Weighted À-Trous Linear Regression (WALR) for Real-Time Diffuse Indirect Lighting Denoising

Sylvain Meunier
Advanced Micro Devices, Inc.
France
Sylvain.Meunier@amd.com

Takahiro Harada
Advanced Micro Devices, Inc.
USA
Takahiro.Harada@amd.com

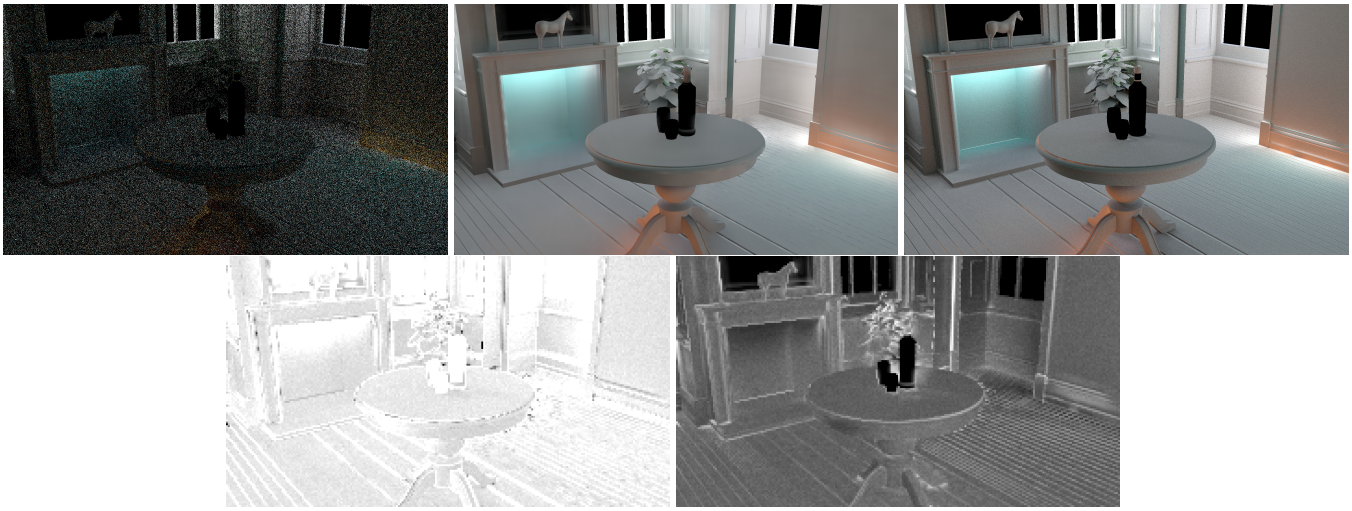

Figure 1: First line: Left image is the input diffuse indirect lighting we obtain using path tracing (1 spp), middle image is denoised using WALR, right image is a reference path-traced image (1024 spp). Second line: left image is SSIM heat map (white is better), right image is RMSE heat map (black is better).

ABSTRACT

We propose a new regression-based method for denoising images obtained from path-traced global illumination with a small number of rays per pixel. The denoising pipeline relies on temporal accumulation and a new per-pixel weighted linear regression solver we use as a spatial filter. We describe ways to maximize both performance and quality of the new solver, and provide a complete analysis and comparison to the only previous real-time regression-based method. Our work gives temporally more stable results (according to VMAF) and matches the reference images better (according to SSIM and RMSE) when running in about 3.0 ms on a modern graphics hardware at 1920x1080 resolution.

KEYWORDS

denoising, global illumination, path tracing, real-time, regression

1 INTRODUCTION

Although hardware-accelerated ray tracing is now widely available in gaming desktops and consoles, even the fastest GPUs today can still trace a small number of rays for each pixel in real-time applications. Real-time path tracing is actively researched and more effective sampling methods are continuously developed to render images with low noise. On the other hand, real-time denoising is developed for reconstructing more satisfying results from noisier lighting scenarios. Denoising is essential to make path tracing a practical solution for games. Although machine learning (ML) based denoising method is actively studied, hardware-accelerated ML approaches are still limited to high-end graphic cards, and most proposed ML methods are barely interactive without proper hardware-acceleration [Huo and Yoon 2021].

Real-time denoising methods are built around temporal accumulation , bilateral filtering, and noise-free attributes (like positions and normals from a G-Buffer) [Paris et al. 2009; Yang et al. 2020]. Bilateral filters are non-linear techniques that can blur an image while respecting strong edges and they are well suited for denoising images. Unfortunately, using large bilateral filters or repeating small bilateral filters lead to piecewise constant images.

Indirect shadows and highlights typically create color gradients near edges, and we propose to use the weighted linear regression to preserve them (Figure 2). A denoising pipeline using a blockwise solver is proposed by [Koskela et al. 2019], and we extend their denoising pipeline by replacing the blockwise solver with a pixelwise solver. We describe solutions to accelerate computations, and demonstrate that they can achieve higher quality and temporal stability with comparable performance.

Our contributions are as follows:

email: { Sylvain.Meunier, Takahiro.Harada }@amd.com
Advanced Micro Devices, Inc. Technical Report No. 22-12-1c2e, December 15, 2022.

Bilateral filter | Weighted linear regression

Figure 2: The bilateral filter erases color gradients on the small scooter elements, but the weighted linear regression preserves them. Both images use the same filter size and the same edge-stopping function. The weighted linear regression gives more freedom for using large filter sizes without reducing quality, thus filtering noisier images effectively.

- A weighted linear regression solver designed for denoising images;
- An edge-stopping function adapted to the solver;
- A real-time per-pixel denoising pipeline for reconstructing diffuse indirect lighting.

2 RELATED WORKS

Real-time path tracers are actively researched. Researchers mainly focus on direct and indirect lighting, trying to make the most of every sample under a low ray budget [Lin et al. 2022; Majercik et al. 2021; Ouyang et al. 2021].

Before path tracing was considered for real-time applications, denoisers were already used for reconstructing various stochastic effects like Ambient Occlusion (AO), Screen-Space Reflections (SSR), and soft shadows in video games [Stachowiak 2018]. Real-time denoising pipelines share most components but makes different tradeoffs to maximize performance and quality. Common practices include: building edge-stopping functions for preserving features during spatial filtering, splitting diffuse and specular interreflections, decoupling materials and lighting, adapting filter sizes for reducing the amount of bias and increasing performance, adapting temporal accumulation speed, and mitigating temporal reprojection artifacts.

Path tracing typically generates more noise, and new denoising methods were developed. Schied *et al.* adapted the bilateral filter proposed by Dammertz *et al.* in a denoising pipeline using temporal accumulation and variance estimation [Dammertz et al. 2010; Schied et al. 2017]. The temporal loop accumulates a slightly filtered indirect lighting which is further denoised using the spatial filter. They propose an adaptive filter size guided by the indirect lighting variance. Koskela *et al.* demonstrated the first real-time denoising pipeline using linear regression [Koskela et al. 2019]. They reused the pipeline Schied *et al.* proposed, and they proposed [Schied et al. 2017] to replace the bilateral filter with a regression-based method porting the most efficient ideas from offline denoising methods [Moon et al. 2014, 2015, 2016]. Zhdan proposed an alternative denoising pipeline where a bilateral filter is used inside the temporal loop [Zhdan 2020]. This is presented as an optimization for reducing the amount of blur used during reconstruction. The effective filter size results from the temporal accumulation speed and the bilateral filter size, and they adapt both to reduce the reconstruction bias.

3 WEIGHTED À-TROUS LINEAR REGRESSION

Blockwise Multi-Order Feature Regression (BMFR) solver is based on the HouseHolder QR factorization [Golub and Van Loan 1996; Koskela et al. 2019] and existing parallel implementations focus on solving huge least square problems on various parallel systems. Denoising is different by nature, and we solve many least square problems by sharing a lot of samples. Their blockwise implementation workarounds the lack of a proper parallel implementation, and the pipeline is built to remove the solver blocky artifacts. Our solver reuses neighbor computations between pixels, thus parallelizing computations, and makes a per-pixel denoising pipeline possible.

WALR (Weighted À-trous Linear Regression) is a weighted linear regression solver built on top of an edge-aware à-trous averaging technique and edge tracing.

3.1 Weighted linear regression using averages

Linking weighted linear regression and feature averages is straightforward. Given n samples containing one noisy value Y_i and m noise-free features $X_{i,j}$:

$$\{Y_i, X_{i,1}, \dots, X_{i,m}\}_{i=1}^{n},$$

We are going to reconstruct denoised value $\hat{Y}_i$ as a linear combination of these noise-free features $X_{i,j}$ as follows:

$$\hat{Y}_i = \sum_{j=1}^{m} X_{i,j}\hat{\beta}_j,$$

where $\hat{\beta}_j$ are the weights for these values. We also set $X_{i,1} = 1$, thus, the intercept is handled as these features, and later equations are simplified. Estimating $\hat{\beta}_j$ parameters is achieved by solving this optimization problem:

$$\arg\min_{\hat{\beta}_j} \sum_{i=1}^{n} w_i \left(Y_i - \sum_{j=1}^{m} X_{i,j}\hat{\beta}_j \right)^2,$$

where w_i is a positive weight. The problem has a unique solution given by the normal equations:

$$\frac{\partial}{\partial \hat{\beta}_k} \sum_{i}^{n} w_i \left(Y_i - \sum_{j}^{m} X_{i,j}\hat{\beta}_j \right)^2 = 0 \text{ with } k \in \{1, \dots, m\}.$$

Let us take an example where $m = 3$, so we can expand the system of equations:

$$\begin{aligned}
\overline{XX}_{1,1}\hat{\beta}_1 + \overline{XX}_{1,2}\hat{\beta}_2 + \overline{XX}_{1,3}\hat{\beta}_3 - \overline{YX}_1 &= 0,\\
\overline{XX}_{1,2}\hat{\beta}_1 + \overline{XX}_{2,2}\hat{\beta}_2 + \overline{XX}_{2,3}\hat{\beta}_3 - \overline{YX}_2 &= 0,\\
\overline{XX}_{1,3}\hat{\beta}_1 + \overline{XX}_{2,3}\hat{\beta}_2 + \overline{XX}_{3,3}\hat{\beta}_3 - \overline{YX}_3 &= 0,
\end{aligned}$$

where:

$$\overline{XX}_{p,q} = \frac{\sum_{i=1}^{n} X_{i,p} X_{i,q} w_i}{n},$$
$$\overline{YX}_{p} = \frac{\sum_{i=1}^{n} X_{i,p} Y_i w_i}{n},$$

then we rewrite the system using linear algebra:

$$\begin{bmatrix} \overline{XX}_{1,1} & \overline{XX}_{1,2} & \overline{XX}_{1,3} \\ \overline{XX}_{1,2} & \overline{XX}_{2,2} & \overline{XX}_{2,3} \\ \overline{XX}_{1,3} & \overline{XX}_{2,3} & \overline{XX}_{3,3} \end{bmatrix} \begin{bmatrix} \hat{\beta}_1 \\ \hat{\beta}_2 \\ \hat{\beta}_3 \end{bmatrix} = \begin{bmatrix} \overline{YX}_1 \\ \overline{YX}_2 \\ \overline{YX}_3 \end{bmatrix},$$

and generalize easily to:

$$\begin{bmatrix} \overline{XX}_{1,1} & & \overline{XX}_{1,m} \\ & \ddots & \\ \overline{XX}_{1,m} & & \overline{XX}_{m,m} \end{bmatrix} \begin{bmatrix} \hat{\beta}_1 \\ \vdots \\ \hat{\beta}_m \end{bmatrix} = \begin{bmatrix} \overline{YX}_1 \\ \vdots \\ \overline{YX}_m \end{bmatrix}.$$

function *Cholesky*($\overline{XX}$)
- $C \leftarrow 0$
- **foreach** $i \in \{1 \dots m\}$ **do**
  - $C_{i,i} \leftarrow \overline{XX}_{i,i}$
  - **foreach** $j \in \{1 \dots i-1\}$ **do**
    - $C_{i,i} \leftarrow C_{i,i} - C_{i,j} C_{i,j}$
  - $C_{i,i} \leftarrow \sqrt{max(C_{i,i}, \epsilon_i)}$
  - **foreach** $j \in \{i+1 \dots m\}$ **do**
    - $C_{j,i} = \overline{XX}_{i,j}$
    - **foreach** $k \in \{1 \dots i-1\}$ **do**
      - $C_{j,i} \leftarrow C_{j,i} - C_{i,k} C_{j,k}$
    - $C_{j,i} \leftarrow C_{j,i} / C_{i,i}$
- **return** C

Algorithm 1: We decompose the symmetric and positive-definite matrix $\overline{XX}$ into a triangular matrix C satisfying $CC^t = \overline{XX}$. ϵ_i should be big enough to remove precision issues.

function *Forward*(C, Y)
- **foreach** $i \in \{1 \dots m\}$ **do**
  - $F_i \leftarrow Y_i$
  - **foreach** $j \in \{i-1 \dots 1\}$ **do**
    - $F_i \leftarrow F_i - C_{i,j} F_j$
  - $F_i \leftarrow F_i / C_{i,i}$
- **return** F

function *Backward*(C, F)
- **foreach** $j \in \{m \dots 1\}$ **do**
  - $X_j \leftarrow F_j$
  - **foreach** $i \in \{m \dots j+1\}$ **do**
    - $X_j \leftarrow X_j - C_{i,j} X_i$
  - $X_j \leftarrow X_j / C_{j,j}$
- **return** X

Algorithm 2: Forward and backward substitutions are used for computing $\hat{\beta}$ from the Cholesky decomposition C and $\overline{YX}$. Both Y and F can be scalars or vectors, thus making noisy colors cheap to support.

The average matrix $\overline{XX}$ is always symmetric, so if the matrix is also positive-definite, we can solve this system using the Cholesky decomposition (Algorithm 1) followed by forward and backward substitutions (Algorithm 2). In practice, we assume $\overline{XX}$ is positive-definite.

3.2 Edge-aware à-trous averages

The WALR solver needs to compute a lot of edge-aware averages with large windows. This is computationally expensive, so we propose adapting the à-trous wavelet transform to compute these averages efficiently. The à-trous wavelet transform hierarchically filters over multiple iterations. This transform decomposes any image into a smooth base image and a set of detailed images. Reconstructing the input image is just the summation of these images. Detailed images hold the noise, and denoising is achieved by discarding ([Dammertz et al. 2010]) or weighting ([Hanika et al. 2011]) details. We adopt the simpler [Dammertz et al. 2010] method:

$$Z_{t+1}(x,y) = \frac{\sum_{u,v} f(x,y,u,v) Z_t(x+u, y+v)}{\sum_{u,v} f(x,y,u,v)},$$

with:

$$f(x,y,u,v) = h_t(u) h_t(v) w(x,y,x+u,y+v),$$

where $Z_0(x,y)$ is a noisy color image, w is a positive edge-stopping function, and h_t is a separable filter kernel. [Dammertz et al. 2010] uses the edge-aware à-trous wavelet transform for approximating a Gaussian bilateral filter. Each iteration increases the h_t kernel footprint by introducing $2^t - 1$ zeros between the h_0 kernel values. The method is fast because the edge-stopping function and spatial filter evaluations are amortized between neighbor pixels. We want to compute edge-aware averages as fast as possible, and adapting the transform requires three modifications:

- We specialize the transform using a box filter kernel $h_0 = (1,1,1)$.
- We change the number of zeros the à-trous transform adds for each iteration, so each sample is averaged once.
- We maintain the sample count of averages for computing the weighted average of averages, so each sample is averaged once iteratively (Figure 3).

The box filter kernel is defined by (only first three iterations):

$$h_0 = (1,1,1),$$
$$h_1 = (1,0,0,1,0,0,1),$$
$$h_2 = (1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1).$$

The sample count of averages is given by:

$$W_0(x,y) = 1,$$
$$W_{t+1}(x,y) = \sum_{u,v} f(x,y,u,v) W_t(x+u, y+v),$$

then $\overline{XX}_{i,j}$ averages are iteratively computed with:

$$\overline{XX}_{i,j,0}(x,y) = X_i(x,y) X_j(x,y),$$
$$\overline{XX}_{i,j,t+1}(x,y) = \frac{\sum_{u,v} f(x,y,u,v) W_t(x+u, y+v) \overline{XX}_{i,j,t}(x+u, y+v)}{\sum_{u,v} f(x,y,u,v) W_t(x+u, y+v)},$$

and $\overline{YX}_i$ averages with:

$$\overline{YX}_{i,0}(x,y) = Y(x,y) X_i(x,y)$$

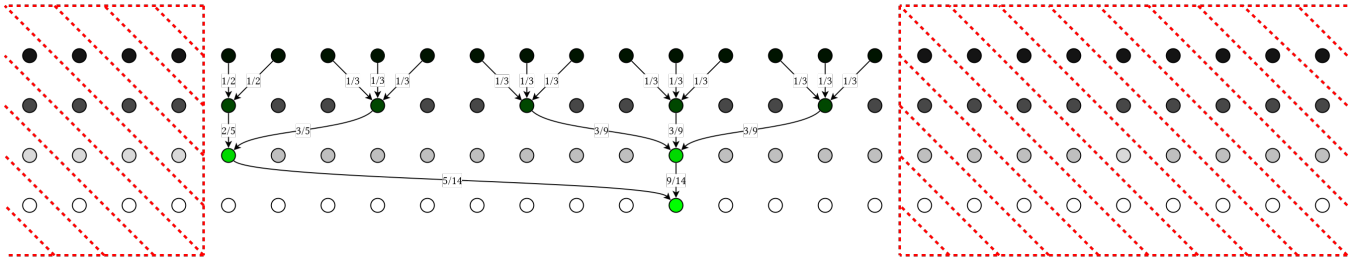

Figure 3: Three iterations of the edge-aware à-trous average method. Top line dots are the first iteration inputs, bottom line dots are the third iteration results. Dashed dots are discarded by the edge-stopping function. We only plot the strictly positive weights. The 14 dots are weighed $1/14$ after three iterations, so the method computes the average.

No edge tracing Edge tracing

Figure 4: Evaluating the edge-stopping function using the center pixel and a corner pixel generates aliasing on the wall between suspended objects. Edge tracing and jittering mitigate this issues.

$$\overline{YX}_{i,t+1}(x,y) = \frac{\sum_{u,v} f(x,y,u,v) W_t(x+u,y+v) \overline{YX}_{i,t}(x+u,y+v)}{\sum_{u,v} f(x,y,u,v) W_t(x+u,y+v)}.$$

3.3 Edge tracing

Although the edge-aware à-trous transform improves the performance of computing averages, it can bring artifacts, as shown in Figure 4. We analyze issues creating these artifacts and propose a solution for both.

The first issue is the edge-stopping function w being evaluated between pixels (x, y) and $(x + u, y + v)$. h_t kernel generates a set of 3x3 positions centered on the filtered pixel for each iteration. h_0 covers a 3x3 pixel area while h_1, h_2 and h_3 cover respectively 7x7, 19x19 and 55x55 areas. If we compare positions and normals for detecting edges, small objects might be ignored, and lighting from unrelated surfaces might be denoised together. We propose to mitigate this issue by using edge tracing. Edge tracing evaluates the edge-stopping function along the segment between (x, y) and $(x+u, y+v)$, and returns its smallest value; thus, no object is ignored.

The second issue happens when the edge-stopping function changes abruptly between two filtered pixels, thus discarding different averages. Averages computed for these two pixels might be different, and seams might be created, thus revealing the à-trous grids. We propose to replace these seams with soft transitions by replacing each discarded average with another one picked randomly on its segment near the edge (so no average is discarded). The replacement average weight is also evaluated again. This method introduces small errors in computed averages near object edges.

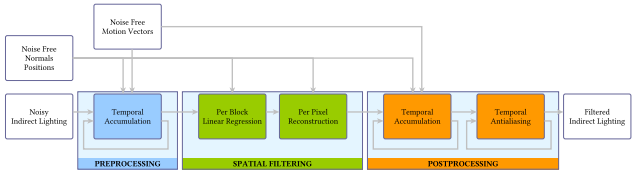

Figure 5: BMFR pipeline includes a specific temporal accumulation phase for removing blocky artifacts.

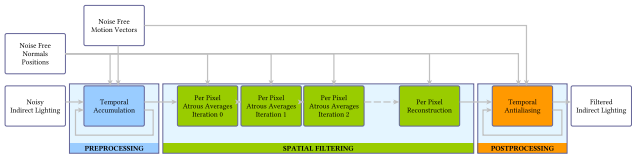

Figure 6: WALR pipeline includes an iterative spatial filtering phase. Adding iterations expands the filter size.

4 DENOISING PIPELINE

We slightly adjusted the BMFR pipeline (Figure 5) proposed by [Koskela et al. 2019] for easily comparing performance and quality. The WALR pipeline (Figure 6) can be divided into three main phases: preprocessing, spatial filtering, and postprocessing. Both preprocessing and postprocessing phases rely on temporal accumulation for reducing noise and artifacts, while the spatial filtering phase uses WALR.

4.1 Inputs

The inputs consist of a noisy path-traced indirect lighting image and its accompanying noise-free motion vectors, view space positions, and normals. Global frame jittering is important for postprocessing, as it helps remove aliasing and small artifacts.

We want to preserve the quality of materials and textures while we try to maximize the number of samples we average together. Thus, we remove material evaluation on first vertices and delegate the materials and direct lighting composition to the renderer.

4.2 Preprocessing

The preprocessing phase reduces the indirect lighting noise using temporal accumulation. Temporal accumulation introduces less

bias than spatial filtering for removing noise, but it adds temporal lag, so we need to find a good tradeoff between lag and bias, and the proposed pipeline offers enough flexibility to cover most environments. Scenes with fast-moving lights and occluders would require a shorter temporal loop and a larger spatial filter, while almost static scenes could use a longer temporal loop and a smaller spatial filter.

The preprocessing temporal accumulation uses three components: temporal reprojection, history rejection, and moving average. The temporal reprojection uses motion vector dilation and bicubic texture sampling for resampling both the preprocessing history and the postprocessing history as described in Sec. 4.4. If a global frame jittering is enabled, we remove it from motion vectors before reprojecting histories. The preprocessing history rejection is done by comparing the current and reprojected depth. The reprojected depth uses the depth offset (third coordinate of the motion vector) for accurately comparing depths. We need to consider edges when using depth rejection. If jittering is used for rendering geometry buffers, depth rejection will cancel accumulation and limit denoising on edges. In practice, edges are slightly noisier, and we rely on the spatial filtering phase to mitigate this issue.

We do not use the exponential moving average because it lowers denoising speed by weighing the first frame more than the later ones. We slightly adjust ([Koskela et al. 2019]) technique and start by computing a cumulative moving average of the samples, storing the sample count, and using an accumulation weight proportional to the sample count. After a fixed number of samples, we return to an exponential moving average by clamping the accumulation weight, so older history samples vanish.

4.3 Spatial filtering

WALR is used to filter the remaining noise out of the preprocessed indirect lighting. We propose an edge-stopping function for diffuse indirect lighting. The function uses normals and positions. We form a plane using the center position and normal, then we remap the distance between the plane and the traced positions in $[0, 1]$ with user parameters and use this value as a weight. The edge-tracing algorithm can create some star-shaped artifacts with temporal accumulation ghosting, and we rely on global frame jittering for fixing these issues during the postprocessing phase.

4.4 Postprocessing

The postprocessing phase uses temporal accumulation to remove small aliasing and remaining instability problems using global frame jittering. It also uses three components: temporal reprojection, history rectification, and moving average. The temporal reprojection of the postprocessing history is done during the preprocessing phase using the same resampling technique. We use variance clipping instead of depth rejection because the filtered indirect lighting noise is low, and variance clipping is a good tradeoff between antialiasing and ghosting for history rectification. We also reuse the preprocessing cumulative moving average technique.

The processing phase is almost a standard Temporal Anti-Aliasing (TAA) implementation and could be removed if the renderer already uses TAA or another temporal upsampling method.

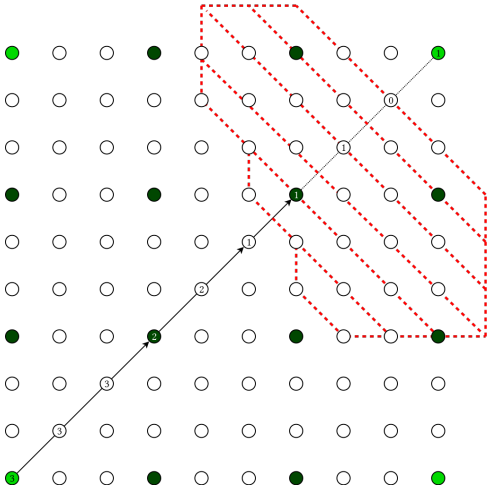

Figure 7: Edge tracing is computed between the center (bottom left dot) and a corner (top right dot). Step sizes (arrows) are chosen using the last iteration SDF. We plot the SDF along the traced edge (numbers in dots). Once an edge is found (dashed dots), edge tracing is stopped.

5 IMPLEMENTATION

We already described all the components of the WALR pipeline. This section presents the implementation we used for generating quality and performance results.

5.1 Features selection

Before discussing feature selection, we assume diffuse indirect lighting, and we need to understand how weights and features work together. Applying a null weight to a sample simply discards it from the reconstruction, while using a weight from 1 to 0 offers a fade-out transition. Features distribute noisy colors in the feature space when optimizing the linear model. During reconstruction, we interpolate or extrapolate the denoised pixel colors from the linear model, its parameters, and the denoised pixel features.

The BMFR solver does not use weights. Edges and details are preserved by selecting a set of features computed from attributes. They proposed to use world space normals, normalized world space positions, and normalized squared world space positions. Each feature preserves an interesting property of indirect light during filtering. Positions are used to preserve object edge, so color samples lying on different surfaces are not blended. Normals are used to preserve irradiance directionality. Squared positions are used for preserving indirect shadows, so the estimated model looks like a plane with a hole or a bump (typically, shadows over a bright surface).

The WALR solver can use the BMFR features if we set $w = 1$, but the tradeoff between performance and quality is not great. The performance cost of adding features is higher for WALR than BMFR. This is due to reading and writing a lot of data to global memory for computing averages over several iterations. Using weights for preserving object edges is less expensive, so we remove positions from features. Squared positions are mandatory for BMFR blocks,

but WALR per pixel processing could use an adaptive filter size for preserving indirect shadows and highlights. Using an adaptive radius would also increase performance. We keep normals as features because we want to interpolate the indirect lighting between normals.

5.2 Performance improvements

The main WALR bottlenecks are bandwidth and requests used for reading and writing data to the global memory, so we need to write a minimal amount of data and pack it carefully to achieve the best performance. Some presented optimizations are approximations, trading small quality losses for large performance gains.

View space normals target the camera, so we use octahedral mapping ([Cigolle et al. 2014]) for transforming normals into two features instead of three. When normal mapping creates degenerate normals, we flip them. There are no visible seams during reconstruction because mapped normals do not wrap around in front of the camera.

In Section 3, we set $X_{i,0} = 1$ for simplifying equations; thus, the first average is $\overline{XX}_{0,0} = 1$. We do not need to store or compute $\overline{XX}_{0,0}$ and use 1 during reconstruction. We also simplify equations by removing W from computations, and this approximation gives more weight to samples near edges.

Features include indirect lighting and remapped normals, so the range of all averages is proportional to the indirect lighting range. If the renderer stores lighting using RGBA16_FLOAT texture format, averages should use RGBA16_FLOAT too. If the renderer uses R11G11B10_FLOAT instead, averages can use this format too, but we need to take care of the normal signs. Octahedral mapping outputs in $[-1, +1]$, and we propose to remap it to $[0, 1]$. The error will not be the same for negative and positive features, but we find this preferable to storing signs in an additional texture. The texture format used for storing averages also affects the Cholesky decomposition, and epsilons must be chosen accordingly. All the presented results used FP16 data formats.

Packed FP16 instructions could be used for computing averages and edge tracing. They increased the occupancy of all kernels, but we did not observe any performance gain and did keep FP32 for computations.

When samples are made of one noisy color and a remapped normal, we need to store 14 values for averages. If we use the RGBA16_FLOAT texture format, averages are packed with three textures with formats RGBA32_UINT, RG32_UINT, and R32_UINT. If we use R11G11B10_FLOAT instead, averages are packed with two textures with formats RGBA32_UINT and R32_UINT.

During each WALR iteration, edge tracing is used to evaluate weights. We implement edge tracing extending DDA (Digital Differential Analyzer) with an SDF (Signed Distance Field) for better performance (Figure 7). The SDF stores the closer edge distance found during the last WALR iteration. This distance is computed using an edge-stopping function centered on the traced position instead of the actual filtered position; therefore, we might overestimate the distance to the edge. We propose to use a percentage of the distance instead for being more conservative and slightly slower. The SDF is updated during each iteration of WALR by storing the closer edge distance amongst eight traced edges. The edge-tracing algorithm uses the SDF for stepping by a conservative number of pixels. The SDF stores the closer edge of the last iteration for all samples Edge tracing access both positions and the SDF, so we pack positions and distances together using a unique RGBA16_FLOAT texture. If the renderer uses a pinhole camera model, we could pack depth and SDF using R32_UINT and reduce bandwidth pressure further.

All shaders can be inlined. No loop is necessary for edge tracing because we know the maximum length of segments at compilation time. In practice, we do four iterations, and the maximum length is 27 pixels, so the shader disassembly stays small.

6 RESULTS

[Koskela et al. 2019] demonstrated BMFR denoising pipeline was competitive with real-time state-of-the-art pipelines. We show WALR pipeline provides higher quality for similar execution timings. We believe WALR denoising pipeline is also competitive with others, but we focus on comparing the solvers instead of the pipelines. An accurate comparison with other denoising pipelines would require further developments (see Section 7).

The proposed method has been implemented using DirectX, and the experiments were performed on a PC with an AMD Radeon™ PRO W6800 GPU, an AMD Ryzen™ Threadripper™ 3960X CPU, 128GB RAM, Windows 10, and AMD Software PRO Edition 22.6.1. Following [Koskela et al. 2019], we use these quality metrics for comparing reference path-traced image sequences and denoised image sequences:

- Root Mean Square Error (RMSE).
- Structural SIMilarity (SSIM) [Wang et al. 2004].
- Video Multi-Method Assessment Fusion (VMAF) [Aaron et al. 2015].

All image sequences are exposed and tonemapped before metrics are computed. These metrics are commonly used for providing objective (RMSE, SSIM) and subjective (VMAF) ratings to denoised image sequences. VMAF also rates temporal stability. We provide heat maps for locating main quality issues easily. White spots are the worst issues in RMSE heat maps, while black spots are for SSIM heat maps.

We used AMD Radeon™ ProRender to generate a sequence of 1080 p images, including diffuse indirect lighting and accompanying attributes [Advanced Micro Devices, Inc. 2020]. Motion vectors we reconstruct from these sequences have up to 1-pixel errors for the target resolution. This issue coupled with global frame jittering creates problems on object edges during temporal accumulation, so we disabled global frame jittering instead of dealing with additional artifacts in the denoising pipelines.

We also developed REFerence Weighted Linear Regression (REFWLR). This solver shares the filter size, the edge-stopping function, and other heuristics with WALR, but it uses no à-trous transform, with edge tracing between the center and all other pixels in the filter window, and we don't replace discarded samples (like we did for averages). This reference method almost always delivers better quality but is barely interactive. We use it to quantify the error we introduce with WALR optimizations and average replacement.

Denoising pipeline results depend heavily on the input color noise, so we picked four scenes with different noise profiles. All

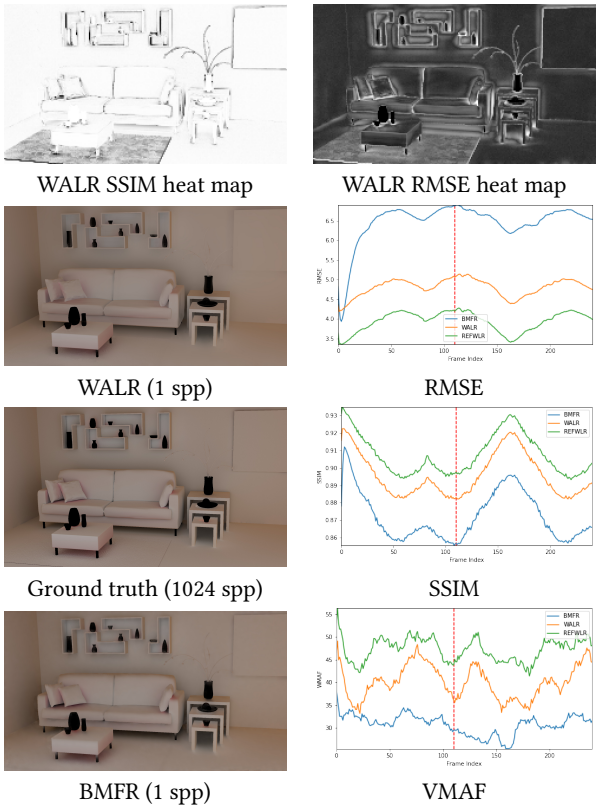

Figure 8: The Modern Living Room. Noise is very low, and quality is very high. BMFR blocky artifacts are only visible in the first frames. REFWLR (green curve) and WALR (orange curve) are close, and WALR outperforms BMFR (blue curve). The main source of errors is the carpet on the ground. An interesting point is that the RMSE heat map gives an idea of the small errors that edge tracing and average replacement add around suspended objects.

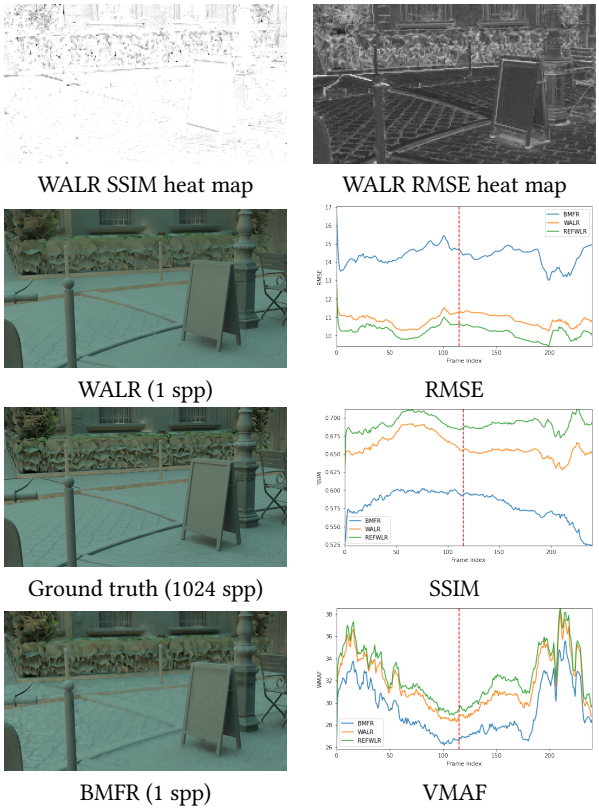

Figure 9: Bistro Exterior. Noise is moderate, and all the denoisers provide acceptable quality. BMFR outputs a few artifacts. REFWLR (green curve) and WALR (orange curve) are close, and WALR outperforms BMFR (blue curve). The main source of error is the foliage shadows in the background.

scenes include an animated camera moving quickly enough for reviewing disocclusions:

- The Modern Living Room (TMLR) is an indoor scene where the only light source is a spot. Figure 8.
- Bistro Exterior (BE) is an outdoor scene with a sun and a sky. Figure 9.
- The Grey White Room (TGWR) is an indoor scene with a sun and a sky. Figure 10.
- The Dining Room Again (TDRA) is an indoor scene where the sun and the sky are behind window shutters, two spot lights add color bleeding on the table. Figure 11.

Tables 1 and 2 show the timings by phase and the total time spent executing both denoising pipelines. We can see that the total time ranges are close between both pipelines, so comparing quality metrics makes more sense. There are still differences we can explain. BMFR preprocessing is slightly slower because its implementation uses larger texture formats than WALR implementation for storing motion vectors, history, and attributes. BMFR postprocessing is

	Preprocess	Spatial Filter	Postprocess	Total
TMLR	0.702	1.611	0.492	2.806
BE	0.613	1.557	0.433	2.604
TGWR	0.666	1.728	0.451	2.846
TDRA	0.677	1.531	0.456	2.665

Table 1: BMFR averaged timings (in ms) over multiple runs and various sampling rates.

	Preprocess	Spatial Filter	Postprocess	Total
TMLR	0.363	2.442	0.082	2.887
BE	0.357	2.707	0.078	3.144
TGWR	0.364	2.483	0.081	2.929
TDRA	0.360	2.529	0.079	2.969

Table 2: WALR averaged timings (in ms) over multiple runs and various sampling rates.

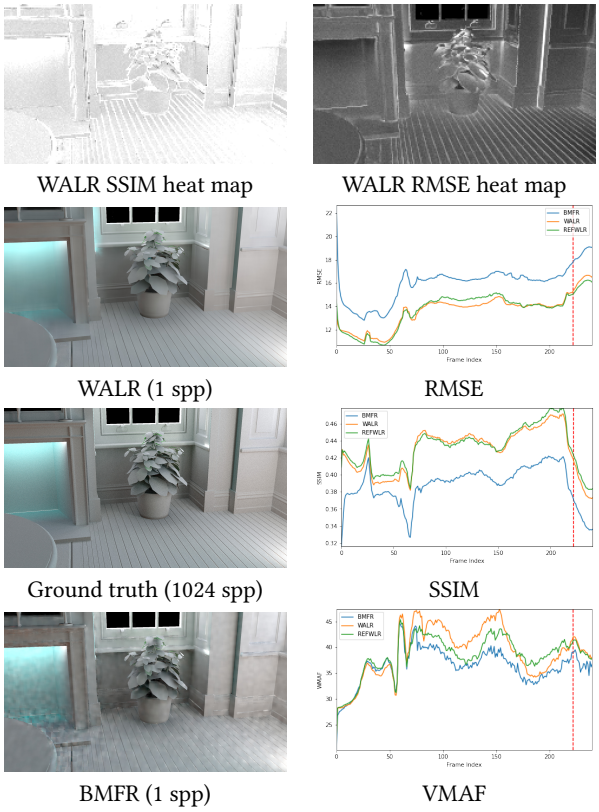

Figure 10: The Grey White Room. Noise is significant, and BMFR outputs a lot of artifacts. REFWLR (green curve) and WALR (orange curve) are close, and WALR outperforms BMFR (blue curve). The main sources of errors are the plant shadows and a blue lighting leak near the chimney (due to edge tracing boosting a ghosting issue).

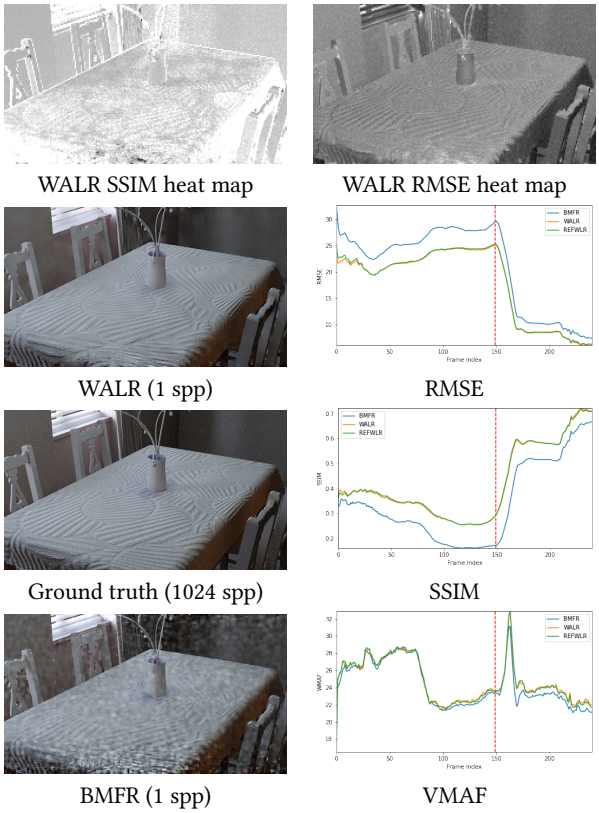

Figure 11: The Dining Room Again. Noise is very high, and overall quality is low for all the denoisers. REFWLR and WALR output a few artifacts, while BMFR outputs a lot. REFWLR (green curve) and WALR (orange curve) are close, and WALR outperforms BMFR (blue curve). The main source of errors is the tablecloth.

also slower because the pipeline requires an extra temporal loop for cleaning the blocky artifacts introduced by the solver. WALR spatial filtering is slightly slower because we use four iterations instead of three, so timings are close for both pipelines. Using three iterations would make WALR way faster. We can note that BMFR uses a filter area of 32x32 pixels (a limitation of the provided implementation), whereas WALR uses a filter area of 80x80 pixels.

7 GOING FURTHER

We proposed a denoising pipeline for comparing WALR and BMFR, and there are improvement opportunities:

- We could use an adaptive filter size based on the actual indirect lighting variance, distance, or another heuristic. This would bring greater performance and also increase quality (Figure 12).
- The à-trous transform constrains the filter size heavily. Another technique might be used ([Gwosdek et al. 2011]).
- With more freedom to choose the filter size, anisotropic filtering (computing averages over projected circles) could be used for grazing angles.

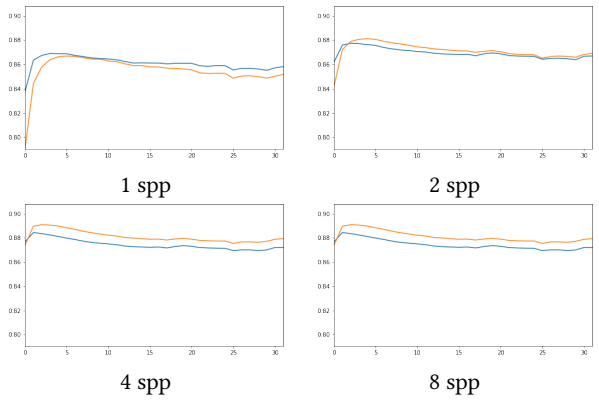

Figure 12: WALR SSIM for three iterations (orange curve) and four iterations (blue curve). The space between curves gives insights about the used filter size. Close curves signify that the filter size is too large, while distant curves indicate a larger filter could further reduce the noise.

- Edge tracing cannot walk over edges, so we cannot average similar surfaces behind occluders, such as grids or fences. We could build a statistical profile of averages and merge them to reduce noise further.
- Some heuristics (edge-stopping function, feature selection...) are valid because we assume diffuse indirect lighting. Denoising specular interreflections could be explored.
- After optimization, the main bottleneck is still reading and storing averages at the target resolution. Decoupling averages' resolution from attribute resolution seems a promising way to increase performance.

8 CONCLUSION

In this paper, we presented WALR, a new weighted linear regression solver built on top of an edge-aware à-trous averaging technique. This pixelwise solver efficiently replaces the blockwise solver proposed by [Koskela et al. 2019] and makes weighted linear regression a practical solution for real-time denoising.

ACKNOWLEDGMENTS

We thank Bruno Stefanizzi and Prashanth Kannan for the support of the research, as well as Aaryaman Vasishta, Artem Kharytoniuk, Fabio Camaiora, Guillaume Boissé, and Oleksandr Kupriyanchuk for their help in reviewing this paper. We also thank the blendswap.com artists, especially Wig42, for providing most scenes we used and modified for developing WALR. The Amazon Bistro scene is part of ORCA (Open Research Content Archive) [Lumberyard 2017]. AMD, AMD Radeon, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

Anne Aaron, Zhi Li, Megha Manohara, Joe Yuchieh Lin, Eddy Chi-Hao Wu, and C.-C Jay Kuo. 2015. Challenges in cloud based ingest and encoding for high quality streaming media. In 2015 IEEE International Conference on Image Processing (ICIP). 1732–1736. https://doi.org/10.1109/ICIP.2015.7351097

Advanced Micro Devices, Inc. 2020. Radeon™ ProRender 2.0. https://gpuopen.com/radeon-pro-render/

Zina H. Cigolle, Sam Donow, Daniel Evangelakos, Michael Mara, Morgan McGuire, and Quirin Meyer. 2014. A Survey of Efficient Representations for Independent Unit Vectors. Journal of Computer Graphics Techniques (JCGT) 3, 2 (17 April 2014), 1–30. http://jcgt.org/published/0003/02/01/

Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. 2010. Edge-Avoiding À-Trous Wavelet Transform for Fast Global Illumination Filtering. In Proceedings of the Conference on High Performance Graphics (Saarbrucken, Germany) (HPG '10). Eurographics Association, Goslar, DEU, 67–75.

Gene H. Golub and Charles F. Van Loan. 1996. Matrix Computations (3rd Ed.). Johns Hopkins University Press, USA.

Pascal Gwosdek, Sven Grewenig, Andrés Bruhn, and Joachim Weickert. 2011. Theoretical Foundations of Gaussian Convolution by Extended Box Filtering. In Proceedings of the Third International Conference on Scale Space and Variational Methods in Computer Vision (Ein-Gedi, Israel) (SSVM'11). Springer-Verlag, Berlin, Heidelberg, 447–458. https://doi.org/10.1007/978-3-642-24785-9_38

Johannes Hanika, Holger Dammertz, and Hendrik Lensch. 2011. Edge-Optimized À-Trous Wavelets for Local Contrast Enhancement with Robust Denoising. Computer Graphics Forum 30, 7 (2011), 1879–1886. https://doi.org/10.1111/j.1467-8659.2011.02054.x arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1467-8659.2011.02054.x

Yuchi Huo and Sung-eui Yoon. 2021. A survey on deep learning-based Monte Carlo denoising. COMPUTATIONAL VISUAL MEDIA 7, 2 (2021), 169–185.

Matias Koskela, Kalle Immonen, Markku Mäkitalo, Alessandro Foi, Timo Viitanen, Pekka Jääskeläinen, Heikki Kultala, and Jarmo Takala. 2019. Blockwise Multi-Order Feature Regression for Real-Time Path-Tracing Reconstruction. ACM Trans. Graph. 38, 5, Article 138 (jun 2019), 14 pages. https://doi.org/10.1145/3269978

Daqi Lin, Markus Kettunen, Benedikt Bitterli, Jacopo Pantaleoni, Cem Yuksel, and Chris Wyman. 2022. Generalized Resampled Importance Sampling: Foundations of ReSTIR. ACM Trans. Graph. 41, 4, Article 75 (jul 2022), 23 pages. https://doi.org/10.1145/3528223.3530158

Amazon Lumberyard. 2017. Amazon Lumberyard Bistro, Open Research Content Archive (ORCA). http://developer.nvidia.com/orca/amazon-lumberyard-bistro `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`.

Zander Majercik, Thomas Mueller, Alexander Keller, Derek Nowrouzezahrai, and Morgan McGuire. 2021. Dynamic Diffuse Global Illumination Resampling. In ACM SIGGRAPH 2021 Talks (Virtual Event, USA) (SIGGRAPH '21). Association for Computing Machinery, New York, NY, USA, Article 24, 2 pages. https://doi.org/10.1145/3450623.3464635

Bochang Moon, Nathan Carr, and Sung-Eui Yoon. 2014. Adaptive Rendering Based on Weighted Local Regression. ACM Trans. Graph. 33, 5, Article 170 (sep 2014), 14 pages. https://doi.org/10.1145/2641762

Bochang Moon, Jose A. Iglesias-Guitian, Sung-Eui Yoon, and Kenny Mitchell. 2015. Adaptive Rendering with Linear Predictions. ACM Trans. Graph. 34, 4, Article 121 (jul 2015), 11 pages. https://doi.org/10.1145/2766992

Bochang Moon, Steven McDonagh, Kenny Mitchell, and Markus Gross. 2016. Adaptive Polynomial Rendering. ACM Trans. Graph. 35, 4, Article 40 (jul 2016), 10 pages. https://doi.org/10.1145/2897824.2925936

Y. Ouyang, S. Liu, M. Kettunen, M. Pharr, and J. Pantaleoni. 2021. ReSTIR GI: Path Resampling for Real-Time Path Tracing. Computer Graphics Forum 40, 8 (2021), 17–29. https://doi.org/10.1111/cgf.14378 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14378

Sylvain Paris, Pierre Kornprobst, Jack Tumblin, and Frédo Durand. 2009. .

Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. 2017. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In Proceedings of High Performance Graphics (Los Angeles, California) (HPG '17). Association for Computing Machinery, New York, NY, USA, Article 2, 12 pages. https://doi.org/10.1145/3105762.3105770

Tomasz Stachowiak. 2018. Stochastic All The Things: Raytracing in Hybrid Real-Time Rendering. https://www.ea.com/seed/news/seed-dd18-presentation-slides-raytracing.

Zhou Wang, A.C. Bovik, H.R. Sheikh, and E.P. Simoncelli. 2004. Image quality assessment: from error visibility to structural similarity. IEEE Transactions on Image Processing 13, 4 (2004), 600–612. https://doi.org/10.1109/TIP.2003.819861

Lei Yang, Shiqiu Liu, and Marco Salvi. 2020. A Survey of Temporal Antialiasing Techniques. Computer Graphics Forum 39, 2 (2020), 607–621. https://doi.org/10.1111/cgf.14018 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.14018

Dmitry Zhdan. 2020. Fast Denoising with Self Stabilizing Recurrent Blurs. https://developer.download.nvidia.com/video/gputechconf/gtc/2020/presentations/s22699-fast-denoising-with-self-stabilizing-recurrent-blurs.pdf.