ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kravec** Jméno: **Jaroslav** Osobní číslo: **453171**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Specializace: **Počítačová grafika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vzdálené realistické zobrazování pro VR a mobilní zařízení

Název diplomové práce anglicky:

Realistic Remote Rendering for VR and Mobile Devices

Pokyny pro vypracování:

Prostudujte metody realistického zobrazování v reálném čase vhodné pro využití v oblasti vzdáleného zobrazování založeného na klient server architektuře, kde realistické osvětlení scény je počítáno na výkonném serveru.
Navrhněte metodu vzdáleného zobrazovaní vhodnou pro VR aplikace, která bude dosahovat vysoké snímkové frekvence a rychlé reakce na změnu pozice kamery.
Vytvořte testovací implementaci, která bude realizovat výpočet osvětlení na výkonném serveru a zobrazovat výsledek na vzdáleném počítači nebo mobilním zařízení. Pro výpočet osvětlení na serveru použijte technologii OpenGL, CUDA, případně i sledování paprsků pomocí NVIDIA OptiX. Pro rekonstrukci osvětlení na klientském zařízení využijte OpenGL shaderů. Důkladně vyhodnoťte rychlost zobrazování, latenci, objem přenášených dat a maximální dosažitelnou kvalitu výstupu pro nejméně dvě testovací scény.

Seznam doporučené literatury:

[1] Hladky, J. , Seidel, H. P. and Steinberger, M. (2019), Tessellated Shading Streaming. Computer Graphics Forum, 38: 171-182.
[2] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg. 2018. Shading atlas streaming. ACM Trans. Graph. 37, 6.
[3] Reinert, B., Kopf, J., Ritschel, T., Cuervo, E., Chu, D., Seidel, H. P. (2016). Proxy-guided image-based rendering for mobile devices. In Computer Graphics Forum (Vol. 35, No. 7).
[4] Ingo Wald and Steven G. Parker. 2019. RTX accelerated ray tracing with OptiX. In ACM SIGGRAPH 2019 Courses (SIGGRAPH '19).
[6] Haines et al. Ray Tracing Gems, Apress, 2019.
[8] Lee, K., Chu, D., Cuervo, E., Kopf, J., Degtyarev, Y., Grizan, S., Flinn, J. (2015). Outatime: Using speculation to enable low-latency continuous interaction for mobile cloud gaming. In Proceedings ACM ICMSAS (pp. 151-165).
[9] Xie, N., Wang, L., Dutré, P. (2018). Reflection reprojection using temporal coherence. The Visual Computer, 34(4), 517-529.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **11.02.2020** Termín odevzdání diplomové práce: **14.08.2020**

Platnost zadání diplomové práce: **30.09.2021**

doc. Ing. Jiří Bittner, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

© ČVUT v Praze, Design: ČVUT v Praze, VIC

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.
Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Realistic remote rendering for VR and mobile devices

Bc. Jaroslav Kravec

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: Open Informatics
Study program: Computer Graphics
August 2020

Acknowledgements

Thanks to ČVUT, supervisor Jiří Bittner and my family for the support and opportunity to work on this project. The 3D models used for the Village scene are a copyrighted material of DEXSOFT-Games (`http://www.dexsoft-games.com`).

Declaration

I declare that this thesis represents my work and that I have listed all the literature used. Prague, 14. August 2020

Abstract

Wired VR headsets provide high visual quality, but restrain the user's movement 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 a 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 clients, thin clients, ray tracing

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt

Klasické VR poskytujú vysokú kvalitu vizualizácie, ale obmedzujú pohyb užívateľa, keďže sú pripojené k PC káblom a mobilné VR majú iba GPU s relatívne nízkym výkonom. Cieľom práce je dosiahnuť dobrý VR zážitok bez obmedzenia na pohyb: vysoká obnovovacia frekvencia a okamžitá zmena zobrazenia pri zmene polohy hlavy. V našom riešení, scéna je vykresľovaná na vzdialenom výkonnom serveri do viacerých vrstiev, zabalená do jednej textúry, skomprimovaná a s potenciálnou viditeľnou množinou trojuholníkov posielaná klientovi. Klient zobrazuje scénu s vyššou obnovovaciou frekvenciou, ako je aktualizovana zo servera.

Klíčová slova: virtualna realita, vzdálené vykreslování, tenky klient, sledování paprsku

Překlad názvu: Vzdálené realistické zobrazování pro VR a mobilní zařízení —

Contents

Project Specification 1

1 Introduction 1
1.1 Goals 1
1.2 Structure of the thesis 2

2 Remote rendering 3
2.1 Potentially Visible Set 3
2.2 Image-Based Rendering 4
2.3 Object-Space Shading 5

3 Global illumination 9
3.1 Rendering equation 9
3.2 Path tracing 10
3.3 Photon mapping 10
3.4 Instant radiosity 11
3.5 Denoising 11
  3.5.1 Edge-Avoiding À-Trous Wavelet Transform 11
  3.5.2 Spatiotemporal Variance-Guided Filtering 12

4 Remote Rendering - System Design 15
4.1 Rendering on the server 16
  4.1.1 PVS 17
  4.1.2 Layers 17
  4.1.3 Packing 18
  4.1.4 Fragment relocation 19
  4.1.5 Color filling 20
4.2 Compression 20
4.3 Rendering on the client 22
  4.3.1 Scene Updating 23
  4.3.2 Rendering 24
4.4 Path tracing 25
  4.4.1 SVGF with layers 25
4.5 Optimizations 25
  4.5.1 Depth peeling with triangle subsets 25
  4.5.2 Pixel mask with 'any' pixels . 26

5 Implementation 27
5.1 Asynchronous processing and communication 28
5.2 Server 29
5.3 Client 30
5.4 Configuration 30
5.5 Compression 30
5.6 Lighthouse 2 30

6 Results 33
6.1 Server Performance 34
6.2 Client Performance 34
6.3 Visual Quality 35

7 Conclusions 37
7.1 Possible improvements 37

A Bibliography 49

B Directory structure of attachment files 53

Figures

2.1 Proxy-guided Image-based Rendering for Mobile Devices: primary and extra view (source: [RKR$^+$16]). 4
2.2 Proxy-guided Image-based Rendering for Mobile Devices: (a) Mesh simplification causes background to be projected onto foreground (neck of the figure) and foreground to be cut off (missing nose). (b) Depth testing removes the incorrect background, but foreground details remain lost. (c) Encaging simplification keeps foreground details. (d) Combining both techniques gives the best results (source: [RKR$^+$16]). 5
2.3 Shading Atlas Streaming: game scenes (top) with corresponding atlas (bottom) (source: [MVD$^+$18]) 6
2.4 From the current viewpoint P_1, future viewpoints, P_2 and P_3 are predicted, with the corresponding FOV shown in color. An EVS is computed for each viewpoint, and the PVS is determined as the union of all objects visible in any EVS (visible objects are marked as white circles). In the example, object A is only visible from P_1, but not from P_2 or P_3. Conversely, object D is only visible from P3. Object C is jointly occluded by objects A and B from P_1, but becomes visible from P_2 (source: [MVD$^+$18]). 7
2.5 Tessellated Shading Streaming: oversampling packing method - they split triangles into two right-angled triangles along the longest edge and pack to shading atlas next to another. (source: [HSS19a]) 7
2.6 Tessellated Shading Streaming: tesselation pattern (top) corresponding to L-packed triangles (bottom). They cut triangles along with yellow triangles and duplicate them to allow for interpolation (source: [HSS19a]). 8
2.7 Tessellated Shading Streaming: They divide the scene triangles to gather shading via Oversampling (blue) and L-packing (green) into a shading atlas used on the client to render near ground-truth novel views (source: [HSS19a]). 8

3.1 The rendering equation (source: [RDGK12]). 9
3.2 Path tracing at a sample (orange point) sends rays in random directions (blue arrows) and bounces them, before linking them with the light (source: [RDGK12]). 10
3.3 Photon Mapping emits particles from light (yellow arrows) that are bounced and stored (yellow circles). To compute the indirect lighting at a location (orange circle), final gathering (blue arrows) or density estimation (blue circle) is used (source: [RDGK12]). 10
3.4 Instant radiosity is similar to photon mapping, but instead of density estimation or final gathering, every stored photon becomes a virtual point light (yellow dot) that sends its light (blue arrow) to all receiver samples (orange point) (source: [RDGK12]). 11
3.5 EAW: Positions of pixels with non-zero coefficients of the kernel (black dots) for three iterations of one-dimensional À-Trous wavelet. Arrows indicate pixels that are used to compute the center pixel for the next level (source: [DSHL10]). 12

3.6 EAW with the increasing number of edge weights: unfiltered ray-traced buffer (input), no additional weights: (“pure” À-Trous), ray-traced buffer only (bilateral approx.), two buffers (ray traced, normal) and all three buffers (ray traced, normal, position) (source: [DSHL10]). 12
3.7 SVGF: they demodulate direct and indirect illumination from albedo to preserve high-frequency texture detail during reconstruction filter. After filtering, they recombine illumination with albedo, apply tone mapping and temporal antialiasing (source: [SSK+17]). 13
3.8 SVGF: An overview of the core reconstruction filter. They temporally filter frame buffers (left) to get temporally integrated color and moments. They use an estimated luminance variance to drive an edge-aware spatial wavelet filter (center). The wavelet filter’s first iteration provides a color and moment history for future frames (source: [SSK+17]). 13

4.1 Application architecture. The diagram shows execution loops for the rendering on the client and server, scene updating on the client and the client-server communication. 15
4.2 Server-side rendering. The server generates PVS, renders it to the layers, packs to one texture, compresses data, and sends it to the client. 16
4.3 First (left) and second (right) rendered layer. 18
4.4 Packing of tiles from all layers (up) to one texture using block-first order (bottom). 19
4.5 Color filling comparison example: before (top), after (bottom). Tiles are upscaled (original size: 32x32px). 20
4.6 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. 21
4.7 Scene packed to texture with two full layers at the bottom and the rest of layers divided to tiles at the top. 22
4.8 Diagram of the client-side scene updating process. Blocks represent data and arrows jobs. Client decompresses received data on the CPU, uploads to GPU, generates layer using depth peeling and relocates fragments on the GPU. . . 23

5.1 Illustrative timeline showing parallelization and pipelining of the scene updating process. Every color represents an individual scene update. Both server and the client can utilize GPU and CPU simultaneously. Parts processed on the CPU are further parallelized to multiple threads. . . 28

7.1 Client rendering and scene updating. CPU / GPU / Update: accumulated CPU / GPU time of updating (decompression, depth peeling and fragment relocation); GPU / Frame: rendering and GPU tasks; Latency: elpased time between scene request and its first rendering on the client; Processing: elapsed time between start of scene processing and its first rendering on the client; Delta: time between updates. 40

7.2 Server rendering and compression times. Vertical axis represents timeline per scene update at given timepoint at horizontal axis. PVS compression starts after PVS and runs in parallel with rest of steps. Similarly Video and Pixel Mask Compression runs in parallel starting after Copy to RAM step. JPEG and other compressions are not shown due to its small time contribution (<2ms). 41
7.3 Client scene updating times. Vertical axis represents timeline per scene update at given timepoint at horizontal axis. Green / red areas shows time ranges of processing CPU / GPU tasks. Darker areas shows real time spent processing tasks (parallel on CPU serial on GPU) and brighter areas shows idle time or waiting for dependency, e.g., data from the server or GPU task for CPU task. Lighter GPU area also contains time spent on rendering of the current scene, as GPU tasks and scene rendering is interlaced. 42
7.4 Scene compressed size (individual parts are in legend) with texture size (light blue line) to show correlation. 43
7.5 Image quality (DSSIM, NRMSE) for every frame of rendered sequence with scene latency (gray line) to show correlation. 44
7.6 Client rendering errors, left images are reprojected on the client, and rights are references. Top: fast rotation causes missing geometry in PVS (black area on the right) or not having shading for geometry (gray parts of geometry), because it is outside of the rendered frustum . Middle: too perpendicular or facing away triangles doesn't have correct shading in the texture (left part, especially top green drapery), also current implementation doesn't support transparent geometry (leaves). Bottom: coming from the corner causes missing geometry as PVS creation failed to predict it. . 45
7.7 Comparisson of reprojected images on the client (left) and reference images (right) for Sponza Crytek with dynamic lights (top), Sponza Crytek with static lights (middle) and Village (bottom) 46
7.8 Village scene rendered on the client. 47
7.9 Sponza Crytek scene with dynamic lights (top and middle) and static lights (bottom) rendered on the client. Note: bottom has sharper shadows than middle, because denoise filter works better for static scenes. 48

Tables

7.1 Client benchmark statistics. 38
7.2 Server benchmark statistics. 39

Chapter 1

Introduction

Providing high-quality visualization in real-time is still difficult nowdays. One of the most resource-consuming task is to compute realistic lighting, mainly indirect lighting, and global illumination. For that, many real-time approximate techniques are used in combination with rasterization, or it is preprocessed with simulation methods, like raytracing, but then it cannot be used in dynamic scenes. Nowadays, with hardware support for ray tracing, it is possible to achieve interactive global illumination.

VR has higher requirements than real-time visualization on the PC, mainly higher framerate and low perceivable latency on the head movement. The reason is to reduce nausea and other health-related problems. In the present, we have two types of VR: tethered, that need to be connected to PC and untethered, that have a mobile computer inside them. It is not a problem to achieve high quality with tethered VR, but they limit a person to a smaller area, and untethered VR does not have sufficient performance for computation of realistic illumination.

We can achieve high-quality visualization and not be connected with the cable using remote rendering on a high-performance server and streaming to a thin client on untethered VR, but wireless transmission has lower bandwidth compared to tethered connection and often larger latency. Therefore we study methods that get around these limitations and use hardware of both devices simultaneously: compute resource-demanding parts on high-performance server and the rest on mobile VR.

1.1 Goals

The goal of this project is to provide high-quality visualization with dynamic global illumination on mobile devices with relatively low HW performance using the utilization of high-performance remote server for rendering with data streaming over common wireless network (40Mbps). 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. The following technologies will be used: OptiX and CUDA on the server for ray tracing and data processing, and OpenGL on the client for rendering.

1.2 Structure of the thesis

Chapter 2 contains analysis of existing methods for remote rendering and chapter 3 of global illumination techniques. In chapter 4 we present our remote rendering method, and in chapter 5 its implementation details. We evaluate performance and visual quality in chapter 6, and finally summarize the work in chapter 7.

Chapter 2

Remote rendering

Remote rendering represents techniques to offload rendering tasks to another computer. We focus on real-time remote rendering methods, where rendered data are streamed to the client instantaneously. These techniques usually provides a frame-rate upsampling method that allows to render multiple frames from the different viewpoints on the client for a short period of time from the same data. This way, the scene can be updated with a lower rate and still have a high refresh rate on the client and immediately respond to the camera rotation or movement. 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 higher latency from requesting frame to rendering it on the client. Many solutions require to stream additional information, e.g., depth or geometry in conjunction with the color information.

2.1 Potentially Visible Set

Potentially visible set (PVS) is a term usually referring to occlusion culling algorithms, where candidate set of potentially visible objects or polygons are pre-computed and used to reduce the cost of frame processing by not rendering non-visible parts of the scene. Laakso [M.03] provided an overview of these techniques, including the exact 3D solution.

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 few 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 [MVD$^+$18] or with a more sophisticated method that computes it in the camera offset space [HSS19b].

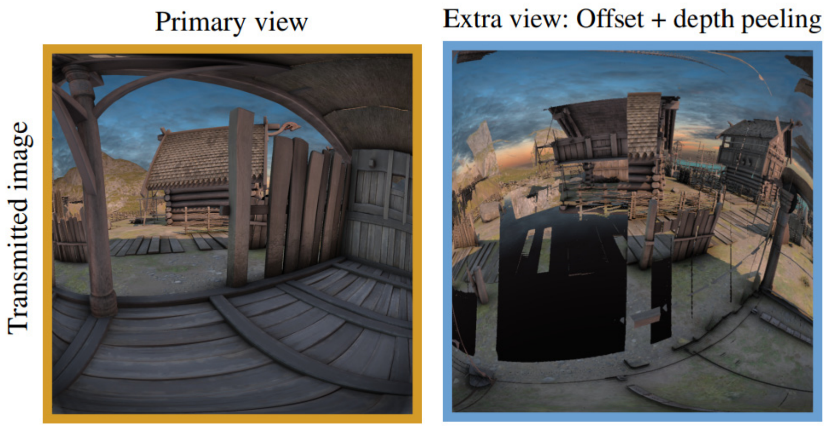


Figure 2.1: Proxy-guided Image-based Rendering for Mobile Devices: primary and extra view (source: [RKR+16]).

2.2 Image-Based Rendering

Image-based rendering (IBR) are methods that generate and render a 3D model from sets of 2D images. It can be used to hide latency and as a frame upsampling method. Asynchronous time warping (ATW) by Oculus is a technique used in VR that shifts the rendered image to adjust for changes in head movement [Ocu20]. This technique works well for rotation but does not handle disocclusions when the view position has changed. Advanced warping methods use additional information, e.g., depth buffer. The depth buffer is used as a geometry proxy (grid), which is rendered with perspective texture mapping [MMB97]. 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 [DER+10] or by adaptively grouping pixels into coherent blocks [CW93] [DRE+10]. These methods are an approximation of the true warp but provide higher performance.

Reinert et al. [RKR+16] proposed a method where the server renders dual views with wide-angle non-linear projection (see Fig. 2.1), and the client renders it using IBR with simplified preprocessed geometry. They use hemispherical fish-eye projection modified that horizontally stretches each row to fill the full height of the texture. With such projection, straight lines in world-space turn into curves in screen-space, therefore they employ tesselation shader to adaptively subdivide triangles to cover less than a pixel. They render a primary view from the last known client location and extra view with quarter resolution from an offset location to provide extra visibility coverage and depth-peeling to prevent the inclusion of redundant pixels from the primary view. They use preprocessed geometry proxy on the client, which is faster to render, because of lower triangle count, but causes artifacts in silhouettes, when background samples are projected on the foreground (Fig. 2.2a) or has missing parts of foreground geometry (e.g. nose in Fig. 2.2a). To solve this, they use a strictly encaging simplification of the original model

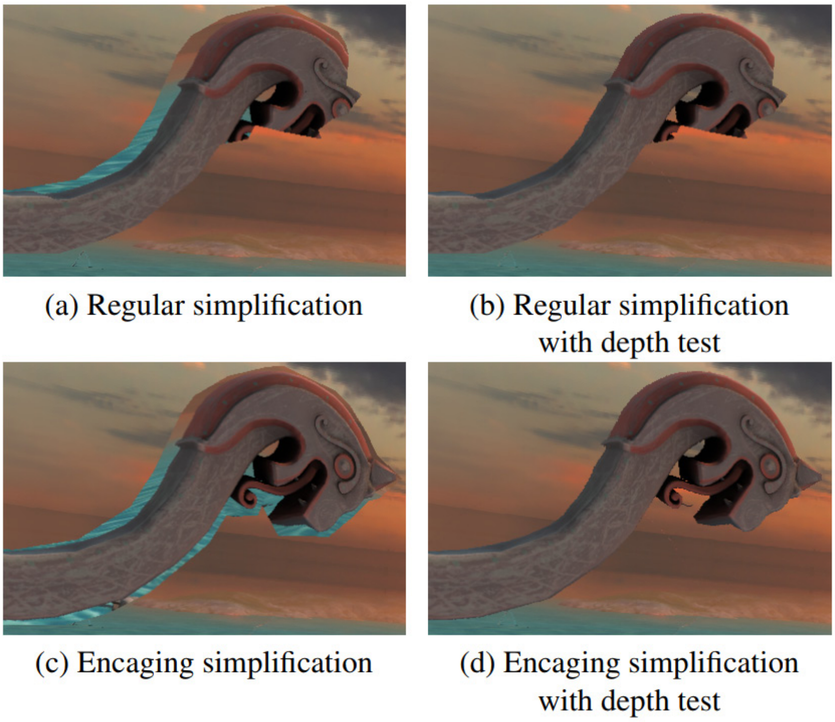

(a) Regular simplification
(b) Regular simplification with depth test
(c) Encaging simplification
(d) Encaging simplification with depth test

Figure 2.2: Proxy-guided Image-based Rendering for Mobile Devices: (a) Mesh simplification causes background to be projected onto foreground (neck of the figure) and foreground to be cut off (missing nose). (b) Depth testing removes the incorrect background, but foreground details remain lost. (c) Encaging simplification keeps foreground details. (d) Combining both techniques gives the best results (source: [RKR+16]).

(Fig. 2.2c) and custom depth test to discard fragments with larger error distance compared to transmitted depth maps (Fig. 2.2c,d).

Lochmann et al. extended common diffuse and opaque image warping technique to the reflective and refractive case. They use a ray tree of RGBZ images, where each node contains one RGB light path, which is to be warped differently depending on the depth Z and the type of path: diffuse, reflective, and refractive. The warp diffuse flow using a simplified version of Image Warping algorithm [BMS+12] and specular flow using their novel method based on modified gradient descent procedure to find pixel with the closest reflected/refracted direction.

2.3 Object-Space Shading

Object-space shading is an alternative to image-space shading, where shading occurs before rasterization, independently on the screen-space, and can be often packed in some way in textures. It exploits world-space coherency instead of image-space coherency.

Mueller et al. [MVD+18] proposed a method that packs patches (a group

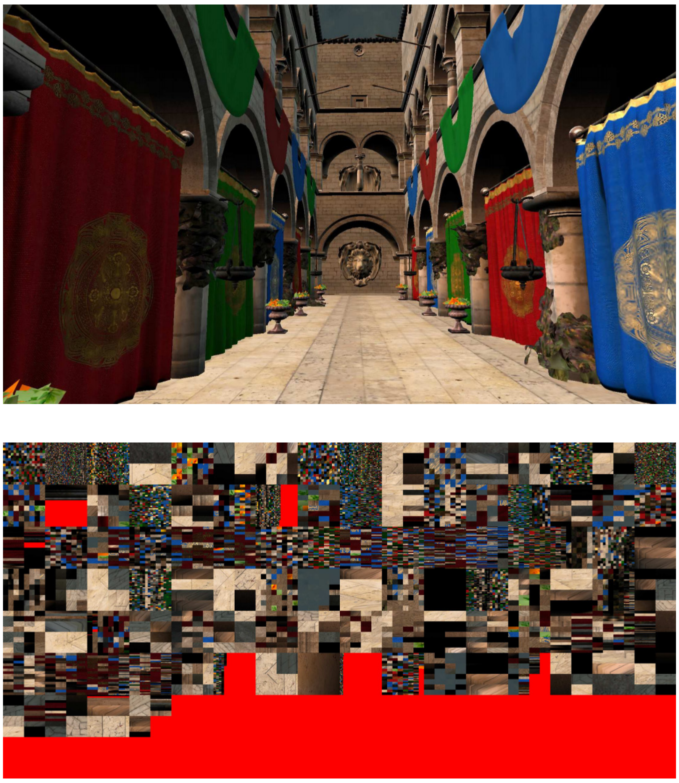


Figure 2.3: Shading Atlas Streaming: game scenes (top) with corresponding atlas (bottom) (source: [MVD+18])

of two or three adjacent triangles preprocessed from triangle meshes) to a texture atlas, which is streamed using MPEG, see Fig. 2.3 for an example. The processing of every frame on the server consists of several steps. The server determines the visibility of patches (PVS) by generating and merging of multiple exact visibility sets (EVS). EVS is computed for a specified viewpoint by rastering of patches to id buffer with depth test enabled. Viewpoints for EVS are determined from the prediction of head motion for a short period into the future, see Fig. 2.4. Every patch inside PVS is mapped from screen-space to rectangles in atlas space. The method provides a parallel atlas memory allocation system, which keeps allocated patches visible in subsequent frames unchanged and fills gaps with newly added patches. It improves temporal coherence for MPEG compression. Patches are shaded in object-space and stored inside theirs allocated locations in the atlas.

Hladky et al. [HSS19a] proposed a similar method, which packs shading of triangles to two texture atlases using two methods: oversampling for slanted triangles and L-packing for other cases, see Fig. 2.7 . Fig. 2.6 shows the structure of L-packed triangles. When a triangle is classified for oversampling, they split it by the longest edge into two right-angled triangles and pack it to separate atlas next to another, see Fig 2.5. The size of triangles is computed from the screen-space. They exploit tesselation shader to dynamically launch threads for effective packing triangles of different sizes. They adjust sample

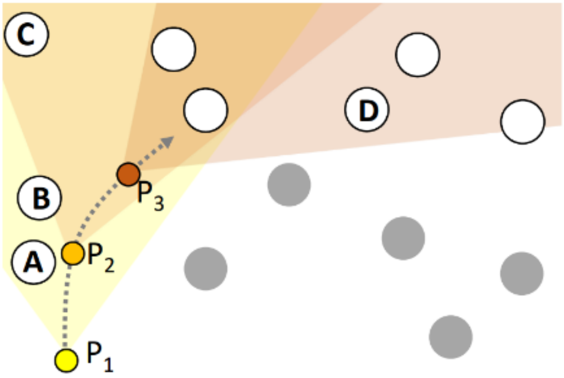
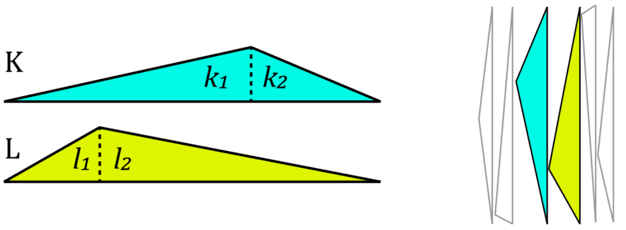


Figure 2.4: From the current viewpoint P_1, future viewpoints, P_2 and P_3 are predicted, with the corresponding FOV shown in color. An EVS is computed for each viewpoint, and the PVS is determined as the union of all objects visible in any EVS (visible objects are marked as white circles). In the example, object A is only visible from P_1, but not from P_2 or P_3. Conversely, object D is only visible from P3. Object C is jointly occluded by objects A and B from P_1, but becomes visible from P_2 (source: [MVD$^+$18]).



(a) *Screen space* **(b)** *Atlas space*

Figure 2.5: Tessellated Shading Streaming: oversampling packing method - they split triangles into two right-angled triangles along the longest edge and pack to shading atlas next to another. (source: [HSS19a])

positions to increase screen-space uniformity [MHAM08]. Shading atlases are compressed using JPEG. They compute PVS using their method that uses the camera offset space [HSS19b].

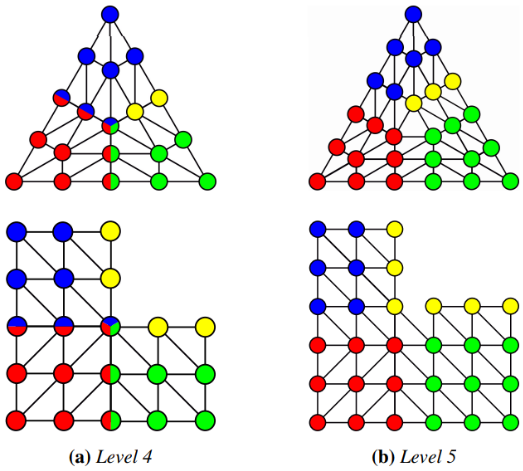


Figure 2.6: Tessellated Shading Streaming: tesselation pattern (top) corresponding to L-packed triangles (bottom). They cut triangles along with yellow triangles and duplicate them to allow for interpolation (source: [HSS19a]).

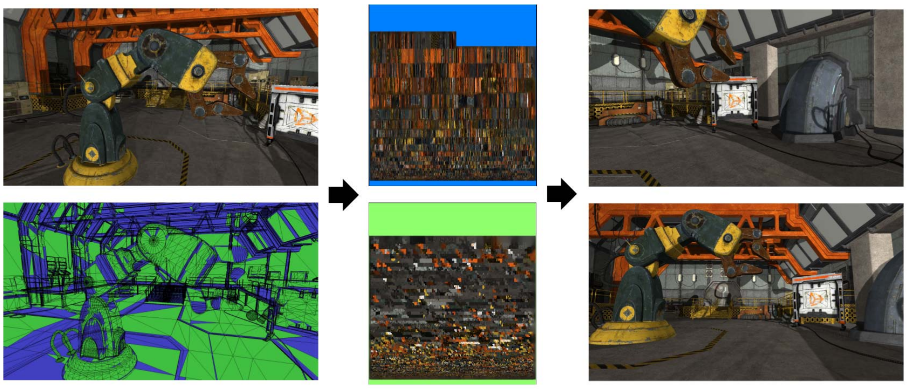


Figure 2.7: Tessellated Shading Streaming: They divide the scene triangles to gather shading via Oversampling (blue) and L-packing (green) into a shading atlas used on the client to render near ground-truth novel views (source: [HSS19a]).

Chapter 3

Global illumination

Global illumination is a term used for techniques that are used to produce more realistic lighting in the 3D scenes by taking into account not just direct lighting – light that comes directly from the light source, but also indirect lighting – light often reflected multiple times from other surfaces. A good survey of such techniques and related methods is covered in [RDGK12]. This chapter contains an overview of the few most commonly used techniques, including path tracing that is used in our project.

3.1 Rendering equation

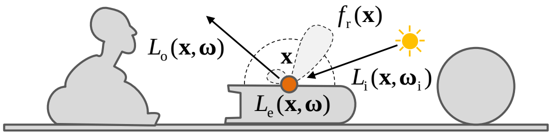

$L_o(\mathbf{x},\boldsymbol{\omega})$
$f_r(\mathbf{x})$
$\mathbf{x}$
$L_i(\mathbf{x},\boldsymbol{\omega}_i)$
$L_e(\mathbf{x},\boldsymbol{\omega})$

Figure 3.1: The rendering equation (source: [RDGK12]).

Light transport between surfaces are computed based on the rendering equation [Kaj86] (Fig. 3.1):

$$L_o(x,\omega) = L_e(x,\omega) + L_r(x,\omega) \tag{3.1}$$

which states that outgoing radiance L_o at the surface located at x in direction ω is equal to sum of emitted radiance L_e and reflected radiance L_r, which is computed as:

$$L_r(x,\omega) = \int_\Omega L_i(x,\omega_i)\ f_r(x,\omega_i \rightarrow \omega)\ max(0, N(x)\cdot\omega_i)\ d\omega_i \tag{3.2}$$

where Ω is upper hemisphere oriented around the normal $N(x)$ at x and f_r is bi-directional reflectance function (BRDF) [Nic65], that determines the amount of light for different incoming (ω_i) and outcoming (ω) directions at specified location (x).

Solving rendering equation requires visibility determination between surfaces and lights, for which ray tracing is most commonly used. Such computation can be computational expensive and therefore uses spatial data structures such as k-d tree or bounding volume hierarchy (BVH) to speed-up ray intersection computation.

3.2 Path tracing

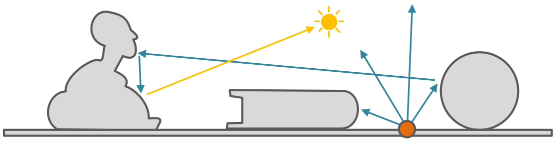


Figure 3.2: Path tracing at a sample (orange point) sends rays in random directions (blue arrows) and bounces them, before linking them with the light (source: [RDGK12]).

The rendering equation can be approximated using Monte Carlo techniques [Kaj86]. A high number of directional samples (paths) are tracked using ray tracing, and the result is an average of samples belonging to the same pixels. Paths start from the camera origin as primary rays for every pixel and bounces repeatedly creating other rays based on BRDF function, see Fig. 3.2. The path needs to operate some light or emissive material to have a contribution, therefore usually from every hit point on the path is shot shadow ray to randomly selected light source (next event estimation). Path tracing has many variants and improvements. Importance sampling is a technique where rays are sent more often in the directions where it is expected that the rendering equation will have hight values. Bi-directional path tracing shots particles from the camera and selected light source at the same time and connect them using shadow rays [LW93].

3.3 Photon mapping

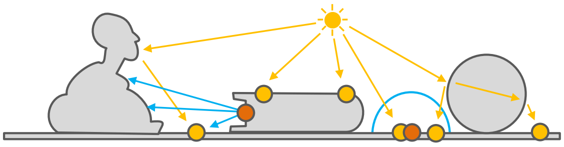


Figure 3.3: Photon Mapping emits particles from light (yellow arrows) that are bounced and stored (yellow circles). To compute the indirect lighting at a location (orange circle), final gathering (blue arrows) or density estimation (blue circle) is used (source: [RDGK12]).

Photon mapping [Jen96] computes illumination in two passes: first, a large number of photons are path traced from light sources and stores in a photon map at each hit point. A Photon map is a spatial data structure, like kd-tree, that allows for effective searching for the nearest points. The second pass computes the final image by tracing rays from the camera and estimating radiance from photons closest to its first hit point, see Fig. 3.3. This method is suitable for rendering caustics and uses for its second photon maps, where photons in the first pass are shot onto objects with refractive materials. For specular reflection is used Monte Carlo path tracing.

3.4 Instant radiosity

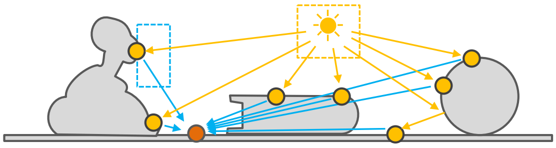


Figure 3.4: Instant radiosity is similar to photon mapping, but instead of density estimation or final gathering, every stored photon becomes a virtual point light (yellow dot) that sends its light (blue arrow) to all receiver samples (orange point) (source: [RDGK12]).

Instant radiosity [Kel97] is similar to photon map. In the first pass, photons are emitted from light sources and bounced inside the scene. The second pass is different, every stored photon is considered as virtual point light (VPL). Indirect light is computed by gathering – shooting rays to VPLs or rasterization with shadow maps, instead of density estimation as used in photon mapping, see Fig. 3.4.

3.5 Denoising

Monte Carlo path tracing requires a large number of samples to converge to a result with acceptable noise. This is not possible for real-time rendering using current hardware, including recent GPU with support for ray tracing. To get around this, denoising algorithms were created, that filter raytraced output in post-process, often in screen-space. This approach adds some bias compared to the ground-truth reference but provides visually plausible results in a small amount of time.

3.5.1 Edge-Avoiding À-Trous Wavelet Transform

Dammertz et al. created Edge-Avoiding À-Trous Wavelet Transform for fast Global Illumination Filtering (EAW) [DSHL10]. À-Trous wavelet transform computes discrete wavelet transform by repeating convolution with generating kernels. Each step number of non-zero coefficients in kernel is same, but are filled in between with zeros as shown in Fig. 3.5. The filter radius is doubled each step.

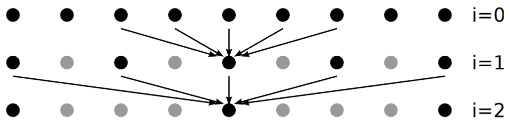


Figure 3.5: EAW: Positions of pixels with non-zero coefficients of the kernel (black dots) for three iterations of one-dimensional À-Trous wavelet. Arrows indicate pixels that are used to compute the center pixel for the next level (source: [DSHL10]).

Edge-avoiding filtering is achieved using a data-depending weighting function. They extend the intensity-based edge-stopping function of the bilateral filter to combine multiple edge-stopping functions computed from the ray-traced image, normal buffer, and position buffer, see Fig. 3.6

Figure 3.6: EAW with the increasing number of edge weights: unfiltered ray-traced buffer (input), no additional weights: ("pure" À-Trous), ray-traced buffer only (bilateral approx.), two buffers (ray traced, normal) and all three buffers (ray traced, normal, position) (source: [DSHL10]).

3.5.2 Spatiotemporal Variance-Guided Filtering

Schied et al. proposed Spatiotemporal Variance-Guided Filtering (SVGF) [SSK$^+$17] based on EAW. They use path-tracer that outputs direct and indirect illumination separately and rasterization to generate G-buffer: depth, object-space normals, mesh IDs. Illumination is demodulated from albedo before filtering, see Fig. 3.7. In other words, they filter untextured illumination components and reapply texture after reconstruction.

Their reconstruction filter performs three main steps: temporally accumulating one spp path-traced inputs to increase the sampling rate, using these temporally augmented color samples to estimate local luminance variance, and use this variance to drive a hierarchical à-trous wavelet filter, see Fig. 3.8. In the end, they apply temporal antialiasing (TAA).

For temporal accumulation, they back project samples from the current frame to screen-space position in prior frame, similarly like TAA, and check the depth, normal, and mesh IDs for consistency. Consistent samples are accumulated using exponential moving average.

The key idea of using variance to drive wavelet filter is that reconstruction should less change samples in regions with less noise (e.g., fully shadowed regions) while altering more in sparsely sampled, noisy regions. They estimate the per-pixel variance from the first and second accumulated raw moments.

Edge-stopping functions are, similarly as EAW, based on depth, object-space normals, and luminance.

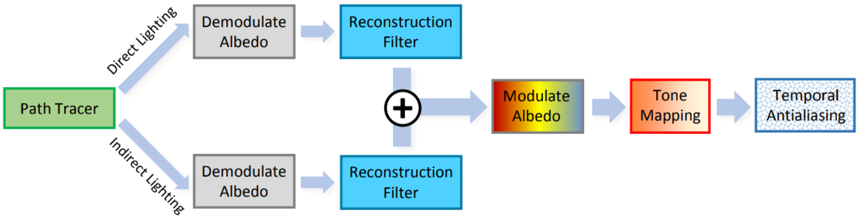

Path Tracer
Direct Lighting
Indirect Lighting
Demodulate Albedo
Reconstruction Filter
Demodulate Albedo
Reconstruction Filter
Modulate Albedo
Tone Mapping
Temporal Antialiasing

Figure 3.7: SVGF: they demodulate direct and indirect illumination from albedo to preserve high-frequency texture detail during reconstruction filter. After filtering, they recombine illumination with albedo, apply tone mapping and temporal antialiasing (source: [SSK+17]).

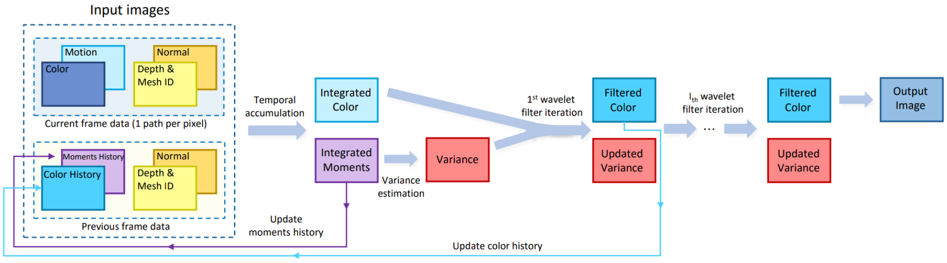

Input images
Motion
Color
Normal
Depth & Mesh ID
Current frame data (1 path per pixel)
Moments History
Color History
Normal
Depth & Mesh ID
Previous frame data
Temporal accumulation
Integrated Color
Integrated Moments
Variance estimation
Variance
Update moments history
1st wavelet filter iteration
Filtered Color
Updated Variance
Ith wavelet filter iteration
...
Filtered Color
Updated Variance
Output Image
Update color history

Figure 3.8: SVGF: An overview of the core reconstruction filter. They temporally filter frame buffers (left) to get temporally integrated color and moments. They use an estimated luminance variance to drive an edge-aware spatial wavelet filter (center). The wavelet filter's first iteration provides a color and moment history for future frames (source: [SSK+17]).

Chapter 4

Remote Rendering - System Design

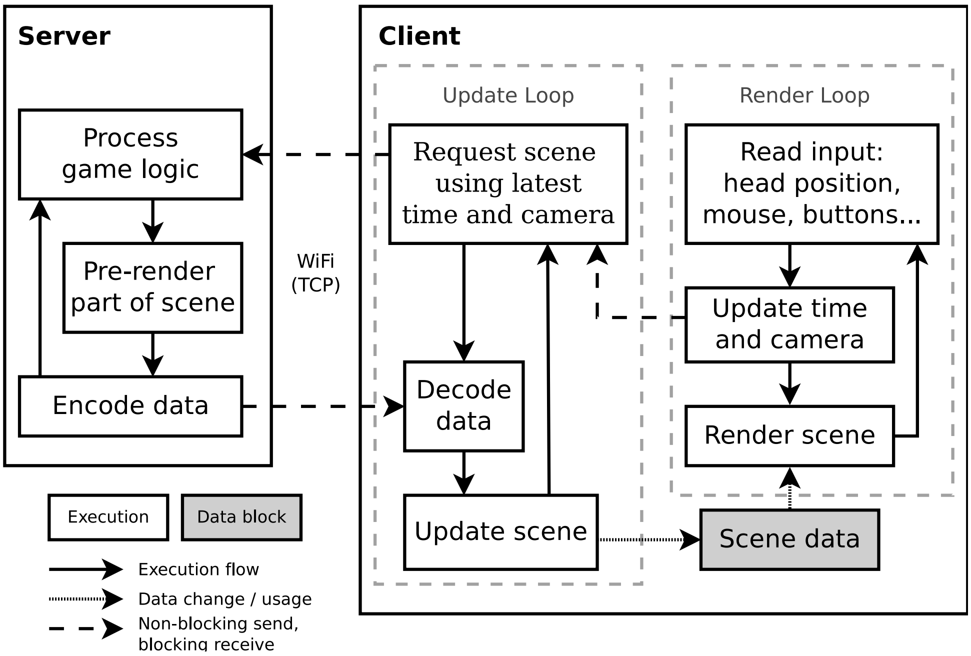

Server
Process game logic
Pre-render part of scene
Encode data
WiFi (TCP)
Client
Update Loop
Request scene using latest time and camera
Decode data
Update scene
Render Loop
Read input: head position, mouse, buttons...
Update time and camera
Render scene
Scene data
Execution
Data block
Execution flow
Data change / usage
Non-blocking send, blocking receive

Figure 4.1: Application architecture. The diagram shows execution loops for the rendering on the client and server, scene updating on the client and the client-server communication.

Our solution is based on the streaming of multiple layers rendered using perspective camera packed to one texture skipping unused partitions. We have been inspired by Reinert's et al. method [RKR$^+$16], which uses two layers with depth peeling and by object-space methods [MVD$^+$18] and [HSS19a] to pack shading data effectively using blocks to one texture. Our method is designed to use lossy image and video compression (like JPEG or H264) and provide a way to work around some visual glitches caused by them: fragment relocation and color filling. All processing steps behind some compression methods are implemented using the GPU.

We use client-server 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 from the latest viewpoint provided by the client, encodes it, and sends it to the client. The data consists of geometry (triangles) and a compressed texture with packed shading samples. The client receives data, decodes, updates the scene in the background and repeatedly renders it with the latest viewpoint, until new data arrive, to support framerate upsampling. Fig. 4.1 shows the architecture diagram.

4.1 Rendering on the server

The server first 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 and optionally other data to improve visual quality or to reduce processing time on the client. Fig. 4.2 contains a schematic overview of the method.

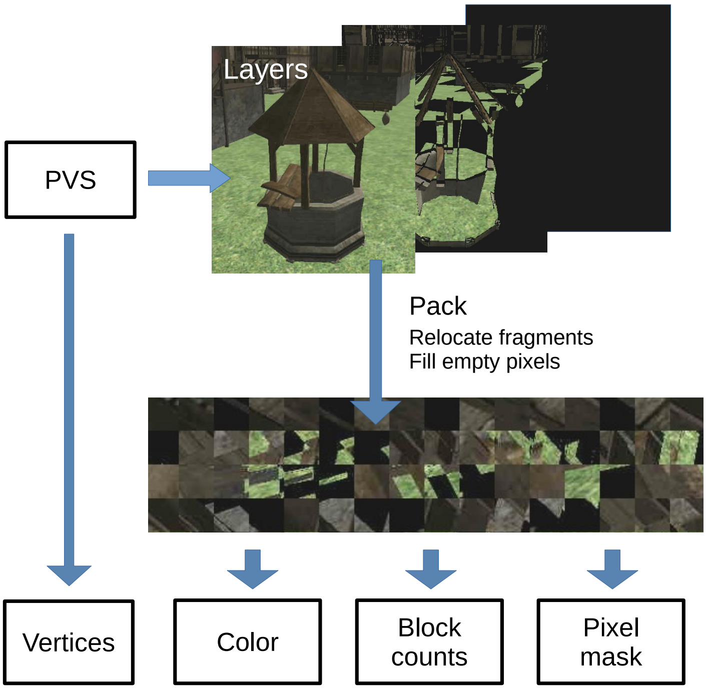


Figure 4.2: Server-side rendering. The server generates PVS, renders it to the layers, packs to one texture, compresses data, and sends it to the client.

4.1.1 PVS

PVS can be created using the algorithm described in Tessellated Shading Streaming [HSS19a] or Shading Atlas Streaming [MVD^+18]. For simplicity of implementation, we choose the second algorithm — rasterizing triangles from multiple predicted viewpoints to cover an area where the client could potentially go within the next few frames. The field of view (FOV) is slightly enlarged to the FOV the client uses to cover small rotations. We predict viewpoints the way [MVD^+18] do – current viewpoint and 2x extrapolation in time, but we use additional four viewpoints: corners on the plane perpendicular to the main viewpoint. The offset of the corners to the main viewpoint is computed based on the server frame rate. See Algorithm 1. The client receives a PVS in the form of vertices transformed into the global space.

```
Data: Scene triangles, current camera view
Result: PVS
Predict camera views: current, 4x corners with adaptive offset, 2x
  extrapolate in time; 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.

4.1.2 Layers

The server renders the scene (PVS) using perspective projection to several layers. We consider a layer an image where every pixel contains a fragment closer to the camera than the next layer. We use the same enlarged FOV as the PVS uses. The layer size is computed from the client's resolution and an enlarged FOV. An example of the first two layers is in Fig. 4.3.

Layers can be generated using rasterization or by ray-tracing. We implemented both methods, rasterization mostly for development purposes and ray-tracing for realistic rendering with global illumination.

We implemented the rasterization method introduced by Yang et al. [YHGT10], we call it fragment linked lists. The method rasters 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.

An alternative rasterization 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 [Cas20].

With ray tracing, we generate layers by repeatedly shooting primary rays from the same origin and in the same direction for every pixel, but we increase the nearest ray distance to skip already captured geometry in previous layers. Because we need to store in layers only geometry in PVS, we add an additional test to ignore other triangles based on their IDs. This test is used only for primary rays, bounced, and shadow rays uses the whole scene.

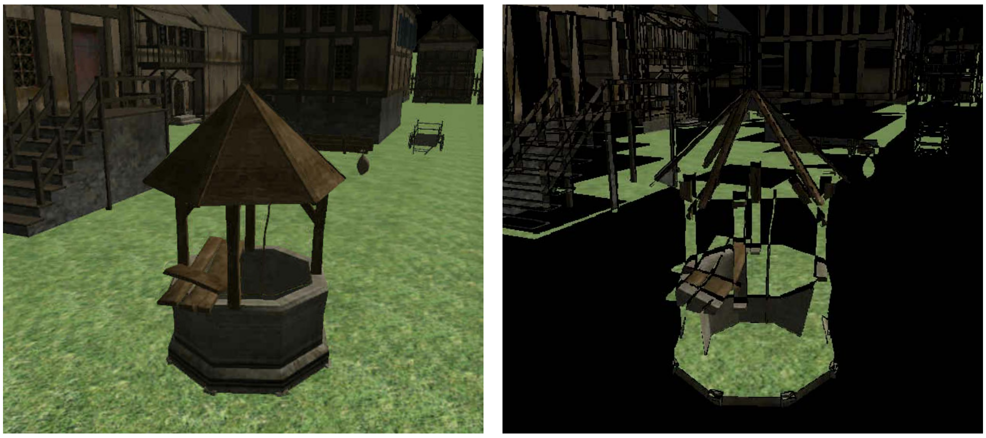


Figure 4.3: First (left) and second (right) rendered layer.

4.1.3 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. 4.4). 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 row-major order. The server packs tiles to the texture also in row-major order, with position computed from the tile index. We designed the packing method to effectively support JPEG compression, which uses blocks of size 8x8px. The client needs to receive block counts to be able to recover original layers or to compute indices for indirect addressing.

The first layer is often fully covered. Therefore we choose to stream it (or first several layers) without packing. We call them full layers. This way, they can be compressed effectively using video compression, and the client accesses them directly. We store full layers in the start of the same texture as blocks, each under another, see Fig. 4.7 for example.

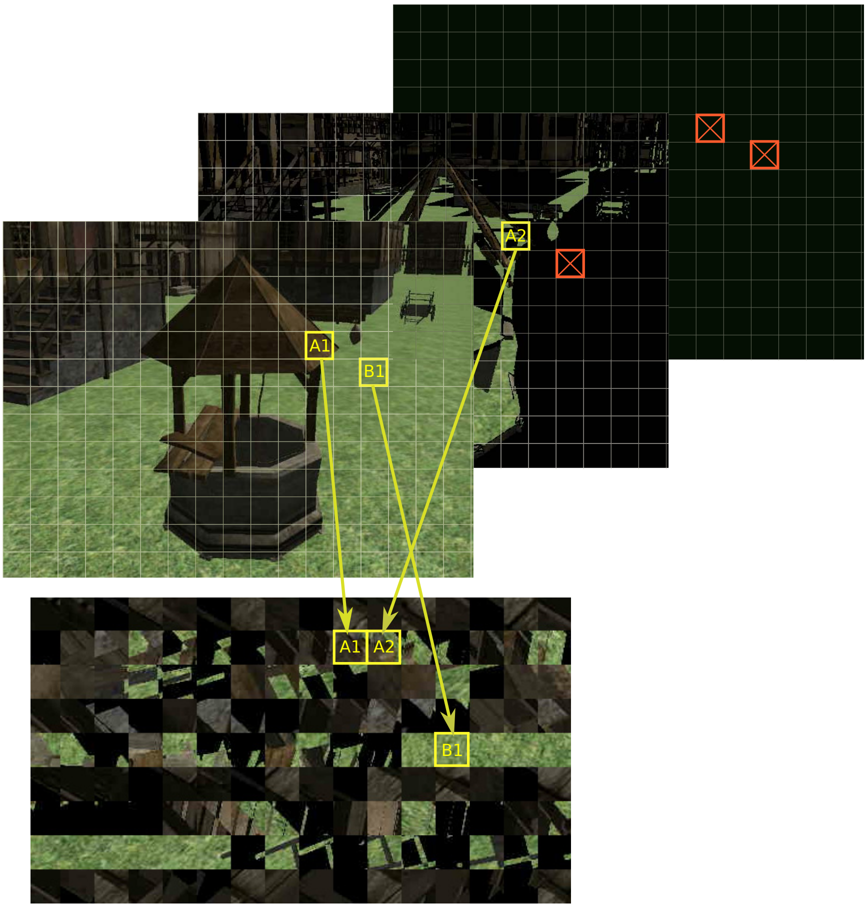


Figure 4.4: Packing of tiles from all layers (up) to one texture using block-first order (bottom).

4.1.4 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 lossy compression like JPEG, which leads to creating artifacts when rendered on the client with different viewpoints — the outline of the front geometry is visible on the geometry behind on reprojected view (see Fig. 4.6). Fragments within the same block can be moved to different layers to improve depth continuity, keeping the order of fragments the same, see Algorithm 2. The client needs to receive a pixel mask for packed texture to be able to skip empty pixels during rendering or to relocate pixels back during the scene updating.

```
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 on the GPU to improve depth continuity for better JPEG compression and reprojection quality on the client.

4.1.5 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). 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. 4.5 contains an example of the filling.

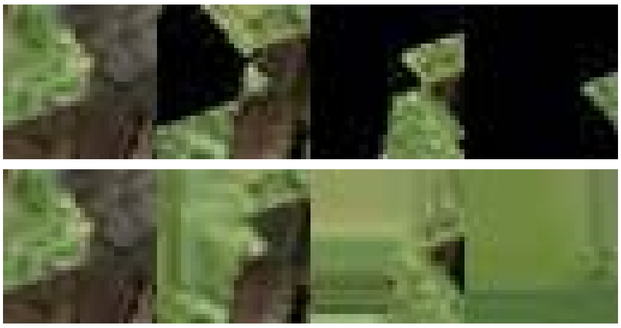


Figure 4.5: Color filling comparison example: before (top), after (bottom). Tiles are upscaled (original size: 32x32px).

4.2 Compression

For color texture, we use JPEG or video compression with codec H264. Video compression does work well only for full layers, because other parts do not

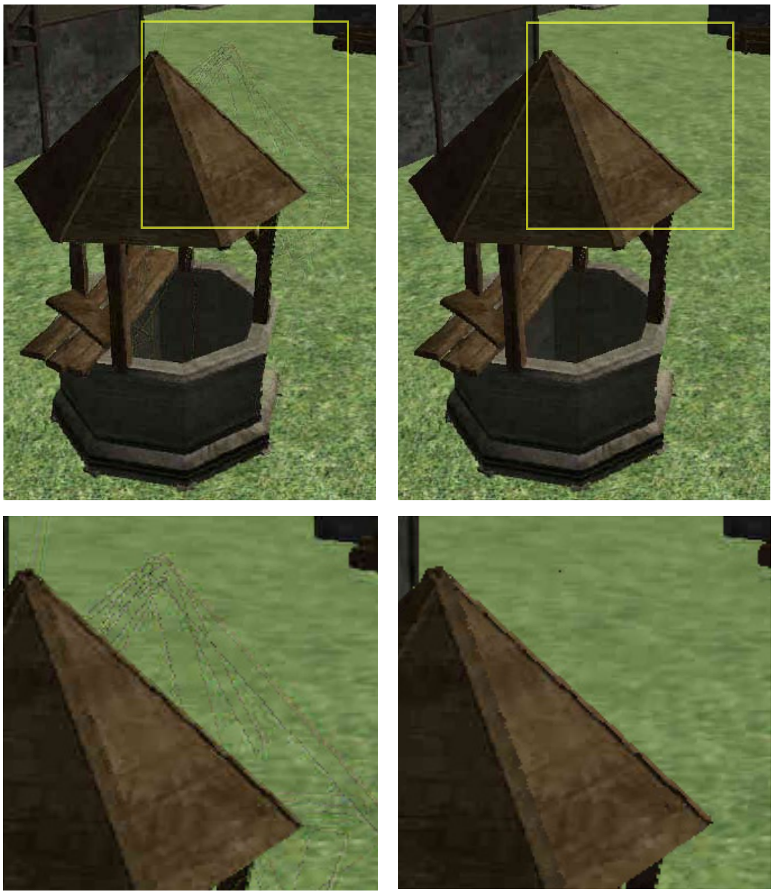


Figure 4.6: Comparison of the rendered image on the client without (left) and with (right) using fragment relocation. View position on the client is slightly different from the original position the server used.

have image-space coherence outside of tile – neighboring tiles are affected using video compression, that causes visual artifacts after reprojection on the client even with fragment relocation. JPEG compresses every 8x8px block independently, therefore, it has no such problems. We hybrid approach to give the best visual quality to compressed size ratio: one full layer compressed with H264 and rest of blocks using JPEG. The quality of both compressed parts is adjusted separately, which gives more flexibility.

We compress pixel mask, required for fragment relocation, using binary quadtree method [Sam84]. We tried combination binary RLE with Huffman coding, but it gives a smaller compression ratio (10 to 30% compared to quadtree).

To minimize the transport size of PVS, 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. We compute PVS difference in linear time from triangle IDs, which are implicitly sorted due to the way the PVS generation works. For new triangles, we first try to find the same vertices in previous PVS and reference to it. This gives 2-4 bytes for index instead of 12 bytes for the whole vertex. Similarly, every new vertex is sent only once and is referenced from subsequently added triangles when need. The patch is additionally compressed with Huffman encoding.

We compress other data (e.g., block counts) using Huffman encoding on the top of the RLE.

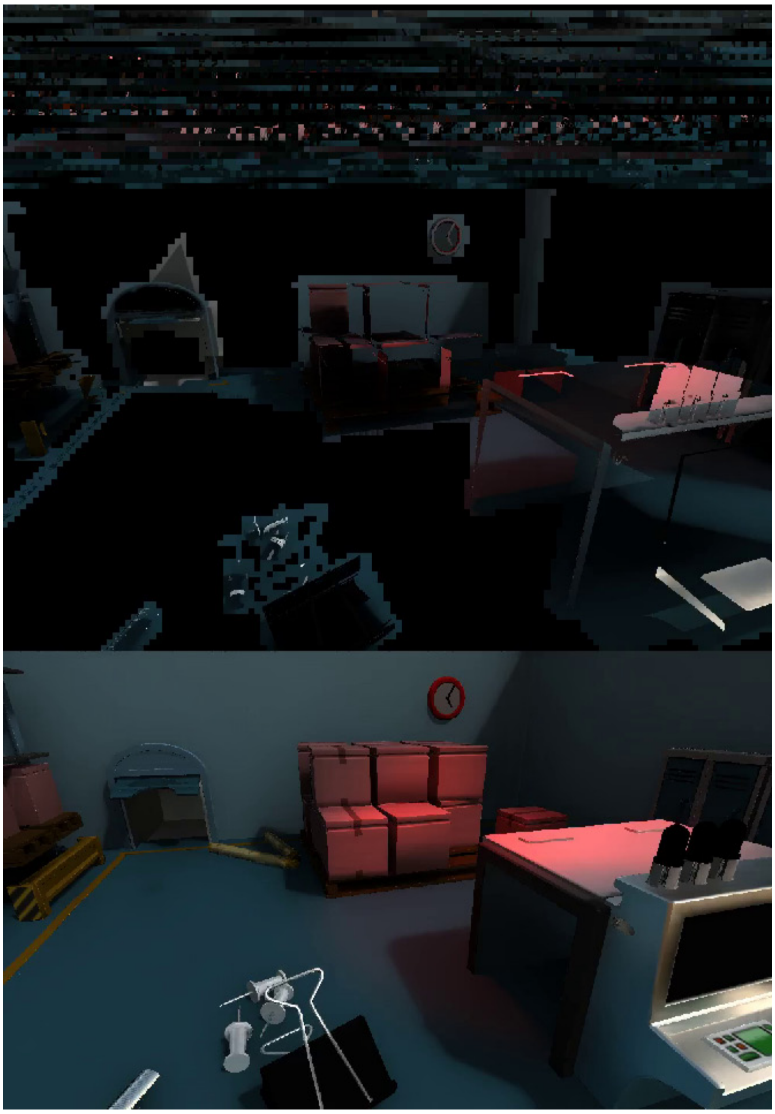


Figure 4.7: Scene packed to texture with two full layers at the bottom and the rest of layers divided to tiles at the top.

4.3 Rendering on the client

The client consists of two logical loops, one for the rendering and one for the scene updating, see Fig. 4.1). The rendering loop repeatedly renders the scene with the latest view (reprojection of the scene) until the new scene is available. The update loop receives, decodes, and updates scene data in the background. Communication between the client loops is asynchronous.

4.3.1 Scene Updating

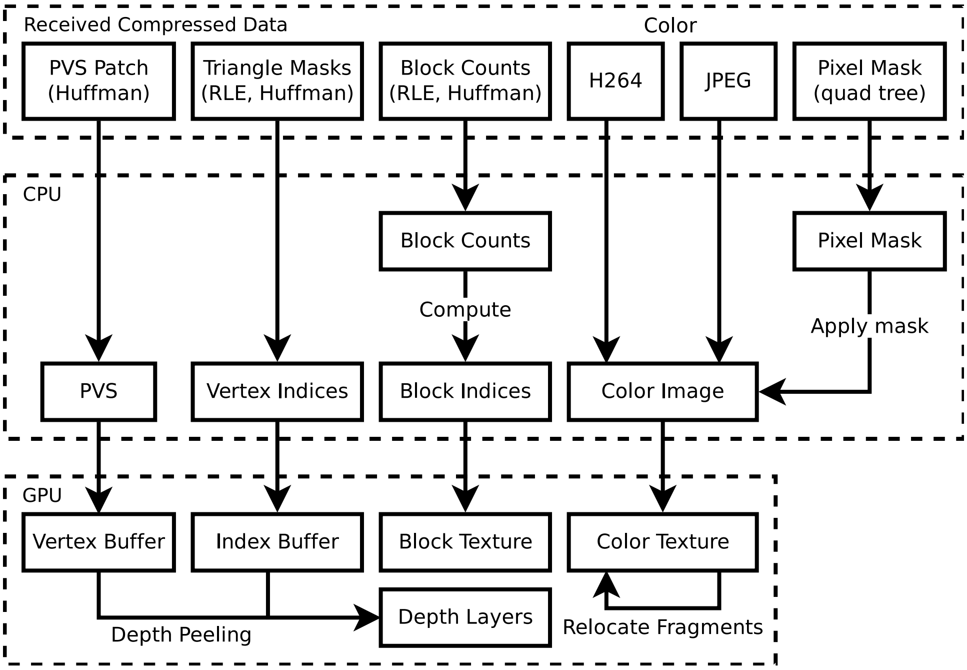

Received Compressed Data
Color
PVS Patch (Huffman)
Triangle Masks (RLE, Huffman)
Block Counts (RLE, Huffman)
H264
JPEG
Pixel Mask (quad tree)
CPU
Block Counts
Pixel Mask
Compute
Apply mask
PVS
Vertex Indices
Block Indices
Color Image
GPU
Vertex Buffer
Index Buffer
Block Texture
Color Texture
Depth Layers
Depth Peeling
Relocate Fragments

Figure 4.8: Diagram of the client-side scene updating process. Blocks represent data and arrows jobs. Client decompresses received data on the CPU, uploads to GPU, generates layer using depth peeling and relocates fragments on the GPU.

The client needs to preprocess new scene data received from the server before they can be rendered. Fig. 4.8 shows the task diagram of the scene updating process. The client first decompresses data using several methods, based on data type: RLE for block counts and triangle masks, H264 and/or JPEG for color, and quadtree for pixel mask. From block counts, the client computes block indices, that are used during rendering to compute tile positions in color texture. The client uses a pixel mask to mask out pixels in the color image using the alpha channel: zero alpha means the pixel is unused. The client uploads decompressed and computed data to the GPU.

Until this point, processing occurs in the background on the CPU, expect for video decompression, if hardware decoding is available, and individual tasks can perform in parallel, as they are often independent on each other. The client then computes on the GPU depth layer using depth peeling from PVS vertices and optionally indices per layer from triangle masks, if triangle subsets optimization is enabled (see section 4.5.1 for details).

If the pixel mask is provided, the client can relocate fragments on the GPU back to their original locations during the scene preprocessing, or unused pixels will be skipping during the rendering. Relocating back increases preprocessing time, but speed up rendering, as layers can be directly accessed through block indices and not incrementally traversed.

Scene preprocessing performed on the GPU occurs in parallel with the rendering of the previous scene. If hardware or API does not support parallel execution, it needs to be interleaved with the rendering of frames. To prevent framerate shuttering by having spikes created from preprocessing tasks, we can divide them into smaller tasks and execute them within a few frames, not just one. Depth peeling can be limited to process only a few layers at once and fragment relocation by processing only a smaller portion of texture at once.

4.3.2 Rendering

The client renders geometry using the latest viewpoint, applying layers as perspective projected textures. Vertex shader transform geometry to two screen-space coordinate systems: current space using the latest viewpoint available on the client, and original space using the viewpoint that the server used for rendering. The current space is for rasterization and original for texture projection. Fragment shader iterates through depth layers, comparing it with depth from the original space to find the correct layer number. If pixels are not relocated back during the scene update, the shader iterates over color layers skipping empty pixels, otherwise it accesses only one color layer. Fragment shader computes location to color texture from original screen coordinate, layer number, and block texture as described in algorithm 4.1.

```
vec4 getColor(vec2 screenCoord, layer) {
  if(layer < fullLayersCount)
    // skip block texture and compute position directly
    return texture(colorTexture, vec2(0, layer * frameHeight) + screenCoord);
  else {
    // offset layer index, skip full layers
    layer = layer - fullLayersCount;

    if(useLayerFirstOrder) {
      // layer-first: 3D texture has index for every tile from every layer
      tileIndex = texture(blockTexture, vec3(screenCoord / blockSize, layer));
    } else {
      // block-first: 2D texture has index of first tile for every block
      tileIndex = texture(blockTexture, screenCoord / blockSize) + layer;
    }

    tilePosition = vec2(tileIndex % tilesInRow, tileIndex / tilesInRow) * blockSize;
    positionOffset = vec2(0, fullLayersCount * frameHeight);
    colorCoord = positionOffset + tilePosition + screenCoord % blockSize

    return texture(colorTexture, colorCoord);
  }
}
```

Algorithm 4.1: Retrieval of color texel for specific screen coordinate and layer from packed color texture using index from block texture.

4.4 Path tracing

To create more realistic global illumination, we use real-time Monte Carlo path tracer with SVGF [SSK+17] for denoising, extended to support multiple layers. Layer generation using raytracing is described in section 4.1.2. The path tracer implements Disney BRDF [MHH+12], supports punctual lights, area lights, and environmental map illumination. Every hitpoint at the path is connected with shadow ray to randomly selected light source (next event estimation). It uses Russian roulette based on survival probability of path, to determine when to end stop bouncing and multiple importance sampling for light sources. We trace more samples per pixels to improve quality and provide better per-pixel variance estimation for the denoiser.

4.4.1 SVGF with layers

The main problem is that SVGF works in screen-space, and we have multiple layers – more fragments corresponding to one pixel. Wavelet decomposition uses 5 x 5 cross-bilateral filter each step. We apply it to every fragment, but at every sample position of filter, where originally only one pixel would be, we iterate over all fragments looking for the fragment closest to the plane in world-space created from the main (center) fragment and its normal. The plane is shifted in the normal direction based on world-space distance to the camera and screen-space distance of processing sample to the center of the filter. The main purpose of shifting is to prevent the inclusion of rear hidden surfaces when more surfaces are close behind each other (e.g., a picture on a wall).

4.5 Optimizations

During implementation we found some optimization, that does noticeably increase performance or reduce compressed size.

4.5.1 Depth peeling with triangle subsets

The client uses depth peeling to generate layers, but it can be time-consuming on a weaker GPU. We found that not all triangles are required to be rasterized on every layer e.g., triangles fully visible in front layers will not have an effect on further layers, and fully obscured triangles in a given layer is pointless to draw to it or layers before it.

We implemented an optimization, where the client uses on every layer different subset of triangles, which is enough to generate correct depth. The server computes subsets during layer generation and packing and sends them as a mask of all triangles compressed using RLE and Huffman coding. For the server computation is no very time consuming because all necessary data (triangles IDs for every pixel in every layer) has available from layer generation, only compression remains.

This optimization reduces the time of layer generation on our tested client by 10 to 40%, but requires to send around 3-5kB more data per scene update.

4.5.2 Pixel mask with 'any' pixels

Inside the same block, some location can have a lower number of layers than the number of tiles the block has and therefore pixels behind them doesn't contain useful information. For the pixel mask, it means that it doesn't matter which value will be stored for them. Compression for these pixels selects values that will more reduce the final compressed size.

We implemented it for both methods: quadtree and RLE. In both cases, it lowered a compressed size to about 10 to 30%. From the implementation perspective, the packing phase provides a ternary mask instead of binary with values: zero, ones, and 'any' for every pixel, and the compression method will take care of the rest.

Chapter 5

Implementation

We implemented server and client in C++14 with CMake as build system. We use cross-platform technologies, but implementation was tested only on Linux Manjaro and Ununtu 18.04. All dependencies are open-source, behind NVIDIA proprietary technologies, that are required only for the server.

Both (server and client) uses:

- OpenGL , GLEW
- Boost: Asio for TCP communication and property tree for configuration management
- GLM - linear algebra library
- FFmpeg - video codecs
- FiniteStateEntropy (FSE) [1] - for Huffman encoding
- PpluX [2] - for task oriented multi-threaded scheduler

Client additionally uses:

- GLFW 3 - window management and input
- TurboJPEG - JPEG decompression
- ImGui [3] - user interface

Server additionally uses:

- CUDA
- EGL - OpenGL context management
- nvJPEG - JPEG compression supporting on NVIDIA GPU
- Assimp - scene loading
- Lighthouse 2 [4] - real-time ray tracing framework
- Optix7 (from Lighthouse 2) - ray tracing platform using CUDA

---

[1] Link: `https://github.com/Cyan4973/FiniteStateEntropy`

[2] Link: `https://github.com/pplux/px`

[3] Link: `https://github.com/ocornut/imgui`

[4] Link: `https://github.com/jbikker/lighthouse2`

5.1 Asynchronous processing and communication

We implemented asynchronous communication using messages. The client repeatedly sends to the server latest viewpoint (UPDATE_VIEWPOINT) and estimated optimal scene update delta time (SET_UPDATE_DELTA) determined from its the scene update throughput (network, scene decoding and GPU updating average times from previous updates), and the server determines for which viewpoints render scene and streams it to the client (UPDATE_SCENE). This pattern allows for pipelining and parallel processing of many steps: rendering on the server, compression, streaming, scene decoding, and updating on the client, see Fig. 5.1. We currently support only one client simultaneously.

We use task-based approach for parallelization of scene processing on the server (Fig. 4.2) and updating on the client (Fig. 4.8). Scheduler supports CPU and GPU tasks. CPU tasks run in parallel in multiple threads, and GPU tasks are processed sequentially in one dedicated, that have OpenGL context. On the client, GPU tasks need to be processed concurrently with the rendering of the current scene. We dedicate the only portion of time between the rendering of the scene on the client for GPU tasks, to prevent framerate shuttering when many tasks are scheduled at once. The elapsed time of GPU tasks is monitored using OpenGL time queries. We divide long-duration tasks into smaller tasks where possible, e.g., depth peeling consists of one task for every layer; this way, more stable framerate can be achieved.

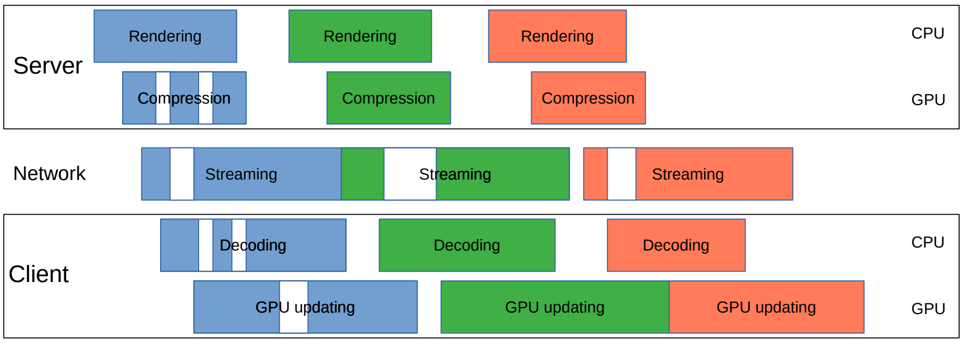


Figure 5.1: Illustrative timeline showing parallelization and pipelining of the scene updating process. Every color represents an individual scene update. Both server and the client can utilize GPU and CPU simultaneously. Parts processed on the CPU are further parallelized to multiple threads.

List of messages (client → server):

- UPDATE_VIEWPOINT - send latest camera viewpoint
- SET_UPDATE_DELTA - request the scene update delta (1 / update rate)
- SET_SETTINGS - send settings parameters in INFO format
- GET_ALL_SETTINGS

- GET_SCENE_LIST
- START_BENCHMARK
- STOP_BENCHMARK

List of messages (server → client):

- UPDATE_SCENE - send scene data
- STOP_BENCHMARK - return benchmark statistics in INFO format
- GET_ALL_SETTINGS - return all server settings in INFO format
- GET_SCENE_LIST - return list of scenes in INFO format

5.2 Server

The server supports currently only NVIDIA GPUs, because it requires technologies CUDA and Optix7. Alternatively, we could use cross-GPU technologies like OpenCL or Vulkan, but because at the time of development, only NVIDIA supports hardware-based ray tracing, we decided to use CUDA for better performance, easier development and intercooperation with ray tracer. OpenGL is used for the rasterization step of PVS generation and layer generation for development purposes. Other parts of the packing method are implemented using CUDA and compression methods on the CPU. We implemented an efficient parallel version of prefix sum using CUDA, that is used to compute tile indices and to filter triangles for PVS.

We generate layers to per-pixel linked lists of fragments (see structure 5.1) and process all subsequent steps (see Algorithm 3) with this data structure. In packing, color filling, and fragment relocation we map our block to CUDA block with the same size (one thread per one pixel with all its layers) and use shared memory to speed up processing (e.g., computing number of layers per block using atomic maximum operation inside shared memory).

Fragment relocation (Algorithm 2) does not create new layers, but instead, it computes for every fragment number of layers to skip during packing – the number of empty layers between current and previous fragment (*skip* variable in structure 5.1). Algorithm copies layer to shared memory, repeatedly "move" fragments to next layer, by increasing *skip* variable and updates it back to global memory before going to the next layer.

```
struct Fragment {
    int next; //index of next fragment or -1 for the last fragment
    glm::u8vec4 color; //RGBA
    float depth; //distance from camera
    int skip; //number of layers to skip (used for fragment relocation)
    int id; //triangle ID (used for triangle-layer optimization)
};
```

Algorithm 5.1: Structure of fragment used in server for rendering.

```
compute PVS (OpenGL + CUDA);
raytrace/raster layers into per-pixel linked lists (OptiX/OpenGL);
relocate fragments (CUDA, optional);
count number of tiles for every block (CUDA);
compute tile indices using prefix sum (CUDA);
allocate and copy fragments to texture (CUDA);
fill empty pixels (CUDA, optional);
```

Algorithm 3: Rendering steps on the server before compression with technologies that it uses.

5.3 Client

The client requires only OpenGL 4.2 and is designed to be relatively easy ported to OpenGL ES, required for mobile devices. The client contains two logical loops (see Fig. 4.1). The render loop corresponds to one thread with OpenGL context, which also handles GPU tasks, and the update loop consists of one thread for receiving data and other threads for handling CPU tasks. The scene updating process is scheduled to CPU and GPU tasks, as described in the section 5.1. We map OpenGL buffers to the client's memory, which allows the background threads to decode data directly to them, reducing time on the render thread.

5.4 Configuration

Both applications load configuration from files. The client can dynamically change the configuration of the server (without the need of restarting it) from GUI. We use INFO format from boost property tree for storing configuration in files and for transferring it through the network.

5.5 Compression

Behind already mentioned technologies, we implemented a few custom compression techniques: binary and 8-bit RLE, binary quad-tree, PVS difference – patches. Although for RLE and quad-tree existing methods can be found, they do not support 'any' pixel optimization (see section Optimizations in chapter Solution). All custom made compressions are implemented on the CPU.

5.6 Lighthouse 2

Lighthouse 2 [5] is an open-source framework for real-time ray tracing/path tracing experiments implemented in C++. It is highly modular and supports multiple platforms, currently: Optix 7, Optix 5 Prime and Vulkan. It supports scenes from glTF and obj format

---

[5] Link: `https://github.com/jbikker/lighthouse2`

It consists of three layers:

- The application layer - application logic and handles user input
- The RenderSystem - scene I/O and host-side scene storage
- The render cores - low-level rendering functionality

We use it as a framework for path tracing with denoising. Our server acts as an application layer for the Lighthouse 2. We choose RenderCore_Optix7Filter as base render core. It has path-tracing implemented using Optix7 and variant of SVGF for denoising in CUDA. We modified it to add support for PVS for primary rays, generating multiple layers and multi-layer denoising. We also needed to expand API, therefore change RenderSystem, to be able to provide PVS to render core and return layers to the application layer. PVS generation, packing and compression, and streaming is implemented outside of Lighthouse 2. Algorithm 4 shows path tracing and denoising steps after our modification.

```
trace and store primary rays to generate all layers (can be replaced
  with rasterization);
repeat sample count times
    generate secondary rays from primary using different randomness
      than previous sample;
    while path length < max path length and bath is not empty do
        trace rays;
        shade;
        generate rays for next batch;
        increase path length;
    end
    accumulate moments and color for filter;
end
prepare filter:
- reproject fragments to the previous frame
- compute variance from moments
- gaussian blur variance
- merge variance and color with reprojected data using exponential
  moving average;
repeat filter phase count times
    apply a phase of À-Trous filter
end
```

Algorithm 4: Path tracing and denoising steps in Lighthouse 2.

Chapter 6

Results

We focus our benchmark tests on visual quality, bandwidth requirements, the scene update latency, server's, and client's performance. We ran tests with the camera automatically following the prerecorded path on three scenes: Sponza Crytek with dynamic lights, and Sponza Crytek and Village with static lights. All scenes use path tracing with four samples per pixel and denoise filter with 5 À-Trous iterations. A Path duration in Sponza Crytek is about 10s and in Village about 30s. We used resolution 1920x1080 on the client, 8x8px block size, first layer compressed with video codec h264 with CRF 25, and rest layers with JPEG quality 25 with subsampling 444. The FOV was enlarged 1.1 times, which gives the layer size 2254x1214. We enabled vertical synchronization on the client and therefore clamped framerate to maximum 60fps. The server has 2x CPU Intel Xeon E5-2630 v3 @ 2.4GHz, 64 GB RAM, the GPU NVIDIA GeForce RTX 2080 Ti, and the client has the CPU Intel(R) Core(TM) i5-4200M, 12 GB RAM and the integrated GPU Intel® HD Graphics 4600. We choose the client with a relatively less powerful GPU to partially simulate a mobile device. The client and server were in different remote locations (distant over 500km) with network throughput max 50Mbps and latency around 17ms. Table 7.1 contains scene and client's performance statistics and table 7.2 contains server's performance, path trace and compressed size statistics. Figures 7.1, 7.2, 7.3, 7.4, 7.5 shows graphs with various benchmark variables on timelines of rendered sequence.

The scene update rate was about 5.4 updates per second on average, which is about 180ms between updates (delta time), but the scene latency per frame (time between update request and rendering it on the client) is 380ms in average, see graphs in Fig. 7.1. The average processing time (rendering on the server, transfer, updating on the client) is 250 ms on average. Delta time is shorter than latency, and the processing time is between them because of pipeline processing and asynchronous communication. The average compressed size of the update was 300kB, maximum size 550kB, that gives around 1.5MBps (12mbps) transferred data trough network, much bellow maximum network throughput. Graphs in Fig. 7.4 shows compressed sizes for scene updates including its parts. The largest part is JPEG (100kB in average), following video and pixel mask (each 50kB in average) and PVS (80kB in Sponza Crytek and 30kB in Village). Size of PVS and video varies more than other data types as they are not stateless – they actualize data from the previous scene update.

6.1 Server Performance

Performance in all three tested cases was about the same. PVS computation took 3.5ms per scene update, packing 2.8ms and path tracing without filter around 20ms on average, see table 7.2 and graphs in Fig. 7.2 for details. The most time consuming is the denoise filter, which took around 70ms on average, but can get up to 120ms in Village scene and up to 200ms in Sponza Crytek scene. Denoise filter has the largest timer variance as its performance highly depends on layer count. PVS, video and pixel mask compression ran on one CPU thread each and took around 26ms each, but they run in parallel. Rest compression methods, including JPEG, took bellow 1ms each. JPEG is faster compared to video because we currently use NVJPEG for JPEG that runs on GPU and FFmpeg for video, that runs on CPU. For pixel mask, we currently used single-threaded CPU quad-tree compression that can be parallelized to multiple threads with the help of GPU in the future. In summary, scene processing time on the server was 135ms on average, which can give a maximum update rate 7.4 (compared to real 5.5). That partially explains not reaching maximum network throughput.

6.2 Client Performance

Scene rendering took 4.8ms in Sponza Crytek scenes and 3.5ms in Village scene on average per frame, that is much bellow 16ms limit for 60fps, but GPU task took additional 4ms on average per frame with spikes up to 30ms in all scenes, and 80ms at the beginning of Village scene, see graphs in Fig. 7.1. That occasionally created framerate shuttering. Better distribution of GPU tasks between frames can mitigate it but can increase scene latency.

Scene decompression took 66ms of CPU time on average. Video decompression is the most time consuming with 30ms on average, and then JPEG and pixel mask with 16ms each on average. Each other decompression tasks took below 3ms. Note, these tasks can run in parallel, therefore mentioned CPU time decompression elapsed time could be shorter than the sum of individual tasks. Scene updating took 30ms of GPU time, from which the most time consuming is depth peeling with 24ms per update on average. Note that GPU tasks interleave with scene rendering, therefore real elapsed time is larger.

Although CPU and GPU times spent per updates are short, tasks are not executed continually but are spread over a large time range, 160ms for CPU tasks, and 130ms for GPU tasks on average. Graphs in Fig. 7.3 shows pipelines of processing scene updates. Real elapsed times are larger because of waiting for dependency, e.g., data from the server, GPU tasks for CPU tasks, and a time spent for the rendering of the current scene between GPU tasks.

6.3 Visual Quality

We tested visual quality with structural dissimilarity (DSSIM) and normalized root-mean-square error (NRSME) for all rendered frames from the benchmarks. During benchmarks, we collected history (timepoints with camera locations for client's frames and server's updates) that we used in the second pass to render reference images on the server, rerender images on the clients, and compare them using mentioned metrics. Path tracer's denoise depends on a sequence of rendered frames and provides better quality with higher framerate. Therefore reference images always provide better visual quality, as denoise is used for every rendered frame (ca. 60fps), not just updates (ca 5.5 per second).

Sponza Crytek scene has average DDSIM 0.067 for the static version and slightly higher 0.088 for the version with dynamic lights, Village scene has much higher average DSSIM: 0.118. Average NRMSE for static scenes: Sponza Crytek and Village was 0.044 and higher for dynamic Sponza Crytek: 0.085. Graphs in Fig. 7.5 shows the mentioned metrics for every frame, where we can also see a correlation with the scene latency. Fig. 7.7 shows comparison of images rendered on the client with reference images, and 7.8, 7.9 contains client's renders of all three scenes.

Visually results are acceptable if we consider that the update rate was only about 5.5 per second and high scene latency (ca. 350ms). The main visual problems are related to the packing method. Geometry perpendicular or facing await from the viewpoint used on the server, have no or insufficient shading in the texture (Fig. 7.6 top and middle). PVS construction can fail to predict and insert needed geometry (Fig. 7.6 bottom). We can also see reduced quality caused by lossy texture compression (video, JPEG) and reprojection, but it is not very noticeable during motion. A low update rate causes a negative effect in dynamic scenes (Sponza Crytek), as the change of lighting is noticeable.

Chapter 7
Conclusions

We have proposed and implemented a novel method for remote rendering using a thin client with low performance. The server renders the scene using path tracer with the denoise filter to multiple layers, packs them to one texture, and with PVS and additional data streams to the client. Both server and client use asynchronous parallel processing and communication to improve performance. The client can provide acceptable visual results even for very small scene update rate (ca. 5.5 per second with bandwidth around 1.5MBps for the client with Full HD). Our contribution is also in expanding SVGF denoise filter [SSK+17] to support multiple layers.

There are several issues — high transfer data requirements per the scene update (ca. 300kB), and slow performance of denoise filter (ca. 70ms per update). Our solution currently does not correctly handle perpendicular or facing away triangles from update viewpoint, moving camera backward, and fast camera rotation. This issue creates visual artifacts.

7.1 Possible improvements

We can render the scene on the server from multiple viewpoints to add hidden triangles (perpendicular or facing away), or to provide larger FOV. It will require to stream multiple packed textures. The slowest part on the server is the denoise filter. It can be optimized by providing a better data layout. By limiting the number of layers, we can reduce transfer size, speed-up denoise filter, but the client needs to implement some post-processing method to guess missing shading from surrounding fragments, possible in screen-space.

Another idea is to demodulate static texture from lighting (similarly as denoise filter works) and stream only lighting, and textures will be applied directly on the client. This approach requires the client to have a preloaded scene, including textures. With this approach, the server could stream lighting with lower resolution and still provide sufficient quality as lighting often varies smoothly.

Statistics										
Scene		Sponza Crytek			Sponza Crytek (static)			Village		
Frame Count		542			538			1640		
Scene Update Count		52			55			155		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
FPS		-	**57.1**	-	-	**56.5**	-	-	**59.6**	-
Scene Update Rate		-	**5.24**	-	-	**5.52**	-	-	**5.52**	-
DSSIM		**0.035**	**0.088**	**0.176**	**0.027**	**0.067**	**0.157**	**0.078**	**0.118**	**0.209**
NRMSE		**0.036**	**0.085**	**0.191**	**0.014**	**0.043**	**0.116**	**0.026**	**0.044**	**0.104**
Scene Latency [ms]		**181**	**383**	**926**	**153**	**340**	**763**	**186**	**343**	**599**
Frames / Update		0.0	10.8	20.0	1.0	10.3	18.0	0.0	10.7	19.0
Vertex Count		13098	105543	204375	13098	104055	199452	9927	31221	63531
Layout Count		6.0	10.5	31.0	6.0	10.5	31.0	9927	12.2	18.0
Fragments / Pixel		1.24	1.94	2.74	1.24	1.93	2.71	1.47	1.81	2.21
Texture	Width	1920	1920	1920	1920	1920	1920	1920	1920	1920
	Height	1408	2317	3248	1408	2304	3224	1848	2301	2904

Client Render Times [ms]										
Scene		Sponza Crytek			Sponza Crytek (static)			Village		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
GPU	Render	2.4	4.8	8.4	2.5	4.8	10.0	2.0	3.5	6.5
	Tasks	0.0	3.7	33.1	0.0	3.9	33.3	0.0	4.3	80.2
	Sum	**2.7**	**8.7**	**39.0**	**2.8**	**8.8**	**39.0**	**2.3**	**7.9**	**84.2**

Client Scene Update Times [ms]										
Scene		Sponza Crytek			Sponza Crytek (static)			Village		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Delta Time		2.0	190.8	494.2	19.0	181.2	499.3	3.2	181.3	496.8
Processing Time		156.6	252.3	544.8	149.3	242.6	563.8	181.2	241.3	470.4
Latency		181.3	290.3	690.1	153.2	263.4	636.2	186.2	260.7	497.0
CPU Parallel Decode	PVS	0.2	3.3	11.4	0.4	3.0	10.3	0.1	0.8	3.5
	Video	17.0	29.5	50.6	17.3	29.1	48.6	20.3	30.9	62.4
	JPEG	3.7	16.9	30.5	4.5	16.8	31.9	9.9	17.5	26.5
	Pixel Mask	4.4	15.0	25.4	4.2	14.8	27.2	9.7	17.2	26.4
	Blocks	0.1	0.4	1.3	0.1	0.4	0.8	0.1	0.5	1.1
	Triangle - Layer Mask	0.1	1.7	5.8	0.1	1.7	6.1	0.3	1.2	3.2
	CPU Time	**48.5**	**66.1**	**134.3**	**36.3**	**63.9**	**122.9**	**47.1**	**65.7**	**116.4**
	Elapsed	**86.6**	**156.6**	**318.5**	**89.9**	**147.7**	**254.1**	**116.0**	**177.5**	**262.6**
GPU	Depth Peeling	12.2	23.3	62.5	12.9	23.0	62.8	18.1	26.7	42.2
	Fragment Relocation	2.2	4.3	6.6	2.6	4.5	6.6	3.1	4.6	8.5
	GPU Time	**15.8**	**29.7**	**70.6**	**18.8**	**29.6**	**71.7**	**24.9**	**33.4**	**48.0**
	Elapsed	**70.1**	**139.1**	**342.0**	**84.0**	**125.6**	**348.5**	**73.9**	**126.7**	**366.4**
Receive		**66.0**	**172.7**	**399.9**	**84.8**	**158.9**	**300.4**	**97.5**	**167.7**	**263.3**

Table 7.1: Client benchmark statistics.

Server Times [ms]										
Scene		Sponza Crytek			Sponza Crytek (static)			Village		
		Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Delta Time		129.2	189.4	337.9	131.6	181.1	285.2	120.3	180.8	258.2
Processing Time		**90.6**	**135.3**	**286.0**	**81.7**	**134.7**	**284.4**	**99.8**	**134.9**	**189.8**
PVS	Raster + Mark	0.4	2.7	3.0	0.4	2.7	3.1	0.3	2.2	2.5
	Filter + Transform	0.3	0.6	0.9	0.2	0.5	0.9	0.2	0.4	0.7
	Map	0.4	0.5	0.5	0.4	0.5	0.5	0.5	0.5	0.5
	Sum	**1.7**	**3.7**	**4.4**	**1.5**	**3.4**	**4.4**	**1.4**	**3.4**	**3.9**
Pack	Sort, Relocate, Count	0.4	1.2	2.4	0.4	1.2	2.5	1.0	1.5	2.5
	Pack Tiles	0.6	1.2	1.6	0.5	1.2	1.6	0.8	1.3	1.5
	Fill	0.2	0.3	0.5	0.2	0.3	0.5	0.3	0.3	0.4
	Map	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3	0.3
	Sum	**1.3**	**2.6**	**4.1**	**1.2**	**2.5**	**4.2**	**2.1**	**3.0**	**4.1**
Copy GPU -> RAM		3.8	5.0	12.2	3.8	4.8	10.7	4.3	4.9	12.3
JPEG		0.3	0.9	2.9	0.3	0.6	2.9	0.5	0.7	2.9
Path Tracer	Trace	7.5	10.7	13.3	6.9	10.6	13.2	5.8	7.6	10.0
	Shade	10.3	14.3	18.7	9.9	14.5	18.1	8.2	10.0	12.1
	Filter	23.5	69.0	207.1	23.5	68.1	201.2	46.6	74.1	116.6
	Sum	**43.5**	**96.4**	**240.8**	**42.5**	**95.5**	**234.8**	**63.2**	**93.8**	**139.2**
CPU Parallel Compress	**PVS**	**4.2**	**24.9**	**46.4**	**3.5**	**23.9**	**47.6**	**3.6**	**10.5**	**18.6**
	Video	**22.9**	**27.4**	**34.3**	**21.9**	**27.7**	**32.0**	**23.4**	**28.4**	**33.7**
	Pixel Mask	**14.2**	**23.1**	**31.0**	**14.8**	**24.7**	**35.1**	**21.7**	**27.6**	**37.8**
	Blocks	0.1	0.2	0.4	0.1	0.3	0.4	0.2	0.3	0.4
	Triangle-Layer Mask	0.1	1.2	4.7	0.1	1.2	4.7	0.2	0.7	1.4

Scene Compressed Size [kB]									
Scene	Sponza Crytek			Sponza Crytek (static)			Village		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Size / Second	-	**1549**	-	-	**1544**	-	-	**1719**	-
Size / Scene Update	**97.3**	**295.5**	**546.4**	**79.9**	**279.9**	**527.4**	**217.9**	**311.6**	550.5
PVS	8.6	80.2	253.9	7.7	75.5	251.4	5.5	31.3	99.7
Video	23.4	57.2	239.9	2.9	51.1	215.8	29.8	63.5	218.2
JPEG	30.5	98.6	190.9	28.4	93.8	181.9	71.8	119.8	207.6
Pixel Mask	11.0	47.5	84.8	11.0	47.6	86.9	54.7	86.4	145.2
Blocks	1.9	4.6	7.0	1.9	4.7	7.0	4.1	5.4	7.3
Triangle - Layer Mask	0.4	7.0	23.1	0.4	7.0	22.5	1.4	5.0	10.3

Server Ray Counts [M]									
Scene	Sponza Crytek			Sponza Crytek (static)			Village		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Primary	2.58	4.02	5.68	2.58	3.99	5.62	3.04	3.75	4.58
Secondary	2.11	3.27	4.36	2.11	3.26	4.47	1.79	2.11	2.56
Deep	0.58	0.90	1.39	0.57	0.90	1.44	0.15	0.22	0.31
Shadow	10.11	14.23	21.35	9.75	14.02	17.39	4.93	9.59	15.23
Sum	**15.37**	**22.42**	**32.46**	**15.01**	**22.17**	**28.45**	**10.59**	**15.67**	**22.53**

Table 7.2: Server benchmark statistics.

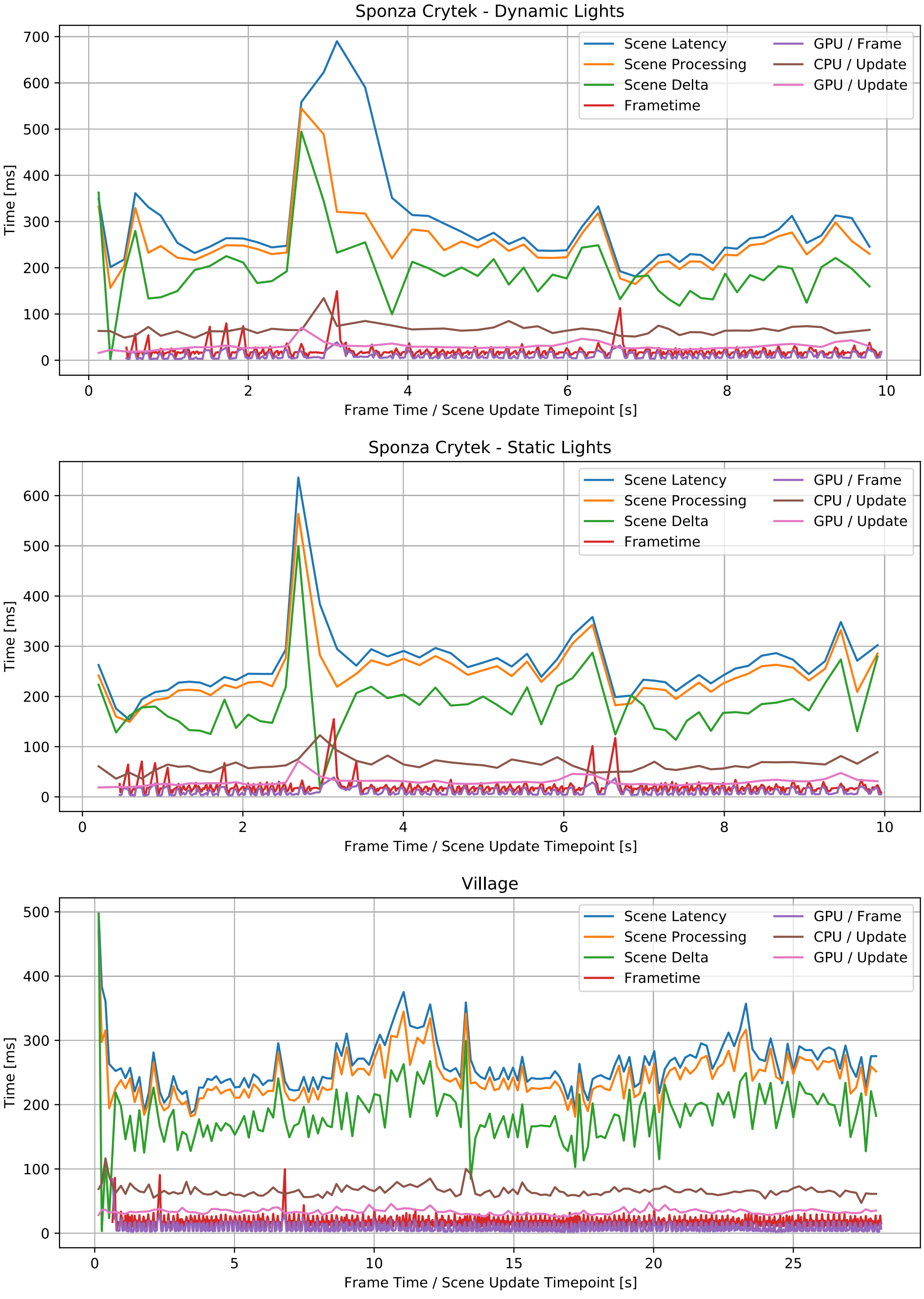


Figure 7.1: Client rendering and scene updating. CPU / GPU / Update: accumulated CPU / GPU time of updating (decompression, depth peeling and fragment relocation); GPU / Frame: rendering and GPU tasks; Latency: elpased time between scene request and its first rendering on the client; Processing: elapsed time between start of scene processing and its first rendering on the client; Delta: time between updates.

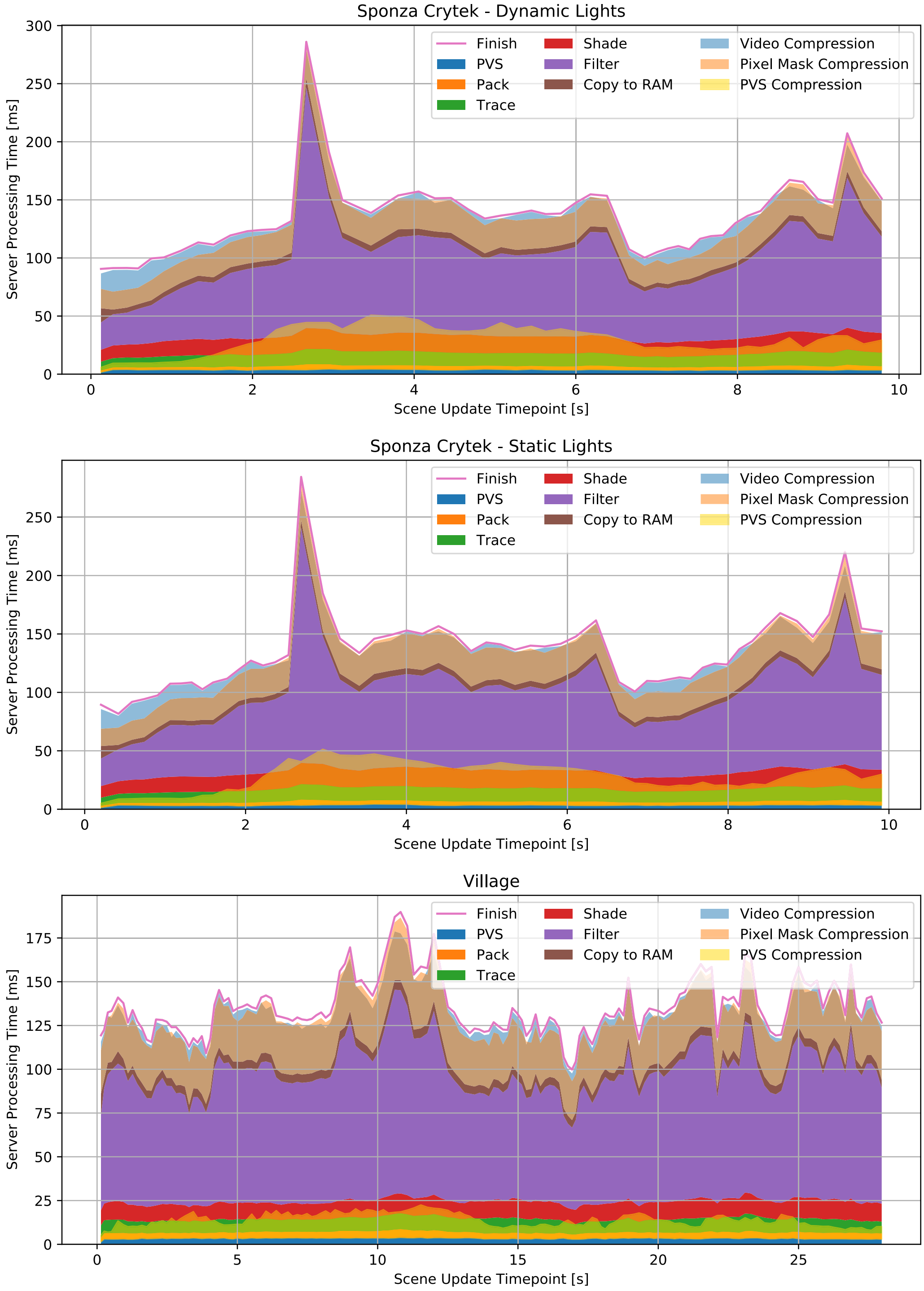


Figure 7.2: Server rendering and compression times. Vertical axis represents timeline per scene update at given timepoint at horizontal axis. PVS compression starts after PVS and runs in parallel with rest of steps. Similarly Video and Pixel Mask Compression runs in parallel starting after Copy to RAM step. JPEG and other compressions are not shown due to its small time contribution (<2ms).

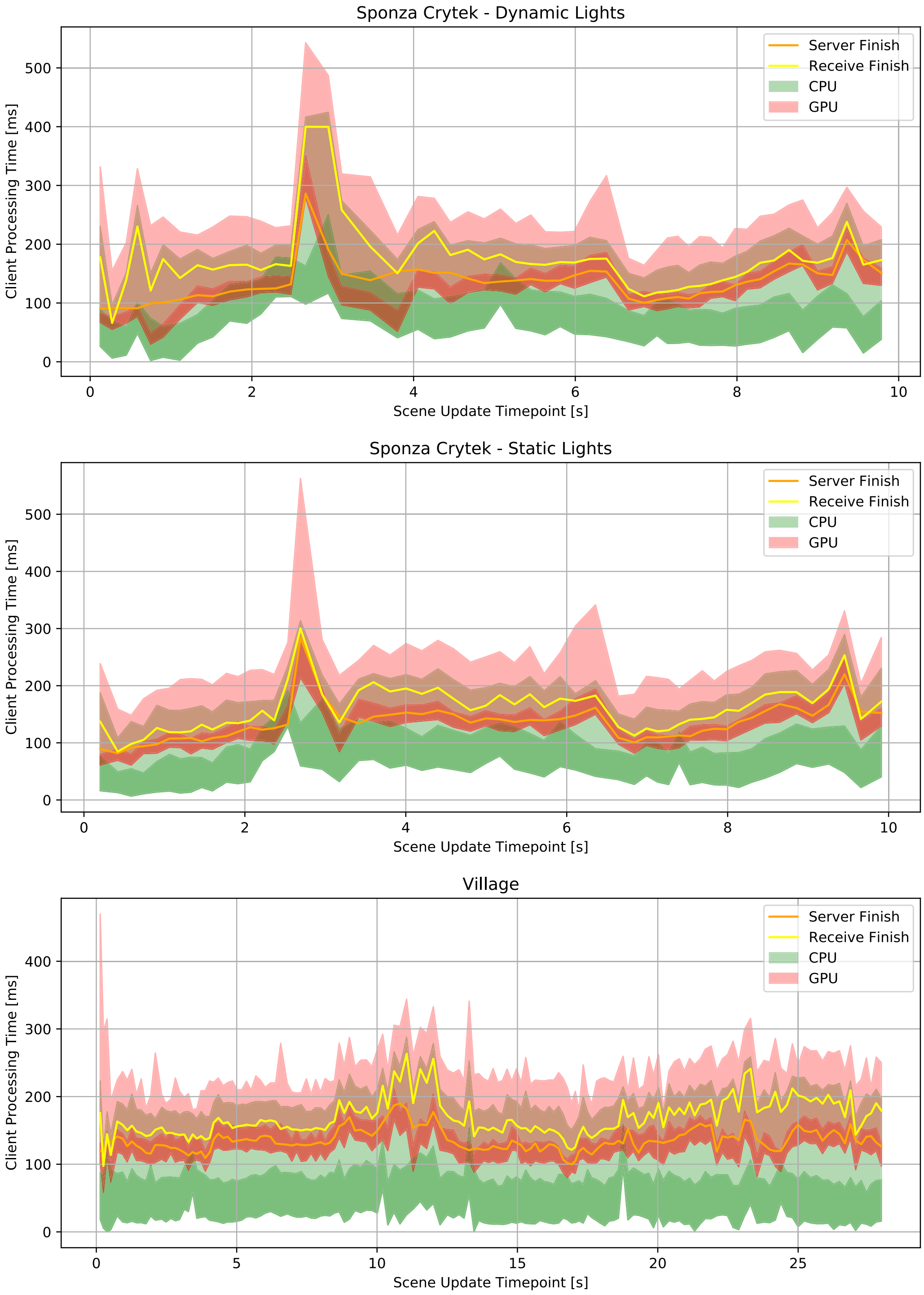


Figure 7.3: Client scene updating times. Vertical axis represents timeline per scene update at given timepoint at horizontal axis. Green / red areas shows time ranges of processing CPU / GPU tasks. Darker areas shows real time spent processing tasks (parallel on CPU serial on GPU) and brighter areas shows idle time or waiting for dependency, e.g., data from the server or GPU task for CPU task. Lighter GPU area also contains time spent on rendering of the current scene, as GPU tasks and scene rendering is interlaced.

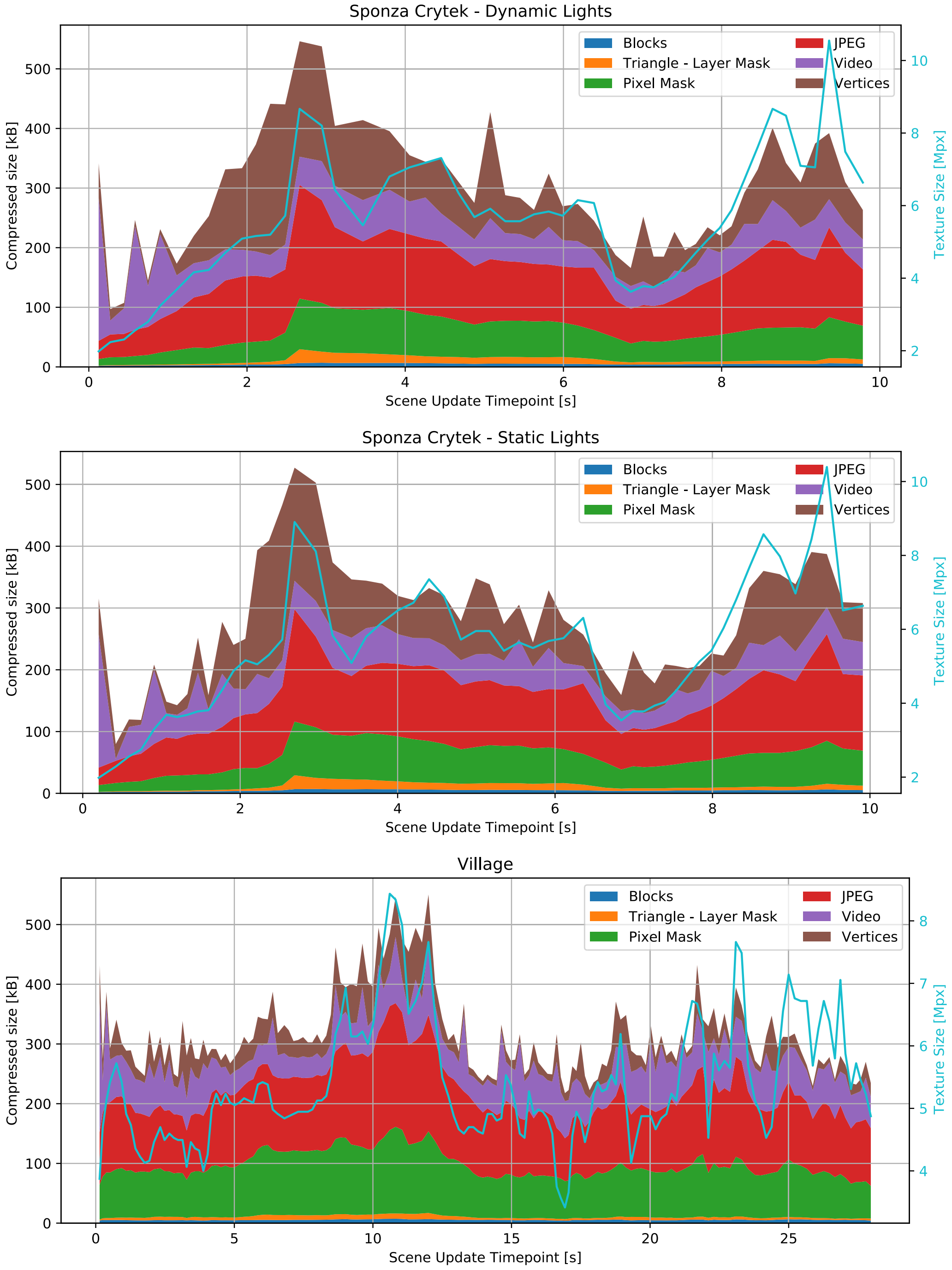


Figure 7.4: Scene compressed size (individual parts are in legend) with texture size (light blue line) to show correlation.

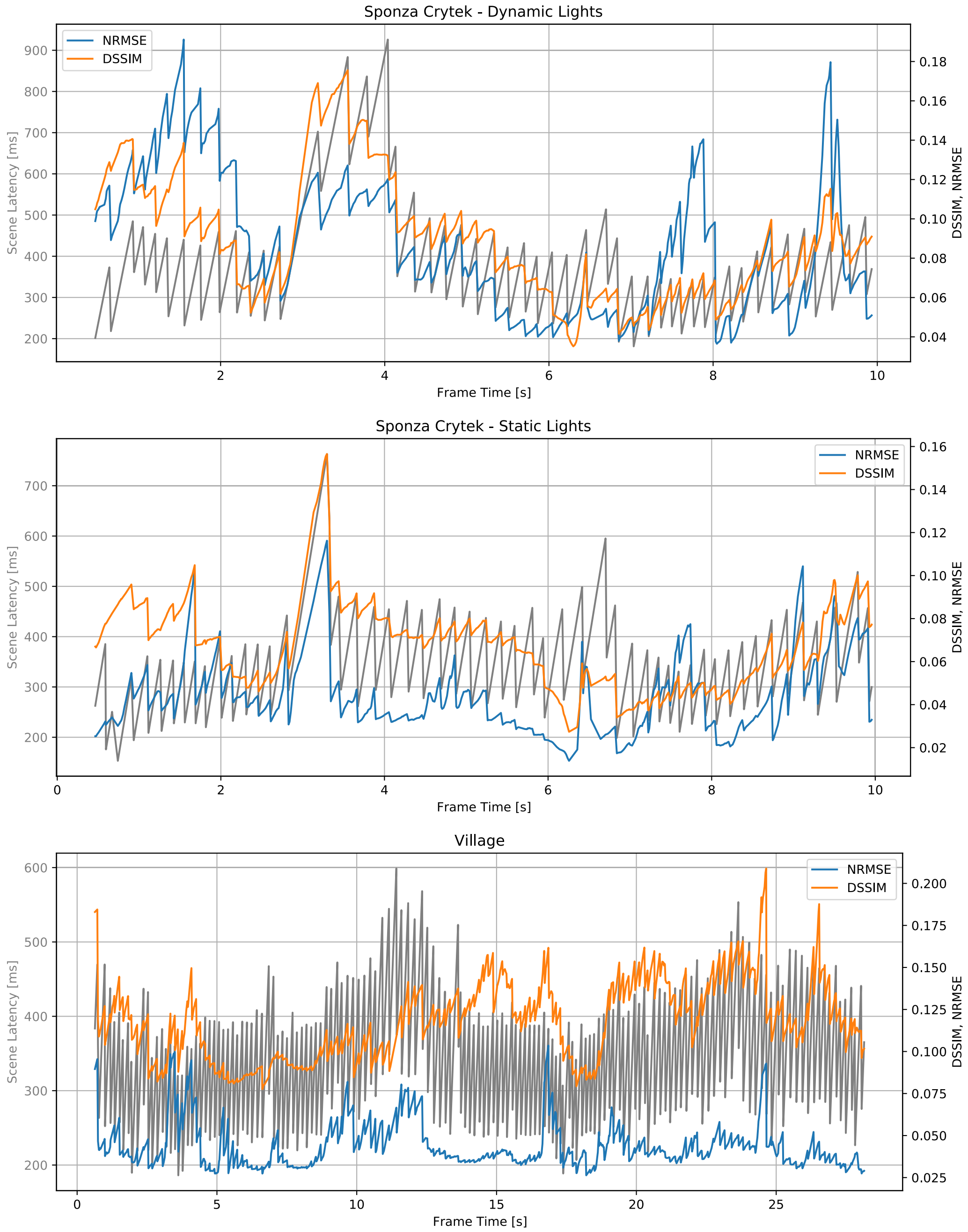


Figure 7.5: Image quality (DSSIM, NRMSE) for every frame of rendered sequence with scene latency (gray line) to show correlation.

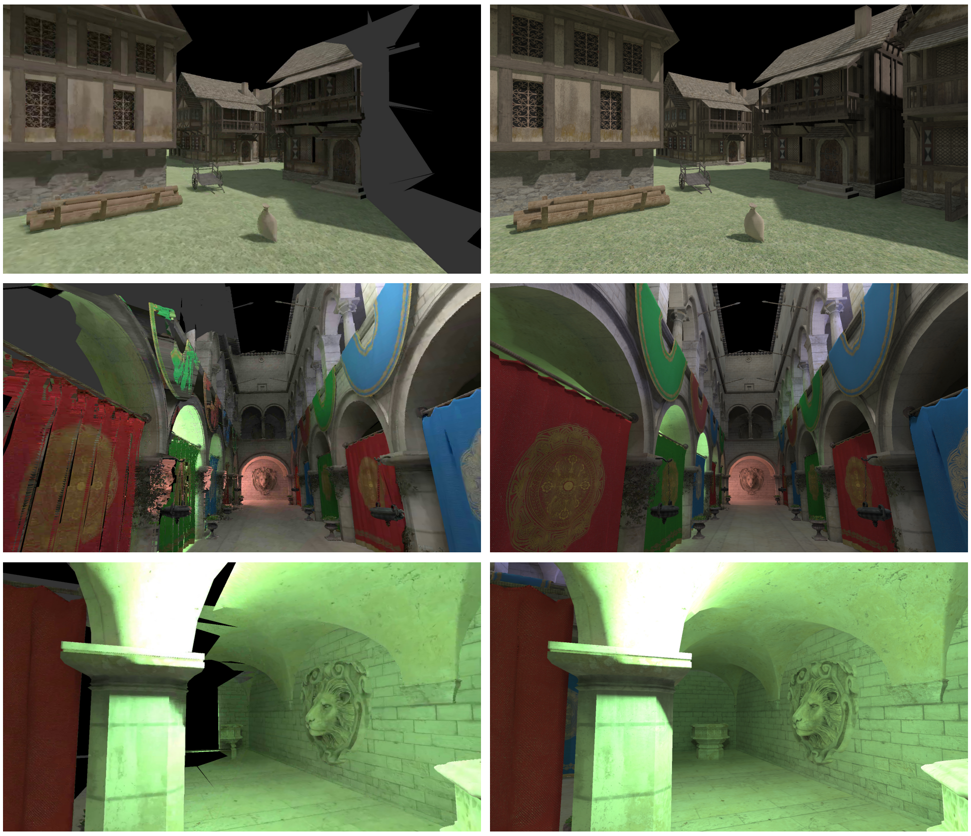


Figure 7.6: Client rendering errors, left images are reprojected on the client, and rights are references. Top: fast rotation causes missing geometry in PVS (black area on the right) or not having shading for geometry (gray parts of geometry), because it is outside of the rendered frustum . Middle: too perpendicular or facing away triangles doesn't have correct shading in the texture (left part, especially top green drapery), also current implementation doesn't support transparent geometry (leaves). Bottom: coming from the corner causes missing geometry as PVS creation failed to predict it.

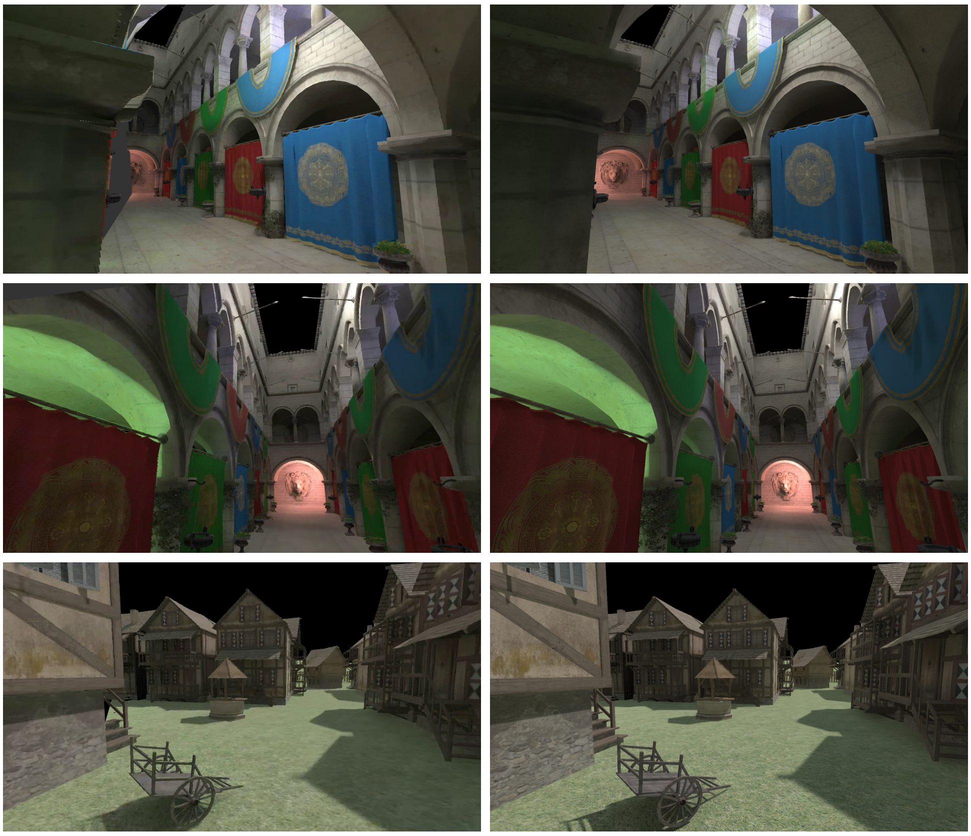


Figure 7.7: Comparisson of reprojected images on the client (left) and reference images (right) for Sponza Crytek with dynamic lights (top), Sponza Crytek with static lights (middle) and Village (bottom)

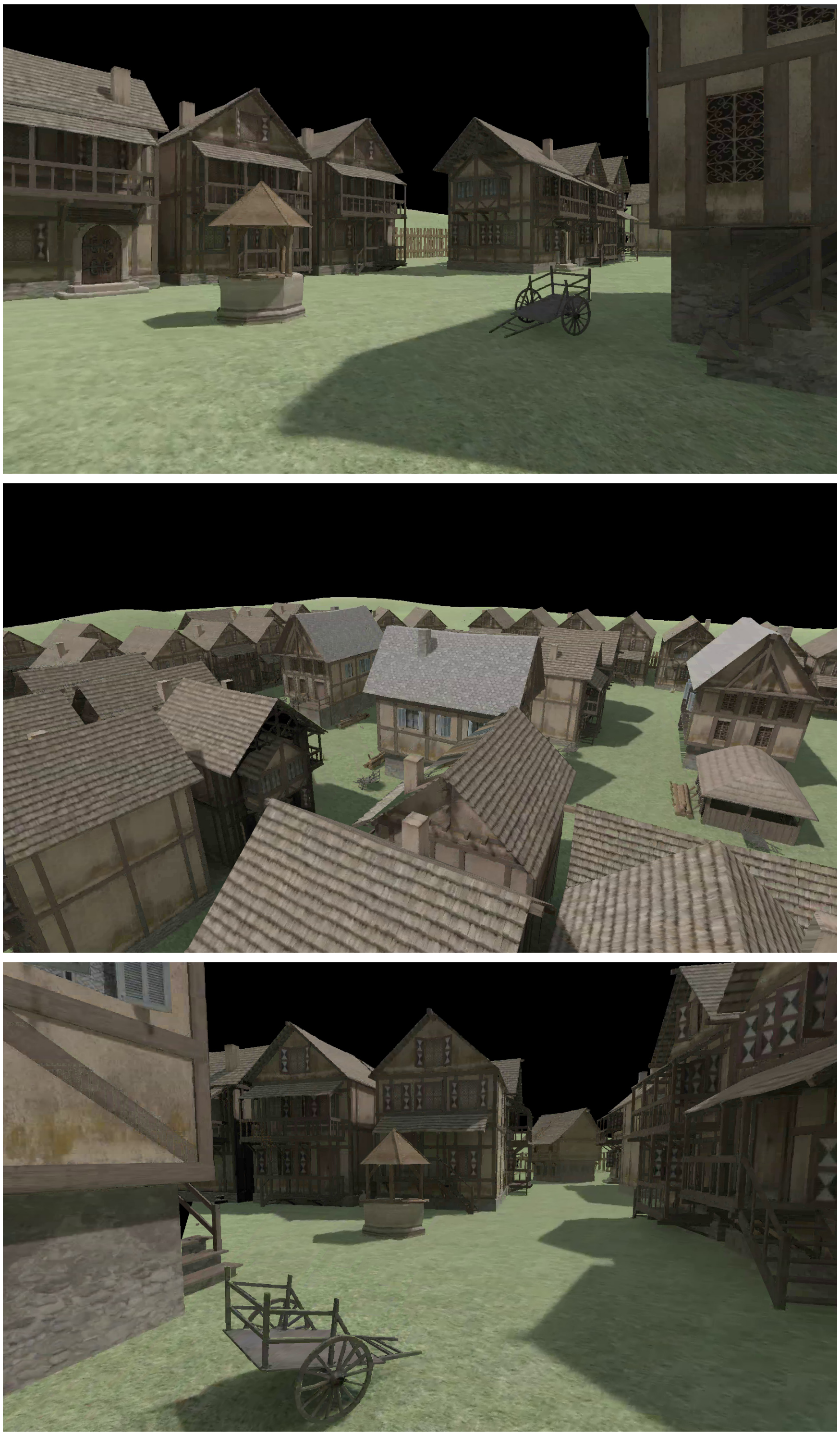


Figure 7.8: Village scene rendered on the client.

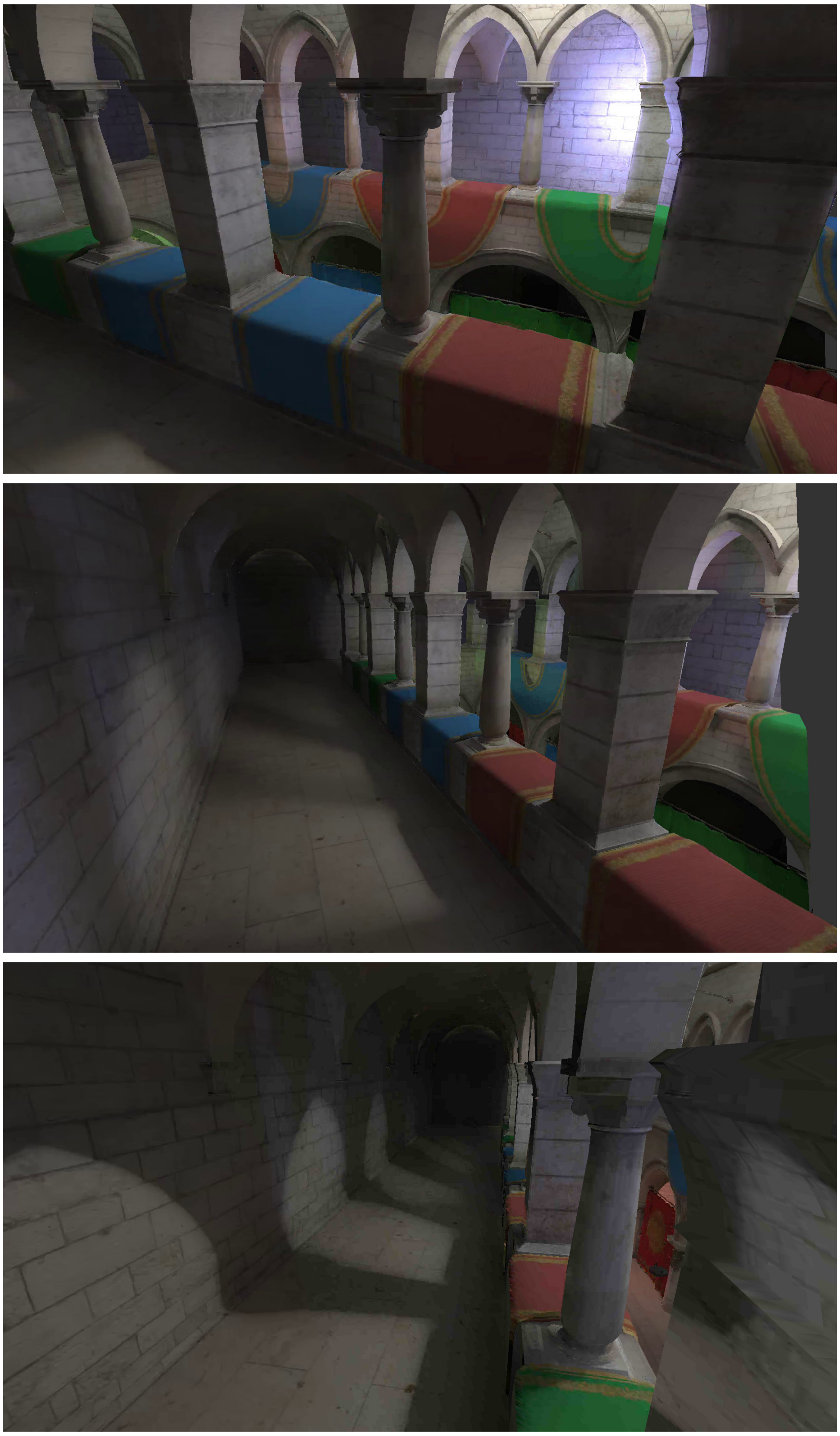


Figure 7.9: Sponza Crytek scene with dynamic lights (top and middle) and static lights (bottom) rendered on the client. Note: bottom has sharper shadows than middle, because denoise filter works better for static scenes.

Appendix A

Bibliography

[BMS+12] Huw Bowles, Kenny Mitchell, Robert W. Sumner, Jeremy Moore, and Markus Gross, *Iterative Image Warping*, Computer Graphics Forum (2012).

[Cas20] Everitt Cass, *Interactive order-independent transparency*, 2020, NVIDIA, Accessed: 2020-01-15.

[CW93] Shenchang Eric Chen and Lance Williams, *View interpolation for image synthesis*, Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '93, Association for Computing Machinery, 1993, p. 279–288.

[DER+10] 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** (2010), no. 2, 713–722.

[DRE+10] Piotr Didyk, Tobias Ritschel, Elmar Eisemann, Karol Myszkowski, and Hans-Peter Seidel, *Adaptive image-space stereo view synthesis*, Vision, Modeling and Visualization Workshop (Siegen, Germany), 2010, pp. 299–306.

[DSHL10] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch, *Edge-avoiding À-trous wavelet transform for fast global illumination filtering*, Proceedings of the Conference on High Performance Graphics (Goslar, DEU), HPG '10, Eurographics Association, 2010, p. 67–75.

[HSS19a] J. Hladky, H. P. Seidel, and M. Steinberger, *Tessellated shading streaming*, Computer Graphics Forum **38** (2019), no. 4, 171–182.

[HSS19b] 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** (2019), no. 6.

[Jen96] Henrik Wann Jensen, *Global illumination using photon maps*, Rendering Techniques '96 (Vienna) (Xavier Pueyo and Peter Schröder, eds.), Springer Vienna, 1996, pp. 21–30.

[Kaj86] James T. Kajiya, *The rendering equation*, Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '86, Association for Computing Machinery, 1986, p. 143–150.

[Kel97] Alexander Keller, *Instant radiosity*, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (USA), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., 1997, p. 49–56.

[LW93] Eric P. Lafortune and Yves D. Willems, *Bi-directional path tracing*, PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS '93, 1993, pp. 145–153.

[M.03] Laakso M., *Potentially visible set (pvs)*, 2003, Helsinki university of technology.

[MHAM08] Jacob Munkberg, Jon Hasselgren, and Tomas Akenine-Möller, *Non-uniform fractional tessellation*, Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, 2008, pp. 41–45 (English).

[MHH+12] Stephen McAuley, Stephen Hill, Naty Hoffman, Yoshiharu Gotanda, Brian Smits, Brent Burley, and Adam Martinez, *Practical physically-based shading in film and game production*, ACM SIGGRAPH 2012 Courses (New York, NY, USA), SIGGRAPH '12, Association for Computing Machinery, 2012.

[MMB97] William R. Mark, Leonard McMillan, and Gary Bishop, *Post-rendering 3d warping*, Proceedings of the 1997 Symposium on Interactive 3D Graphics (1997), 7–ff.

[MVD+18] Joerg H. Mueller, Philip Voglreiter, Mark Dokter, Thomas Neff, Mina Makar, Markus Steinberger, and Dieter Schmalstieg, *Shading atlas streaming*, ACM Trans. Graph. **37** (2018), no. 6.

[Nic65] Fred E. Nicodemus, *Directional reflectance and emissivity of an opaque surface*, Appl. Opt. **4** (1965), no. 7, 767–775.

[Ocu20] OculusVR, *Rendering to the oculus rift*, `https://developer.oculus.com/documentation/pcsdk/latest/concepts/dg-render/`, 2020, Accessed: 2020-01-15.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz, *The state of the art in interactive global illumination*, Comput. Graph. Forum **31** (2012), no. 1, 160–188.

[RKR+16] 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 (2016).

[Sam84] Hanan Samet, *The quadtree and related hierarchical data structures*, ACM Computing Surveys (1984), 2.

[SSK+17] Christoph Schied, Marco Salvi, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, and Aaron Lefohn, *Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination*, 07 2017, pp. 1–12.

[YHGT10] Jason C. Yang, Justin Hensley, Holger Grün, and Nicolas Thibieroz, *Real-time concurrent linked list construction on the gpu*, Computer Graphics Forum **29** (2010), no. 4, 1297–1304.

Appendix B

Directory structure of attachment files

`src` - source files and scripts
`data` - client's and server's data, scens and benchmark results
`imgs` - screenshots of tested scenes
`video` - videos of tested scenes