shader iterates over color layers skipping empty pixels, otherwise the shader accesses only one color layer. The color layers are stored inside packed texture and indirectly accessed through block index texture. If the first layer is fully covered, index to its tiles are computed directly and not received from index texture.

6 Results

We implemented the server and client with C++, OpenGL, and CUDA using nvJPEG for compression on the server and TurboJPEG for decompression on the client. We focus on visual quality, bandwidth requirements, the scene update latency, server's and client's performance. We ran several tests with the camera automatically following the same prerecorded path with a duration around 2m and network speed around 30Mbps. We used resolution

1920x1080 on the client, 8x8px block size, and JPEG quality 30 with subsampling 444. The FOV was enlarged 1.1 times, which gives the layer size 2254x1214, and the first layer was always fully covered. The server has 2x CPU Intel Xeon E5-2630 v3 @ 2.4GHz, 64 GB RAM, the GPU NVIDIA GeForce GTX TITAN Black, and the client has the CPU Intel(R) Pentium(R) CPU 2020M, 8 GB RAM and the GPU Intel® HD Graphics.

6.1 Server performance

PVS computation took 11ms per scene update, rendering, and packing took 33ms and compression 30ms on average. Compressed scene size is 670kB on averages. The scene update latency on the client (time between frame requesting and rendering it) is 400ms and delta time (how often is the scene updated) is 200ms on average. Both times are large, mainly due to high bandwidth requirements. The server renders only simple diffuse shading. The average values of the server benchmark are shown in Table 1 on the right side.

6.2 Client performance

We measured the client performance on a client with a low-performance GPU integrated into the CPU, to partially simulate a mobile device. We tested both block orderings combined with two fragment relocation solutions: skipping pixels during rendering (Skip Mode) and relocation during scene updating (Relocate Mode). The block-first order gives better performance compared to layer-first (around 1.15 shorter rendering and pixel relocation time) because the block index is fetched maximum once for each processed fragment. Table 1 on the left side shows a comparison of both fragment relocation solutions using block-first ordering.

The most time consuming is the depth peeling, which in total requires about 46ms per scene update, divided into multiple frames with limit three layers per frame. The use of a lower limit of processed depth layers per frame increases the scene update latency because the scene rendering time of more frames is counted to the latency. Depth peeling without triangle subsets optimization performed 30% worse on average.

Rendering using Relocate Mode is faster (11ms compared to 20ms), but requires another 22ms in the scene update phase. In our testing, it performed better than Skip Mode (58fps compared to 43fps) because the same scene data are rendered many times (around 9 times).

6.3 Visual Quality

The server currently renders the scene using only simple diffuse shading. Visual testing investigates the degradation of the quality by compression and reprojection on the client. Visual quality highly depends on the used JPEG compression quality and the scene update rate. Fig. 7 shows the images rendered the client and differences with the original images rendered on the server.

Client		
Relocation Mode	Relocate	Skip
Render		
Sample Count	5361	3913
FPS	58.29	43.18
Frame Time (ms)	21.19	29.03
Frames / Update	9.41	7.78
Update Part (ms)	7.94	6.78
Render (ms)	10.99	19.82
Sum (ms)	18.95	26.66
Update		
Sample Count	569	503
Server Time (ms)	53.75	53.55
Updates / Second	5.19	4.72
Delta Time (ms)	199.56	225.62
Latency (ms)	379.70	423.19
Depth Peeling (ms)	46.20	45.72
Relocate Pixels (ms)	21.57	-
Texture Upload (ms)	5.74	5.74
Sum (ms)	73.51	51.46
Decompress (ms)		
Color (JPEG)	40.93	40.30
Color (Mask)	6.13	5.83
Other	3.07	5.02
Sum	50.13	51.15

Server	
Sample Count	569
Layer Count	13.53
Vertex Count	31506
Texture Height	2665
Time (ms)	
PVS	10.71
Render	3.50
Sort, Relocate, Count	9.24
Pack	6.46
Fill	3.61
Sum	33.53
Compression Time (ms)	
Color (JPEG)	4.58
Color Mask	16.21
Block Counts	7.97
Other	1.02
Sum	29.78
Compressed Size (kB)	
Vertices	77.77
Triangle – Layer Mask	5.06
Block Counts	6.56
Color JPEG	421.12
Color Mask	155.84
Sum	666.47

Table 1: Benchmark results, containing an average values over sample count in the same section or table.

7 Conclusions

We have proposed and implemented a method for remote rendering using thin client with low performance. There are several issues in the current implementation — slower client rendering and updating performance and high transfer data requirements (ca. 30Mbps) for very small scene update rate (ca. 5 per second). Visual quality is comparable to a video stream, and frame-upsampling on the client partially compensates for a low update rate.

Our primary focus will be to address the shortcomings already mentioned. The transfer size can be reduced by using a video stream compression for the first layer (e.g., MPEG) and by improving the compression technique for pixel mask. Our solution currently does not correctly handle triangles perpendicular to the projection plane and moving camera view backward.

8 Acknowledgments

The 3D model used for the test is a copyrighted material of DEXSOFT-Games (`http://www.dexsoft-games.com`).

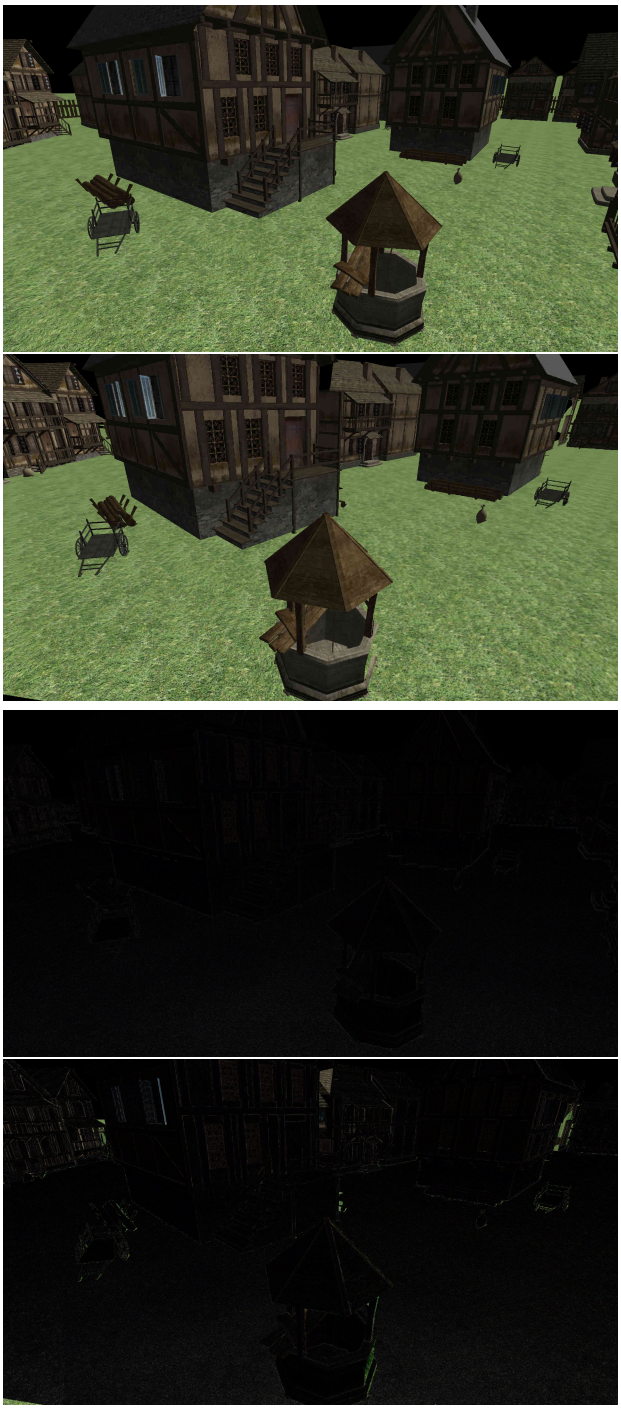

Figure 7: Images rendered on the client (top) and the differences from the original images rendered on the server (bottom). The first image shows the same view as last received scene data, and the second image shows a different view using the same scene data (slightly moved camera).

References

[1] Everitt Cass. Interactive order-independent transparency, 2020. NVIDIA.

[2] Shenchang Eric Chen and Lance Williams. View interpolation for image synthesis. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '93, page 279–288.

[3] Piotr Didyk, Elmar Eisemann, Tobias Ritschel, Karol Myszkowski, and Hans-Peter Seidel. Perceptually-motivated real-time temporal upsampling of 3D content for high-refresh-rate displays. *Computer Graphics Forum (Proceedings Eurographics 2010, Norrköpping, Sweden)*, 29(2):713–722, 2010.

[4] Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel. Adaptive image-space stereo view synthesis. In *Vision, Modeling and Visualization Workshop*, pages 299–306, 2010.

[5] J. Hladky, H. P. Seidel, and M. Steinberger. Tessellated shading streaming. *Computer Graphics Forum*, 38(4):171–182, 2019.

[6] Jozef Hladky, Hans-Peter Seidel, and Markus Steinberger. The camera offset space: Real-time potentially visible set computations for streaming rendering. *ACM Trans. Graph.*, 38(6), 2019.

[7] William R. Mark, Leonard McMillan, and Gary Bishop. Post-rendering 3d warping. *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, page 7–16, 1997.

[8] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. Shading atlas streaming. *ACM Trans. Graph.*, 37(6), 2018.

[9] OculusVR. Rendering to the oculus rift. `https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-render/`, 2020. Accessed: 2020-01-15.

[10] Bernhard Reinert, Johannes Kopf, Tobias Ritschel, Eduardo Cuervo, David Chu, and Hans-Peter Seidel. Proxy-guided Image-based Rendering for Mobile Devices. *Computer Graphics Forum*, 35(7), 2016.

[11] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz. Real-time concurrent linked list construction on the gpu. *Computer Graphics Forum*, 29(4):1297–1304, 2010.