High-Performance Graphics (2023)
J. Bikker and C. Gribble (Editors)

PSAO: Point-Based Split Rendering for Ambient Occlusion

T. Neff† [1], B. Budge [2], Z. Dong [2], D. Schmalstieg [1] and M. Steinberger [1]

[1]Graz University of Technology, Austria
[2]Meta Reality Labs Research, USA

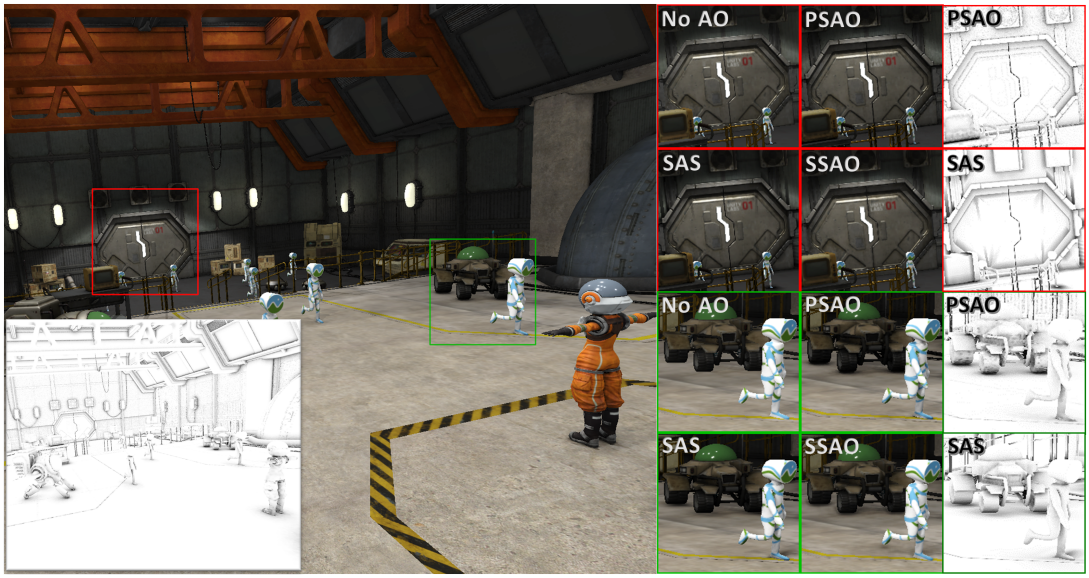

Figure 1: *Example frame from the* Robot Lab *scene rendered using point-based split ambient occlusion (PSAO). PSAO accurately reproduces contact shadows throughout the scene, while still handling long-range ambient occlusion (AO) comparable to a per-pixel ray traced solution. Shading atlas streaming (SAS) struggles with large geometry, blurring contact shadows across surfaces and exhibiting distortion artifacts depending on scene geometry. Screen-space ambient occlusion (SSAO) fails to represent the AO on thin, complex objects such as railings in the background, and produces unnatural halos around depth discontinuities such as around the wheels of the robot buggy.*

Abstract

Recent advances in graphics hardware have enabled ray tracing to produce high-quality ambient occlusion (AO) in real-time, which is not plagued by the artifacts typically found in real-time screen-space approaches. However, the high computational cost of ray tracing remains a significant hurdle for low-power devices like standalone VR headsets or smartphones. To address this challenge, inspired by point-based global illumination and texture-space split rendering, we propose point-based split ambient occlusion (PSAO), a novel split-rendering system that streams points sparsely from server to client. PSAO first evenly distributes points across the scene, and then subsequently only transmits points that changed more than a given threshold, using an efficient hash grid to blend neighboring points for the final compositing pass on the client. PSAO outperforms recent texture-space shading approaches in terms of quality and required network bit rate, while demonstrating performance similar to commonly used lower-quality screen-space approaches. Our point-based split rendering representation lends itself to highly compressible signals such as AO and is scalable towards quality or bandwidth requirements by adjusting the number of points in the scene.

CCS Concepts

• ***Computing methodologies*** → ***Rendering;***

1. Introduction

The recent introduction of hardware ray-tracing on desktop GPUs significantly changed the architecture of real-time graphics pipelines,

† The work was primarily done during an internship at Meta.

DOI: 10.2312/hpg.20231131

by combining traditional rasterization workloads with ray-traced effects for enhanced realism. However, these expensive effects are usually limited to high-end desktop GPUs. In contrast, the majority of real-time graphics applications have to run on low-power devices such as phones, which can very efficiently rasterize and shade simple materials, but cannot benefit from hardware ray-tracing due to power and performance limits. As a result, game streaming services such as *GeForce Now* have become a popular alternative to on-device rendering for low-power devices. These services typically operate by streaming a compressed video from the cloud to the client device, which necessitates a high-bandwidth network connection and requires the server location to be close enough to the client to ensure low network latency. These requirements prohibit the usage of such streaming services on high-latency mobile networks or in scenarios where extremely low latency is required, such as in XR.

To address these issues, recent work has introduced on split-rendering techniques [RLC*11; LD12; MVD*18; MNV*21; HSS19; HSS21; NMSS22] that decouple the rendering system by offloading expensive portions of the rendering equation to a powerful server, while performing simple geometry passes directly on a low-power client device. However, this line of research mostly focuses on computing the shading fully on the server, which can lead to artifacts with view-dependent portions of the rendering equation if server updates are delivered at a lower frame rate than used on the client. Furthermore, these approaches assign an oversimplified workload to the client device, usually consisting only of rasterization and texture mapping, while leaving the client's potential for performing low-complexity fragment shader computations underutilized.

In contrast to computing the full shading workload on the server, we take inspiration from recent hybrid graphics pipelines and advances in ray-tracing [Gau20] and focus on integrating high-quality AO on low-power devices. AO is a popular approximation of portions of global illumination, where surface points are darkened proportionally to the volume of objects in close proximity. Traditionally, a real-time graphics pipeline relies on screen-space ray-marching of the depth buffer [MML12; BS09; BSD08; JWPJ16] to approximate short-range AO. However, these screen-space approximations can produce artifacts as any occluded geometry behind the depth buffer cannot be considered during ray marching, leading to halos, wrongly occluded regions, and over-blurring depending on the reconstruction filter. Consequently, heuristics and carefully tuned reconstruction filters are needed to suppress the most severe artifacts. Since these tuning steps are cumbersome, the recent addition of hardware ray-tracing support led to increased popularity of ray-traced AO [Gau20], which can handle arbitrarily long distances between objects (given sufficient sampling). Depending on the scene, long-range AO can even be a substitute for standard shadow computation of direct shadows. Unfortunately, despite hardware ray-tracing slowly appearing on mobile phones and integrated desktop processors, even a single ray per visible pixel may still be too expensive, because high-quality AO requires hundreds of samples (albeit temporally amortized) to be noise-free.

Inspired by recent trends in split-rendering and ray-traced AO, we aim to close this gap by introducing PSAO. PSAO splits the graphics pipeline between a powerful server that computes high-quality ray-traced AO and a low-power client that autonomously computes simple shading, while optionally compositing the AO on top. We deviate from common split-rendering representations such as texels in UV space [HY16], individual triangles [HSS21; HSS19] or groups of triangles [MVD*18; NMSS22]. Instead, inspired by recent advances in global illumination [HBHB21], we use a sparse hash grid representation of points that stores the AO.

This architecture additionally decouples our split-rendering representation from object geometry, sidestepping distortion and resolution problems as well as the requirement for a unique surface parametrization that commonly plagues texture-space split-rendering approaches.

PSAO works by distributing points on object surfaces, for which AO is computed by ray tracing towards the scene geometry via a powerful server. This server efficiently compresses and encodes points, determines which points change more than a threshold, and only sends these points to the client. Both server and client store points within a hash table that is indexed by the 3D grid cell index of the voxel grid of each mesh instance. To reconstruct the AO, the client computes a simple weighted sum of neighboring grid cells for each fragment, using the hash table to look up the correct point cells. Due to the sparse storage, sparse network updates, and high degree of compression that can be applied to our AO points, PSAO outperforms MPEG-encoded texture-space split-rendering approaches in terms of network bandwidth. With PSAO, we make the following contributions:

- We introduce a novel split-rendering data structure for reconstructing ray-traced AO using points in a 3D hash grid.
- We demonstrate that our point-based representation outperforms common texture-space split-rendering representations in terms of quality, achieving better quality at up to 75% less network bandwidth.
- We show that the client rendering pass of PSAO has similar runtime performance (reconstructing at $\approx 0.7 - 1$ ms on a desktop GPU) as commonly used screen-space AO methods, at a much higher image quality due to being able to represent long-distance AO.
- Finally, we verify that PSAO can tolerate low server frame rates similarly to texture-space shading approaches, leading to further potential savings in network bandwidth and server load.

2. Related work

Real-time ambient occlusion AO [ZIK98] is a popular special effect that simulates the soft shadowing that occurs when ambient light is cast into a scene. These soft shadows usually appear around object creases or caves, or in areas where objects are close to each other, resulting in visually pleasing contact shadows and increased depth cues. Usually, AO is computed by casting rays across the hemisphere around a given surface normal to determine a percentage of rays that are occluded at a given distance [Lan04]. AO was pioneered by the movie production industry for offline rendering [Lan04], which did not need to adhere to real-time rendering requirements, and used ray-tracing to determine the AO.

The first real-time capable AO method was developed for the video game Crysis [Mit07]. It utilized the depth buffer in screen space to sample nearby rays by testing against the depth buffer,

blurring the final result for a smoother output. This approach, SSAO, is the foundation for modern real-time AO approaches to this day. In the years after, methods such as HBAO+ [BSD08] or GTAO [JWPJ16] build on the same principles of SSAO with further improvements in efficiency and quality. While all of these screen-space approaches are widespread and usable on lower-power devices such as phones, they often suffer from artifacts such as ghosting, blurring, or halos. These artifacts can be mitigated for short-range AO using heuristics and filters, but the information provided only by the screen-space depth and normal is fundamentally limited, prohibiting full access to neighboring scene geometry when computing AO in screen space.

As a result, recent work has also explored hybrid or alternative ways to compute AO in real time. VXAO [NVI16] voxelizes the scene in real time and performs cone tracing to compute more accurate occlusion values at the cost of performance. NNAO [HSK16], DeepShading [NAM*17], DeepAO [ZXL*20] and AO-Net [WZZ*23] reformulate the AO computation as a neural network inference problem, demonstrating competitive performance with modern handcrafted methods on some data sets, at the cost of much more expensive computation.

With the introduction of hardware ray tracing support in modern GPUs, it is now feasible to compute ray traced AO [Gau20] in real-time, given adequate reconstruction filters (such as temporal anti-aliasing). However, any ray-traced AO occupies a substantial amount of ray tracing budget per frame, ruling out the application on low-power devices, such as phones.

To summarize, the most popular ways for online AO recompute the AO at least partially for every pixel in every frame, either through ray tracing, via screen-space depth buffer tests or via neural network inference. Neither of these approaches offers a simple way to decouple the AO computation from the rest of the render pipeline, e.g., to compute them at a lower frame rate or stream them over a network.

Split rendering Split rendering denotes a generalization of decoupled shading [RLC*11; LD12], where portions of the graphics pipeline are executed on one device (usually a powerful server), encoded within an intermediate representation, sent to another device (usually a lightweight client), and finally decoded and reconstructed effectively splitting the rendering load between multiple devices. The core of any split rendering algorithm is its *intermediate representation*, i.e., the space and data structure used for storing intermediate rendering results, and the strategy by which server and client cooperate via this representation. These data structures range from on-surface caches [WTS*23; SWTS23] to compute direct illumination, spheres or grids to efficiently cache volumetric clouds [WLT*23] or participating media [SWT*23], or caches operating in texture space [BJ22; MVD*18; NMSS22; HY16].

Recently, research in split rendering has largely focused on *geometry-based* texture-space shading methods [MVD*18; HSS21; NMSS22; HSV*22] that are suitable for MPEG compression and streaming to a lower-power client device. Although these approaches improve network bandwidth efficiency, image quality, and rendering speed, they are fundamentally tied to the underlying geometric representation. Poor geometric quality (e.g., tiny and sliver-y triangles) or poor texture coordinates (e.g., strong distortions) can severely affect the quality that the intermediate representation can provide in texture space. Avoiding these conditions can be notoriously difficult if existing content must be used, since such content is often created without texture-based streaming in mind. Furthermore, texture-based streaming methods typically have elevated computational cost for dynamic memory management on the client, which has to divide resources between decoding and rendering. Hence, achieving reliable frame rate targets may become difficult.

A second group of split rendering methods focuses on dynamically managing, allocating, and shading tiles of texels in mipmapped textures [HY16; Bak16; BJ22]. These methods usually require complicated dynamic memory management as well, including virtual texturing, to handle non-unique surface parameterizations. Furthermore, significant effort has to be made to ensure over-shading due to mipmapping does not significantly increase the compute cost [NMSS22]. Finally, although these variants of texture-space shading methods can be suitable for large-scale shading amortization on the server, it is non-trivial to extend them to a split-rendering approach which has a low-power client as its target.

Decoupled ambient occlusion Recently, alternative representations for storing AO or global illumination have increased in popularity [HBHB21], mostly due to being temporally coherent, which allows efficient spatio-temporal accumulation and filtering [SKW*17] and tight integration with widespread temporal anti-aliasing solutions [YLS20]. These storage representations go all the way back to radiance caching [WRC88], where radiance samples can be stored in world space, and can later be used for reconstruction, e.g., via splatting [GKBP05]. Alternatively, another way to decouple AO is to store it as surface elements [Bun05] containing a position, radius, and normal, and accessing them during fragment shading via a hierarchical data structure or textures. Most relevant to our work, recent advances in ray tracing have inspired hybrid approaches that combine ray tracing with hash-grid based data structures to efficiently store and retrieve AO samples and accumulate them spatio-temporally [Gau20]. However, these methods usually assume powerful GPUs that can handle complex multi-level hash structures, incorporate expensive spatio-temporal filtering, and run on a single system, thus disregarding memory consumption of the internal point representation. With PSAO, we take inspiration from these approaches, simplifying the hash-grid based data structures to be highly compressible and fast to look up in the context of a split-rendering system.

3. Point-based Split Rendering for Ambient Occlusion

Traditional split rendering approaches, which represent shading of geometric primitives in texture space, typically require expensive preprocessing of the geometry to enable fast memory management [MVD*18] while keeping distortion in texture space to a minimum [NMSS22; MVD*18; HSS19; HSS21]. The preprocessing fundamentally modifies the geometric representation by enforcing a maximum triangle size and splitting triangles with highly non-uniform edge lengths. As a consequence, a bloated geometric representation can significantly influence the network transmission rate, making it necessary to re-tune all scene parameters when a different geometric resolution of source assets is desired.

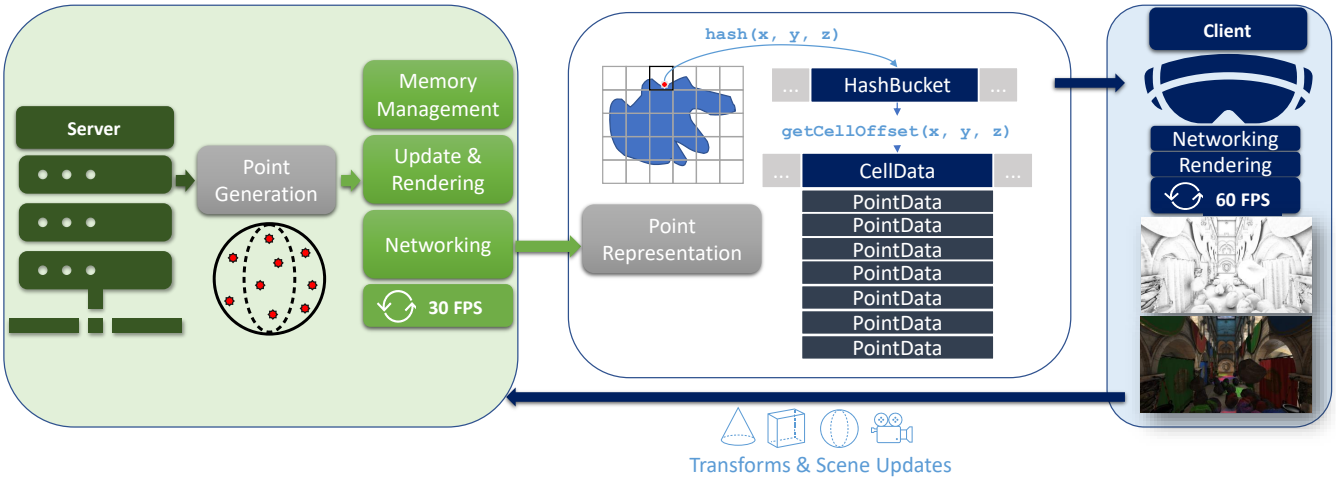

Figure 2: *The PSAO pipeline spans server and client by storing highly compressed points inside grid cells that are accessed via a hash table. The server (left, green) handles initial point generation, dynamically manages the hash grid memory, updates and renders AO points based on scene changes, and compresses points for networking if they changed above a threshold. The client receives updates to the hash and point structures incrementally. It renders the AO on top of a standard geometry pass at its native display rate by looking up the nearest AO points for each fragment, thus effectively decoupling the client from the server. The client sends scene updates and camera transformations to the server to inform the server of any changes that need consideration for the AO points.*

To circumvent the issues caused by tying the split-rendering representation to geometry, we introduce PSAO, an efficient split-rendering method that depends on points in space rather than geometric primitives. With PSAO, we generate points uniformly for each mesh instance, compute high-quality ray-traced ambient occlusion, and efficiently pack, compress, and send points from a powerful server to a lightweight client. The client only needs to render a simple basic color pass and can efficiently sample the ray-traced AO on top to improve the realism of the rendered image.

The full PSAO pipeline is shown in Figure 2: First, initial point positions are generated in a fast *point generation* pass. Second, in the *server update* pass, the server computes the new position, normal and AO for each currently visible point. For each point where changes exceed a threshold, the server modifies the internal hash table structures to efficiently update the changed points. For both server and client, the *networking* pass executes network (de-)compression and copies the incoming or outgoing data into the respective buffers. Finally, the *client render* pass, which is decoupled from the server, uses the updated data structures to efficiently look up points in a virtualized 3D grid and reconstruct the AO contribution per pixel.

3.1. Point generation

For our implementation of PSAO, we generate all initial point positions in a preprocessing pass. To do this, we compute the total surface area of each mesh instance, and spawn a fixed number of points (randomly distributed using a uniform distribution given in triangle barycentrics) per unit area on the surface of each mesh instance. Afterwards, to ensure the points are well-distributed over the whole mesh, we use Poisson disk rejection [Yuk15] to remove half of all the initially generated points. During point generation, we spawn points on both sides of triangles that are double-sided (using the respective normal), and we reject points that would fall on masked transparent surfaces.

3.2. Memory management

After spawning points, we perform an initial memory management pass that assigns all spawned points to a grid cell inside a virtual 3D grid of cells per mesh instance. PSAO generates updates affecting the memory management and networking stage on *grid cells* (`CellData`), where each `CellData` contains at most seven points in our default configuration. Our memory management structure is visualized in Figure 3. First, we determine the grid resolution based on the size of the axis-aligned bounding box of the given mesh instance and the total surface area. Second, we determine a `HashBucket` for each point by hashing the dense 3D grid index of each point. In our default configuration, we store 8 entries in each `HashBucket`, which can each refer to different `CellData`. To compute the hash function of a 3D point, we use a simple hash function of the following form

$$H(\mathbf{p}) = p_x + p_y * D_x * 17 + p_z * D_x * D_y * 31 \qquad (1)$$

where p_x, p_y, p_z are the x, y and z component of a given point, and D_x and D_y are the x and y dimensions of the virtual grid for the given mesh instance. For our hash table, we conservatively allocate memory based on $4\times$ the number of average cells derived from the Poisson disk radius from point generation, divided by the bucket size of 8, which is sufficient in practice to avoid overflows in `HashBucket` due to hash collisions. From the `HashBucket`, we allocate a new `CellData` if necessary, and then assign each point to its corresponding `CellData`. As a result, after initial allocation,

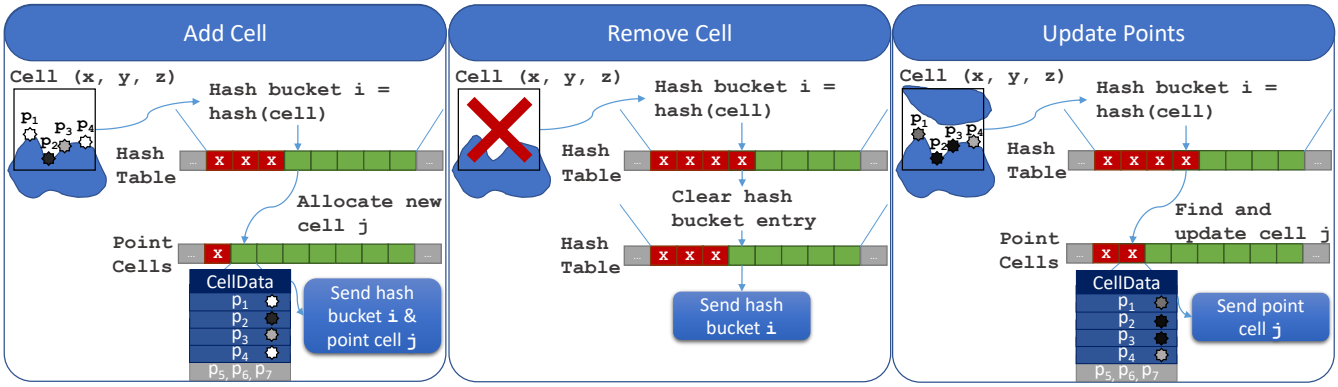

Figure 3: *The memory management of PSAO is done fully on the server-side and kept simple to ensure fast look-ups on the client and efficient network compression. When adding a new point cell, we first hash its integer coordinates and find a free slot in the corresponding hash bucket, before allocating a new point cell from a global point cell buffer. Afterwards, the newly added point cell and updated hash bucket can be compressed and sent to the client. When removing point cells, we clear the corresponding hash bucket entry and free the point cell entry on the server side. We only transmit the updated hash bucket to the client, as that is sufficient for the client to skip the deleted cell during rendering. Finally, if points within a cell are updated, added, or removed, we update them individually on the server side, and send the full point cell to the client for simplicity.*

each `CellData` can be looked up by hashing its 3D grid position to find the correct `HashBucket`, and iterating over the entries in the `HashBucket` to find the matching `CellData`. Note that in our default configuration, we discard points that would overflow entries in either `HashBucket` or `CellData`, which does not lead to noticable artifacts in practice as there are usually a sufficient number of points in neighboring cells. Each point in `CellData` is packed into 32 bits: one bit as an allocation flag, 15 bits for the position (quantized within the grid cell it resides in), eight bits for the oct-encoded normal vector, and eight bits for the AO value. In practice, this packing uses tolerances which are conservative enough to never observe differences in the resulting image quality; we can potentially cut the bit rate to 24 or even 16 bits for higher efficiency.

3.3. Server update

At the beginning of each server frame, we first determine coarse point visibility by skipping every point outside an extended field of view to ensure parity with related methods [MVD*18]. For each visible point, we compute updated point positions and normals (including any normal maps) based on the current frame's skinning information and rigid-body animations. We then compute the updated AO value by tracing rays from each point towards the scene geometry using cosine hemisphere sampling in the direction of the normal. We use hardware ray-tracing extensions for ray tracing and update acceleration structures every server frame. Given the updated AO, position, and normal, we apply a threshold to the difference vs the previous frame, and discard all points below the threshold. This ensures that network bandwidth and memory management performance is kept lean. For each point that requires an update, we use atomics to perform efficient memory management directly on the GPU. If a point moves to a new `CellData`, we allocate memory for it in our `HashBucket` and `CellData` buffers. Simultaneously, each `CellData` that contains changed points is marked as *dirty* to prepare it for network transmission in subsequent passes.

3.4. Networking

After updating all visible points, allocating new `CellData`, and marking changed `CellData` as dirty, the server simultaneously prepares the data that will be sent to the client as well as updates its own internal data structures. To this extent, the server first updates each changed `HashBucket` and copies it to a persistently mapped buffer that can subsequently be compressed on the CPU. Afterwards, each `CellData` within the changed `HashBucket` is also updated with the newly changed positions, values and normals of each point, and similarly copied to a CPU-mapped buffer. For both the `HashBucket` updates and the `CellData` updates, we perform a fast compaction step on the server that reorders them such that empty entries are at the end of each list, which is later used on the client side for faster skipping of empty entries. After preparing both the `HashBucket` and `CellData` updates, the server compresses them on the CPU using off-the-shelf fast compression algorithms, such as *zstd* [Met23] or *lz4* [Col23]. After the client receives the data from the server, it updates its internal representation of the `HashBucket` and `CellData` structures, which it can subsequently use to render frames decoupled from the server.

3.5. Client rendering

The client is decoupled from the server, and autonomously renders a simple color pass using the scene geometry. It sends its current camera pose and all scene updates to the server, and receives updated AO information every time the server updates. The main render pass of the client is a standard forward or deferred rendering pass. After the main scene color is determined (using basic materials and lighting), the client samples the AO information received from the server. To reconstruct the final AO value, the client considers the nearest $2 \times 2 \times 2$ cell volumes around the shaded world-space fragment position. The client hashes each dense grid index in this $2 \times 2 \times 2$ area to look up all 8 `HashBucket` structs. For each `HashBucket`, we

Table 1: *Image quality results of PSAO vs. SAS and SSAO. All methods run at* 60 Hz *on both server and client. For SAS, we denote the per-patch downsampling bias factor B as SAS-B. For PSAO, we denote the number of points per unit area N as PSAO-N. Across the board, PSAO outperforms both SAS and SSAO in terms of quality at a much lower network bandwidth compared to SAS. For* Robot Lab*, the large number of skinned animated characters lead to a higher base network bandwidth compared to the other scenes, but PSAO is still able to compress the AO points much more effectively than the texture-space packing of SAS, leading to a much higher quality even at lower point counts. SSAO cannot compete with the ray traced long-distance AO of PSAO and SAS, only achieving better quality than SAS on the almost fully static* Space *scene.*

	Robot Lab				*Space*				*Sponza*			
Renderer	LPIPS ↓	PSNR ↑	FLIP ↓	Mbit/s	LPIPS ↓	PSNR ↑	FLIP ↓	Mbit/s	LPIPS ↓	PSNR ↑	FLIP ↓	Mbit/s
SAS-32	.0087	31.86	.0380	44.05	.0090	25.70	.0814	17.00	.0053	32.21	.0444	26.68
SAS-8	.0057	33.61	.0341	50.62	.0082	25.98	.0778	26.29	.0037	33.51	**.0410**	43.73
PSAO-512	.0047	34.07	.0313	**25.69**	.0068	26.80	.0666	**4.66**	.0060	31.42	.0500	**7.21**
PSAO-1024	.0034	35.20	.0296	53.81	.0056	27.72	.0614	9.30	.0046	32.55	.0468	14.03
PSAO-1536	**.0030**	**35.61**	**.0289**	83.86	.0050	28.16	.0591	13.81	.0042	32.79	.0456	20.77
PSAO-2048	.0031	35.47	.0291	113.31	**.0048**	**28.40**	**.0578**	18.22	**.0035**	**33.61**	.0444	27.53
SSAO	.0081	31.42	.0497	-	.0072	26.96	.0876	-	.0141	26.90	.0956	-

perform a simple linear search to find the matching `CellData` corresponding to each of the 8 neighboring cells. For each `CellData`, every allocated point is included in a weighted sum to reconstruct the final AO value. The weight α_i is given as:

$$\alpha_i = \max\left(0, \frac{r - \|\mathbf{p_i} - \mathbf{x}\|}{r}\right), \tag{2}$$

where r is a configurable interpolation radius that we set to the average radius computed during the Poisson disk rejection when generating the initial points; $\mathbf{p_i}$ is the AO point with index i stored in `CellData`, and $\mathbf{x}$ is the current fragment position. Note that all positions and vectors are first transformed into the local space of their corresponding mesh instance, which reduces point updates for meshes undergoing simple rigid body transforms. These transformations are done using the last received server transformation data to ensure validity of the sampled AO data relative to the given shaded surface fragment when server and client are not running in lockstep. This way, PSAO does not exhibit screen-space ghosting, and increased network latency merely leads to AO values being applied with a delay on object surfaces, thus also decoupling AO between server and client. We refer to our supplementary video for a showcase of this behavior under different server frame rates. To reduce the impact of AO contributions leaking through surfaces, we additionally apply a heuristic based on the angles of the fragment normal and AO point vectors: We set the weight

$$\alpha_i = 0, \;\; \text{if } (\mathbf{n_{p_i}} \cdot \mathbf{n_x}) \leq T_n \text{ or } \left(\mathbf{n_{p_i}} \cdot \frac{\mathbf{p_i} - \mathbf{x}}{\|\mathbf{p_i} - \mathbf{x}\|}\right)^2 \geq T_d, \tag{3}$$

where $\mathbf{n_{p_i}}$ is the normal of the AO point $\mathbf{p_i}$; $\mathbf{n_x}$ is the normal of the current fragment $\mathbf{x}$; T_n is a threshold for the normal vector criterion, and T_d is a threshold for the distance vector criterion. We experimentally evaluated $T_n = 0.1$ and $T_d = 0.7$ to catch common cases of artifacts due to sampling unsuitable neighboring points, and use these throughout our evaluation. These heuristics enforce similarity to the current fragment normal; AO points that lie directly behind the current fragment are not considered for reconstruction.

The final reconstructed AO value is then given as

$$AO(\mathbf{x}) = \begin{cases} \frac{1}{\sum_{i=0}^{N} \alpha_i} \sum_{i=0}^{N} (\alpha_i \cdot AO(\mathbf{p_i})), & \text{if } \sum_{i=0}^{N} \alpha_i \neq 0 \\ 1, & \text{otherwise,} \end{cases} \tag{4}$$

where N is the total number of points in the surrounding $2 \times 2 \times 2$ volume, and $AO(\mathbf{p_i})$ is the ray-traced AO value that was computed in the last server frame. In practice, given that we keep at most 7 points in each individual cell, we reconstruct the final result by blending at most 56 points within a $2 \times 2 \times 2$ hash cell area surrounding the current fragment. Note that the reconstruction involves only simple instructions, and accessing all points of a given cell is efficient as consecutive points within `CellData` fit into the same cache line.

4. Evaluation

We implemented PSAO inside NVidia's Falcor real-time rendering framework [KCK*22] and compare it against two baselines: SAS [MVD*18], which is a modern real-time texture-space streaming approach, and Falcor's implementation of SSAO. For all baselines, we set the AO distance to be 0.6 meters. For PSAO, the main configuration parameters are the number of points per unit area (which we vary throughout the evaluation) as well as the server frame rate. For network compression of PSAO on the server, we use the *zstd* [Met23] compression library with a compression level of 15 and a compression strategy of 4.

Shading atlas streaming We configure SAS to limit the atlas size and the sizes of patches in the atlas in order to reduce the required network bandwidth to a similar level as PSAO. We only store the AO value inside the atlas encoded as 24-bit RGB. For the measurement of network bit rate, we only measure the required network bandwidth for transmitting the atlas and ignore auxiliary information, such as patch updates that are sent via an auxiliary TCP stream. Note that SAS can benefit from highly optimized MPEG compression and is thus able to potentially shade more texels compared to the straightforward compression utilized by PSAO. To keep the comparison between PSAO and SAS as fair as possible, we do not preprocess the

scene geometry for SAS. This results in slightly suboptimal geometry for SAS, but experiments on a subset of our test scenes do not suggest massive improvements for SAS with better scene geometry. For SAS, we downscale patches by a factors $f \in [8, 16, 32]$, keeping a minimum resolution of 2×2 texels for each patch in the atlas. Furthermore, we vary the server frame rate to evaluate its decoupled rendering performance compared to PSAO. We configure the video encoding of SAS to use a standard lossy real-time codec.

Screen-space ambient occlusion For SSAO, we use the default settings provided in Falcor, and do not vary the server frame rate, as decoupling the screen-space AO buffer from the client frame rate would not be possible without major ghosting artifacts.

4.1. Evaluation setup

Test scenes We test on three scenes that use physically-based materials and have been expanded with dynamic lights and shadows. *Sponza* contains additional moving spotlights and physically animated boulders, which test the limits of update rates for both PSAO and SAS. *Space* contains freely available assets from the Unity asset store, and contains some moving asteroids as well as high-frequency specular materials and animated lights. However, the animated objects in *Space* are positioned relatively far from other objects, making it a mostly static test case for rendering AO. Finally, *Robot Lab* is a Unity sample scene that contains mostly diffuse materials and no animated lights, and we extended it by adding 23 skinned character models from Khronos' glTF sample repository to evaluate the effectiveness of all approaches on skinned/deformable models. We evaluate two camera paths on each scene, each containing 900 frames at 60 Hz for a total duration of 15 s each.

Comparison setup All AO approaches are evaluated by compositing the reconstructed AO on top of a $4\times$ supersampled forward renderer to ensure that any differences in measured quality are solely caused by differences in AO. All methods are compared against a per-pixel ray-traced ambient occlusion reference (RTAO) that is rendered with 512 samples per pixel. To determine the resulting image quality, we measure PSNR, LPIPS [ZIE*18] and FLIP [ANA*20] compared to the reference result. Furthermore, for PSAO and SAS, we measure the network bandwidth required to update the split-rendering representation. All methods are evaluated at a resolution of 1920×1080, running on a workstation containing an NVidia RTX 4090 GPU with 24 GB of memory and an Intel i9-13900K CPU with 64 GB memory.

4.2. Image quality

We present averaged quantitative results in Table 1. Overall, across all test scenes, PSAO achieves the highest quality when measured in LPIPS, PSNR and FLIP, outperforming SAS and SSAO at only 512 points per unit area in *Robot Lab* and *Space*, and outperforming SAS in *Sponza* when using 2048 points per unit area. Especially *Robot Lab* and *Space* contain many large triangles, which can lead to distortion in the texture-space representation of SAS. SSAO is not competitive in terms of quality in any of our test scenes with the chosen AO distance. This is likely because SSAO cannot accurately produce long-distance AO from screen space information alone, leading to massive halo and blur artifacts. Only in the almost fully static *Space* scene can SSAO outperform SAS, but still falls well behind PSAO in even the most sparse configuration. We also compare PSAO against SAS at lower server frame rates of $[5, 15, 30]$ to evaluate the effectiveness of PSAO in even lower bandwidth scenarios, as well as how it performs against SAS with higher latencies between server and client frames. These results are shown in Figure 5. Averaged across all scenes, PSAO adapts to low server frame rates just as well as SAS, which was designed for handling low server frame rates and high latency scenarios. We also present a selection of rendered frames in Figure 4, comparing PSAO against SAS, SSAO and the ground truth ray-traced AO. Across all scenes, PSAO best resembles the ray-traced per-pixel ambient occlusion (RTAO) reference. In contrast, SSAO struggles to resolve the long-distance AO correctly, leading to overall brighter outputs and strong halo artifacts, such as around the boulders in *Sponza* or around the skinned characters in *Robot Lab*. SAS achieves quality on par with PSAO on meshes with finely subdivided geometry, such as the flower vase in *Sponza*, the more detailed machinery in *Robot Lab*, or the space ship in *Space*. However, for coarsely subdivided geometry, such as the floors in all scenes, or sliver-y non-uniform triangles such as the tubes in *Robot Lab*, SAS distorts its AO representation in texture-space as it can't perfectly match the required resolution while staying within the shading budget. These distortions manifest as either blurred patches of occluded or visible regions (such as on the floor in *Sponza*, near the flower pot) or patches that cannot correctly consider normal maps due to lack of texture-space resolution, such as on the tubes in *Robot Lab*. To rule out insufficient subdivision as the cause of quantitative differences for SAS, we also evaluated *Sponza* after subdividing the scene such that triangles are small enough to not run into under-sampling issues. When preprocessed like this, SAS-8 and SAS-32 achieve an average PSNR of 34.00 and 32.69 in *Sponza*, respectively, with a negligible impact on average network bandwidth. This shows that geometric preprocessing can help slightly to improve image quality when storing AO in texture space, although not significantly enough to warrant increased complexity and preprocessing times.

4.3. Network bandwidth

Comparing the results in Table 1 demonstrates that the split AO representation of PSAO can be compressed highly efficiently. In the almost fully static *Space* scene, PSAO outperforms SAS in terms of quality at a data rate of only 4.66 MBit/s. In the highly dynamic *Sponza*, SAS closes the gap in quality, but is still outperformed in terms of required network bandwidth by PSAO—PSAO requires 27.53 MBit/s vs. SAS at 43.73 MBit/s at the highest quality, and SAS at the lowest quality is outperformed by PSAO-1024 at 14.03 MBit/s. Finally, in *Robot Lab*, which contains many animated skinned characters, we again see that the cheapest configuration of PSAO achieves better quality than SAS at half the network bandwidth. Due to the large number of animated meshes, the network bandwidth of PSAO is higher than in the other scenes, but PSAO still achieves much better quality than SAS at equal or lower bandwidth.

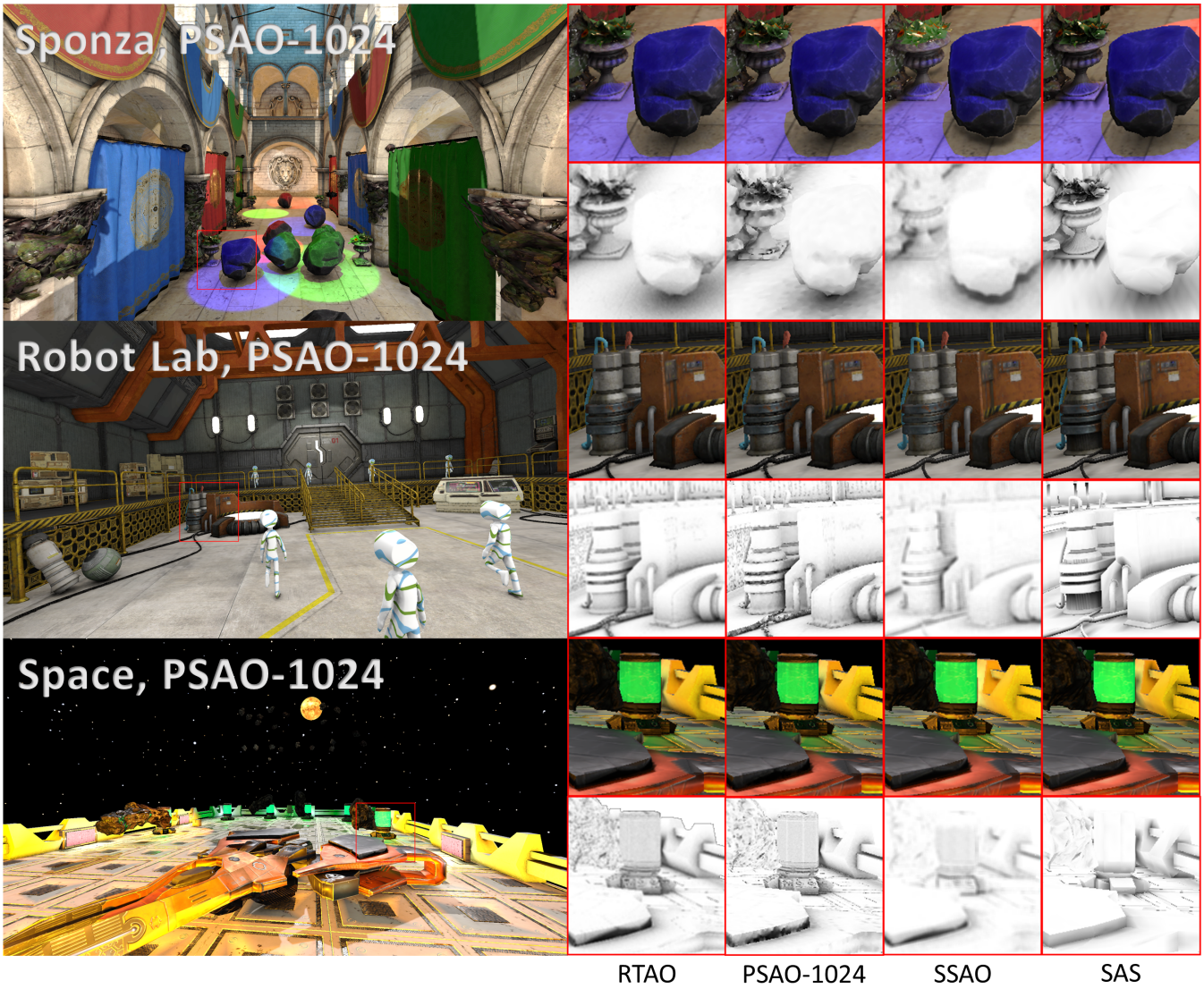

Figure 4: *Qualitative comparison of rendered frames between PSAO using 1024 points per unit area, ray-traced per-pixel ambient occlusion (RTAO), SSAO and SAS.*

4.4. Client rendering speed

For PSAO, reconstructing the final AO value by performing hash lookups and computing a weighted sum of all points in the neighborhood takes $\approx 0.4 - 0.5$ ms. This does not significantly change with larger point counts—PSAO samples a consistent number of points per frame, keeping rendering performance consistent. For SAS, reconstructing the final AO value is essentially free—it is just a bilinear texture lookup. However, SAS additionally requires decoding of MPEG frames and geometric structures, which can substantially increase load on lower-power client devices. Finally, SSAO renders in ≈ 0.3 ms including a blur filter. However, if a depth and normal buffer are not available already, the additional depth and normal prepass required by SSAO can lead to substantially increased render times of up to 1 ms, depending on the scene. Overall, PSAO is competitive with SSAO in terms of real-time rendering performance on the client side, presenting an attractive solution if powerful servers are available for a split AO setup.

5. Discussion and conclusion

With PSAO, we showed that combining sparse point-based representations with modern ray tracing pipelines can provide an attractive solution for split rendering and streaming of AO. Inspired by recent work that stores global illumination surfels, we distribute points evenly across surfaces and compute AO only for these points, rather than per pixel. By storing points in virtualized grid cells that are accessed via a hash table, we can easily access neighboring points of any given fragment during run time. Furthermore, we showed that AO points can be compressed into 32 bits per point by quantizing

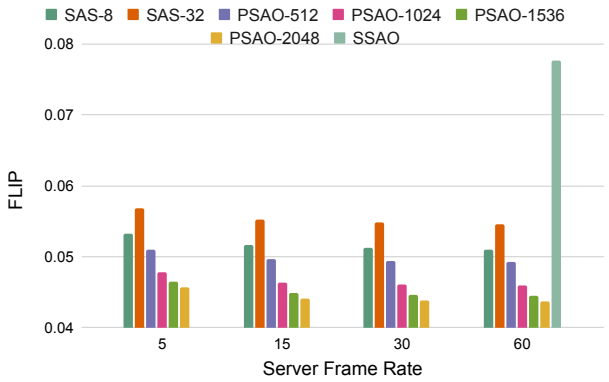

Figure 5: *At varying server frame rates, PSAO outperforms SAS when averaged across all scenes. For SAS, we denote the per-patch downsampling bias factor B as SAS-B. For PSAO, we denote the number of points per unit area N as PSAO-N. Even at only 5 frames per second, PSAO can keep up with SAS, which was tuned for low server frame rate streaming of texture-space shading.*

point positions, normals and AO values. As a result, PSAO achieves rendering speed comparable to screen-space AO approaches, while achieving better quality than texture-space shading approaches. At equal image quality, PSAO achieves up to $2-4\times$ lower network bit rates compared to the MPEG compressed shading of SAS.

It can be noted that while our focus was on split rendering, PSAO is not limited to split-rendering setups. Figure 5 demonstrates that ray-traced ambient occlusion can be rendered at an effective frame rate of 15 or even 5 frames per second while retaining high quality, and so PSAO could be beneficial even for non-split renderers by computing the ray tracing updates across meshes in a round-robin fashion at a reduced frame rate, and using PSAO for AO.

The main limitation of our prototype PSAO implementation is the static point generation—we currently never add additional points to the scene and only spawn them once during point generation. However, our memory management would allow adding points dynamically, which can be combined with visibility culling techniques to spawn and delete points and whole cells dynamically as certain mesh instances or clusters of points become visible.

Second, PSAO uses a very simple heuristic to determine whether or not any given point needs to be re-transmitted over the network—the server needs to ray trace visible points to compute this heuristic. This is sub-optimal for server performance, but could be addressed in the future by checking mesh bounding boxes and rejecting points directly if their distance to moving meshes is larger than the AO distance that is used for ray tracing. It can be noted that this would also help server scalability.

Finally, PSAO currently does not implement countermeasures against leaking of AO during reconstruction. If a fragment samples a point that would be occluded by a surface between point and fragment, the resulting AO will be inaccurate. In practice, a sufficiently dense point distribution keeps these artifacts at a minimum. However, if additional performance budget is available on the client, PSAO could be extended to store highly compressed coarse depth buffers along oct-encoded directions [HBHB21], which can then be used to perform coarse depth tests during point sampling.

In the future, PSAO also could be extended to other view-independent quantities like diffuse GI, although it would be best suited for quantities that can be quantized efficiently and sampled sparsely on object surfaces. Furthermore, PSAO sidesteps many issues that plague traditional texture-space split rendering pipelines by decoupling the representation from geometry, removing the need for geometry preprocessing and enabling gradual level-of-detail by simply scaling the number of points. To summarize, we believe that PSAO demonstrates that point-based split rendering can be a useful alternative for low-frequency quantities like AO, especially when compared to approaches based on MPEG compressed textures.

References

[ANA*20] ANDERSSON, PONTUS, NILSSON, JIM, AKENINE-MÖLLER, TOMAS, et al. "FLIP: A Difference Evaluator for Alternating Images". *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 15:1–15:23 7.

[Bak16] BAKER, DAN. *Object Space Lighting*. Game Developers Conference. 2016. URL: http://www.cogsci.rpi.edu/~destem/gamearch/gdc16/Object-Space-Lighting-Rev-21.pptx 3.

[BJ22] BAKER, DANIEL and JARZYNSKI, MARK. "Generalized Decoupled and Object Space Shading System". *Eurographics Symposium on Rendering*. Ed. by GHOSH, ABHIJEET and WEI, LI-YI. The Eurographics Association, 2022. ISBN: 978-3-03868-187-8. DOI: 10.2312/sr.20221163 3.

[BS09] BAVOIL, LOUIS and SAINZ, MIGUEL. "Multi-Layer Dual-Resolution Screen-Space Ambient Occlusion". *SIGGRAPH 2009: Talks*. SIGGRAPH '09. New Orleans, Louisiana: Association for Computing Machinery, 2009. ISBN: 9781605588346. DOI: 10.1145/1597990.1598035. URL: https://doi.org/10.1145/1597990.1598035 2.

[BSD08] BAVOIL, LOUIS, SAINZ, MIGUEL, and DIMITROV, ROUSLAN. "Image-Space Horizon-Based Ambient Occlusion". *ACM SIGGRAPH 2008 Talks*. SIGGRAPH '08. Los Angeles, California: Association for Computing Machinery, 2008. ISBN: 9781605583433. DOI: 10.1145/1401032.1401061. URL: https://doi.org/10.1145/1401032.1401061 2, 3.

[Bun05] BUNNELL, MICHAEL. "Chapter 14 Dynamic Ambient Occlusion and Indirect Lighting". Vol. 2. Jan. 2005, 223–233 3.

[Col23] COLLET, YANN. *LZ4 - Extremely fast compression*. https://github.com/lz4/lz4. 2023 5.

[Gau20] GAUTRON, PASCAL. "Real-Time Ray-Traced Ambient Occlusion of Complex Scenes Using Spatial Hashing". *ACM SIGGRAPH 2020 Talks*. SIGGRAPH '20. Virtual Event, USA: Association for Computing Machinery, 2020. ISBN: 9781450379717. DOI: 10.1145/3388767.3407375. URL: https://doi.org/10.1145/3388767.3407375 2, 3.

[GKBP05] GAUTRON, PASCAL, KRIVÁNEK, JAROSLAV, BOUATOUCH, KADI, and PATTANAIK, SUMANTA. "Radiance Cache Splatting: A GPU-Friendly Global Illumination Algorithm". *Eurographics Symposium on Rendering (2005)*. Ed. by BALA, KAVITA and DUTRE, PHILIP. The Eurographics Association, 2005. ISBN: 3-905673-23-1. DOI: 10.2312/EGWR/EGSR05/055-064 3.

[HBHB21] HALEN, HENRIK, BRINCK, ANDREAS, HAYWARD, KYLE, and BEI, XIANGSHUN. "Global Illumination Based on Surfels". *ACM SIGGRAPH 2021 Courses*. 2021 2, 3, 9.

[HSK16] HOLDEN, DANIEL, SAITO, JUN, and KOMURA, TAKU. "Neural Network Ambient Occlusion". *SIGGRAPH ASIA 2016 Technical Briefs*. SA '16. Macau: Association for Computing Machinery, 2016. ISBN: 9781450345415. DOI: 10.1145/3005358.3005387. URL: https://doi.org/10.1145/3005358.3005387 3.

[HSS19] HLADKY, JOZEF, SEIDEL, HANS-PETER, and STEINBERGER, MARKUS. "Tessellated Shading Streaming". *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering 2019)* (2019). ISSN: 1467-8659. DOI: 10.1111/cgf.13780 2, 3.

[HSS21] HLADKY, J., SEIDEL, H.P., and STEINBERGER, M. "SnakeBinning: Efficient Temporally Coherent Triangle Packing for Shading Streaming". *Computer Graphics Forum* 40.2 (2021), 475–488. DOI: https://doi.org/10.1111/cgf.142648. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.142648. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.142648 2, 3.

[HSV*22] HLADKY, JOZEF, STENGEL, MICHAEL, VINING, NICHOLAS, et al. "QuadStream: A Quad-Based Scene Streaming Architecture for Novel Viewpoint Reconstruction". *ACM Trans. Graph.* 41.6 (Nov. 2022). ISSN: 0730-0301. DOI: 10.1145/3550454.3555524. URL: https://doi.org/10.1145/3550454.3555524 3.

[HY16] HILLESLAND, KARL E and YANG, JC. "Texel Shading". *Proceedings of the 37th Annual Conference of the European Association for Computer Graphics: Short Papers*. EG '16. Goslar Germany, Germany: Eurographics Association, 2016, 73–76. DOI: 10.2312/egsh.20161018 2, 3.

[JWPJ16] JIMENEZ, JORGE, WU, XIAN-CHUN, PESCE, ANGELO, and JARABO, ADRIAN. "Practical Real-Time Strategies for Accurate Indirect Occlusion". *SIGGRAPH 2016 Courses*. 2016 2, 3.

[KCK*22] KALLWEIT, SIMON, CLARBERG, PETRIK, KOLB, CRAIG, et al. *The Falcor Rendering Framework*. https://github.com/NVIDIAGameWorks/Falcor. Aug. 2022. URL: https://github.com/NVIDIAGameWorks/Falcor 6.

[Lan04] LANDIS, HAYDEN. "Production-Ready Global Illumination". 2004 2.

[LD12] LIKTOR, GÁBOR and DACHSBACHER, CARSTEN. "Decoupled Deferred Shading for Hardware Rasterization". *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '12. Costa Mesa, California: Association for Computing Machinery, 2012, 143–150. ISBN: 9781450311946. DOI: 10.1145/2159616.2159640. URL: https://doi.org/10.1145/2159616.2159640 2, 3.

[Met23] META PLATFORMS, INC. *Zstandard*. https://github.com/facebook/zstd. 2023 5, 6.

[Mit07] MITTRING, MARTIN. "Finding next Gen: CryEngine 2". *ACM SIGGRAPH 2007 Courses*. SIGGRAPH '07. San Diego, California: Association for Computing Machinery, 2007, 97–121. ISBN: 9781450318235. DOI: 10.1145/1281500.1281671. URL: https://doi.org/10.1145/1281500.1281671 3.

[MML12] MCGUIRE, MORGAN, MARA, MICHAEL, and LUEBKE, DAVID. "Scalable Ambient Obscurance". *Proceedings of ACM SIGGRAPH / Eurographics High-Performance Graphics 2012 (HPG '12)* (June 2012). High-Performance Graphics 2012. URL: https://casual-effects.com/research/McGuire2012SAO/index.html 2.

[MNV*21] MUELLER, JOERG H., NEFF, THOMAS, VOGLREITER, PHILIP, et al. "Temporally Adaptive Shading Reuse for Real-Time Rendering and Virtual Reality". *ACM Trans. Graph.* 40.2 (Apr. 2021). ISSN: 0730-0301. DOI: 10.1145/3446790. URL: https://doi.org/10.1145/3446790 2.

[MVD*18] MUELLER, JOERG H, VOGLREITER, PHILIP, DOKTER, MARK, et al. "Shading Atlas Streaming". *ACM Transactions on Graphics* 37.6 (Nov. 2018). DOI: 10.1145/3272127.3275087 2, 3, 5, 6.

[NAM*17] NALBACH, O., ARABADZHIYSKA, E., MEHTA, D., et al. "Deep Shading: Convolutional Neural Networks for Screen Space Shading". *Computer Graphics Forum* 36.4 (2017), 65–78. DOI: https://doi.org/10.1111/cgf.13225. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13225. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13225 3.

[NMSS22] NEFF, T., MUELLER, J. H., STEINBERGER, M., and SCHMALSTIEG, D. "Meshlets and How to Shade Them: A Study on Texture-Space Shading". *Computer Graphics Forum* 41.2 (2022), 277–287. DOI: https://doi.org/10.1111/cgf.14474. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14474. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14474 2, 3.

[NVI16] NVIDIA. "VXAO: Voxel Ambient Occlusion". 2016 3.

[RLC*11] RAGAN-KELLEY, JONATHAN, LEHTINEN, JAAKKO, CHEN, JIAWEN, et al. "Decoupled Sampling for Graphics Pipelines". *ACM Trans. Graph.* 30.3 (May 2011). ISSN: 0730-0301. DOI: 10.1145/1966394.1966396. URL: https://doi.org/10.1145/1966394.1966396 2, 3.

[SKW*17] SCHIED, CHRISTOPH, KAPLANYAN, ANTON, WYMAN, CHRIS, et al. "Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination". *Proceedings of High Performance Graphics*. HPG '17. Los Angeles, California: Association for Computing Machinery, 2017. ISBN: 9781450351010. DOI: 10.1145/3105762.3105770. URL: https://doi.org/10.1145/3105762.3105770 3.

[SWT*23] STOJANOVIC, ROBERT, WEINRAUCH, ALEXANDER, TATZGERN, WOLFGANG, et al. "Efficient Rendering of Participating Media for Multiple Viewpoints". *Computer Graphics Forum* 42.8 (2023). DOI: 10.1111/cgf.14874. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14874 3.

[SWTS23] STADLBAUER, PASCAL, WEINRAUCH, ALEXANDER, TATZGERN, WOLFGANG, and STEINBERGER, MARKUS. "Surface Light Cones: Sharing Direct Illumination for Efficient Multi-viewer Rendering". *Computer Graphics Forum* 42.8 (2023). DOI: 10.1111/cgf.14875. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14875 3.

[WLT*23] WEINRAUCH, ALEXANDER, LORBEK, STEPHAN, TATZGERN, WOLFGANG, et al. "Clouds in the Cloud: Efficient Cloud-Based Rendering of Real-Time Volumetric Clouds". *Computer Graphics Forum* 42.8 (2023). DOI: 10.1111/cgf.14876. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14876 3.

[WRC88] WARD, GREGORY J., RUBINSTEIN, FRANCIS M., and CLEAR, ROBERT D. "A Ray Tracing Solution for Diffuse Interreflection". *Proceedings of the 15th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, 85–92. ISBN: 0897912756. DOI: 10.1145/54852.378490. URL: https://doi.org/10.1145/54852.378490 3.

[WTS*23] WEINRAUCH, ALEXANDER, TATZGERN, WOLFGANG, STADLBAUER, PASCAL, et al. "Effect-based Multi-viewer Caching for Cloud-native Rendering". *ACM Trans. Graph.* 42.4 (Jan. 2023) 3.

[WZZ*23] WANG, JIAYI, ZHOU, FAN, ZHOU, XIANG, et al. "AO-Net: Efficient Neural Network for Ambient Occlusion". *Graphics Interface 2023*. 2023. URL: https://openreview.net/forum?id=b-3cQ_h4kqz 3.

[YLS20] YANG, LEI, LIU, SHIQIU, and SALVI, MARCO. "A Survey of Temporal Antialiasing Techniques". *Computer Graphics Forum* 39.2 (2020), 607–621. DOI: https://doi.org/10.1111/cgf.14018. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14018. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.14018 3.

[Yuk15] YUKSEL, CEM. "Sample Elimination for Generating Poisson Disk Sample Sets". *Computer Graphics Forum (Proceedings of EUROGRAPHICS 2015)* 34.2 (2015), 25–32. ISSN: 0167-7055. DOI: 10.1111/cgf.12538. URL: http://dx.doi.org/10.1111/cgf.12538 4.

[ZIE*18] ZHANG, RICHARD, ISOLA, PHILLIP, EFROS, ALEXEI A, et al. "The Unreasonable Effectiveness of Deep Features as a Perceptual Metric". *CVPR*. 2018 7.

[ZIK98] ZHUKOV, SERGEY, IONES, ANDREI, and KRONIN, GRIGORIJ. "An Ambient Light Illumination Model". *Rendering Techniques*. 1998 2.

[ZXL*20] ZHANG, DONGJIU, XIAN, CHUHUA, LUO, GUOLIANG, et al. "DeepAO: Efficient Screen Space Ambient Occlusion Generation via Deep Network". *IEEE Access* 8 (2020), 64434–64441 3.