Progressive Spatiotemporal Variance-Guided Filtering

Jan Dundr*
Supervised by: doc. Ing. Jiří Bittner, Ph.D.†

Department of Computer Graphics and Interaction
Czech technical university in Prague
Prague / Czech Republic

Abstract

Path tracing is still very hard to do in real-time due to its complex recursive light interactions: even the newest GPUs struggle to keep real-time framerates for more than just a few samples per pixel, which can lead to a very noisy output or even no useful data for some more problematic areas.

This paper uses a recent approach to processing the resulting image: demodulated samples are accumulated from previous frames using reprojection and subsequently filtered by a fast bilateral filter constrained by normals, depth, and variance (which will stop blurring valuable details). This results in a temporally stable noise-free output which converges in a few frames.

We implemented the method using OpenGL and incorporated it in an existing high-performance CPU path tracer. We extended the method by putting it in the progressive rendering framework, where initially less than one sample per pixel is shot to increase interactivity. We evaluate the performance and visual quality of this algorithm in several test cases with mostly diffuse illumination.

Keywords: path tracing, global illumination, real-time rendering, bilateral filtering, temporal filtering, OpenGL

1 Introduction

Computing accurate real-time illumination of a 3d scene is a hard problem. The rendering equation [4] is conceptially simple: outgoing radiance at a surface location is equal to emitted and reflected radiance using an appropriate BRDF model. This must, however, be true for every surface position in the scene and the resulting integral is too complex to be solved analytically. Path tracing [4] (and its many optimized variants) has been the golden standard for calculating a global illumination with photorealistic results for a long time. It approximates the lighting equation in a Monte Carlo fashion by shooting a large number of randomized rays into the scene.

This process can consistently simulate a lot of otherwise hard to fake light phenomena (like caustics, depth of field, etc.) and is very general (one standard setup leads to a photorealistic lighting results with many complex lighting phenomena), but computing a single image can take minutes, hours or even more. GPUs can help: rendering is much faster for simple scenes and a few times quicker for more complex ones (due to big and incoherent memory requirements, if they even fit there). Real-time applications, on the other hand, require at least tens of frames per second.

Our filter works with path-traced image as an input. Very low sample count (around 1 sample per pixel) is used to keep real-time framerates, light is accumulated from previous frames using reprojection, and the image is filtered both in space and time using a fast wavelet-based bilateral filter constrained by normals, depth, and variance (pixels are filtered more when there's a larger variance). The whole process is implemented on GPU using OpenGL, it is executed in a matter of milliseconds even on weaker GPUs and aims to use spatially and temporally localized data. It significantly increases apparent sample rate (results looks like more samples were used) at the cost of bias in situations when there is not enough data yet.

Algorithm needs normal and depth buffers every frame. These buffers can be either generated by primary rays during path tracing or rasterized beforehand. Rasterization is usually faster, but complicates implementation (sometimes requires a completely new rendering pipeline) and can be much slower in case of very complex scenes.

*breyloun@gmail.com
†bittner@fel.cvut.cz

2 Related work

There are many methods to achieve global illumination, Ritschel did a nice overview [7] of them.

Most of the fast real-time systems (30 frames per second and more) use a rasterization approach to lighting: they approximate light by a few fast vector operations, possibly using a shadow map (this is a very crude simplification) and everything is computed on a GPU in matter of milliseconds. Some GI approximations can be found even here: using a constant value everywhere, static lighting baked into a scene, light probes or contact shadow approximation by screen-space ambient occlusion (possibly using its two-layer variant from Mara [5]). None of these approaches are really used in this paper (with a possible exception of generating albedo, normal and velocity buffers using rasterization).

There are other approaches approximating light using more exact methods: Instant radiosity solves only diffuse lighting using simplified geometry, photon mapping works with both rays from the light and a camera which estimates caustics better, voxel cone tracing shots cones instead of rays and samples lower-resolution scene when broader range is needed... Some of these methods are real-time or near real-time, they usually approximate some part of the rendering equation.

Our filter, however, is applied to the path tracing output and should be able to approximate rendering equation much better. It is based on a recent paper from Schied [8] with an extension to use even fewer samples per pixel and a few experimentally found tweaks and simplifications (like a smaller kernel used and omitted spatial variance estimation).

Some similar techniques spawned around recently. Using a machine learning approach by Chaitanya [1] have very good output as well, but they are still much slower using a top of the line GPU (around 50 ms on Titan X Pascal). A very similar paper with comparable results by Mara [6] ignores variance and relies on a more complicated bilateral filters instead. Results of this paper look very comparable to Schied [8], the differences are yet to be determined. Silvennoinen and Lehtinen [9] with interesting results took a bit different approach and relied on automatically generated GI probes: this method is a bit more dependent on scene geometry and it's unclear how it'll behave under motion. Edge-Avoiding À-Trous Wavelet filtering by Dammertz [2] and later modifications by Hanika [3] were used as a part of our algorithm.

3 Paper structure overview

Section 4 is dedicated to the basic algorithm explanation. The section begins with a high-level overview of the algorithm stages. Each stage is then described in its subsection. Section 5 then modifies some parts of the algorithm to be able to accept even less than one sample per pixel.

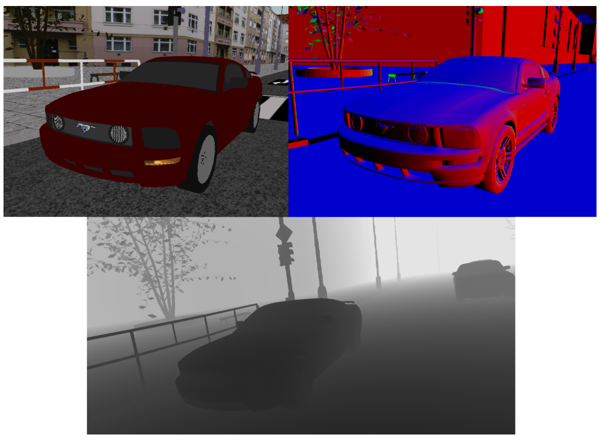


Figure 1: albedo, normal and velocity buffers

Section 6 tells us about the implementation and its results and, finally, section 7 closes the paper by stating method's limitations and possible enhancements in the future.

4 The basic algorithm

Let's assume the basic version of the algorithm: we have one sample per pixel of lighting, normal, depth, albedo and velocity buffer 1 as input (these could come from a path tracer or a rasterizer). We will filter only a diffuse lighting component to simplify our reprojection (described later in Section 4.2). The algorithm can be divided into five consecutive parts:

1. diffuse lighting extraction: diffuse lightig is isolated
2. accumulation: current frame is blended with the last one (without the spatial filter or from the first level)
3. variance smoothing: variance is filtered to alleviate artefacts
4. spatial filtering: some pixels from close neighborhood are blended, determined by constraints
5. blending: the resulting frame is blended with the last one to achieve even less noise and antialiasing

4.1 Diffuse lighting extraction

Diffuse lighting component is extracted from path tracer in some way. Path tracer itself ideally outputs diffuse light separately from the other output. It's possible to work with a path tracer as a black box if we don't want to or can't modify part of it: we can approximately reconstruct a diffuse component by dividing the result by albedo (or using more complex methods). Specular light can now, however, get badly reprojected and/or blurred.

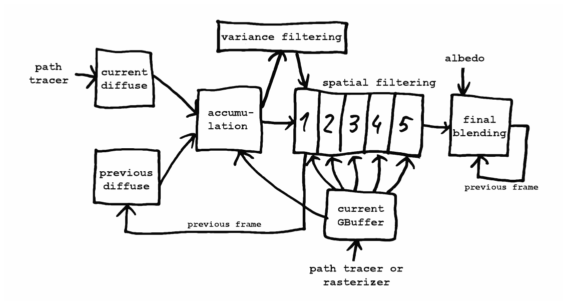


Figure 2: algorithm overview

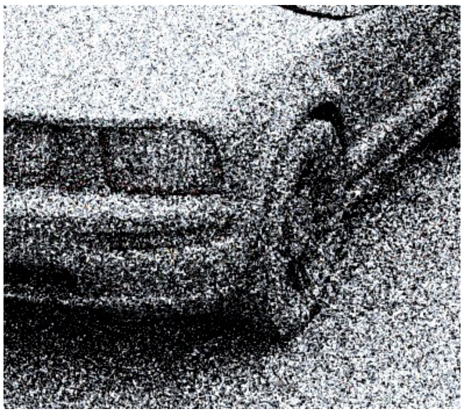


Figure 3: one sample per pixel input

4.2 Accumulation

Current light buffer 3 (path tracing output) is blended with the one from the previous frame. Previous frame (actually the first level of wavelet filtering from the last frame, see Section 4.4) is reprojected into current one using velocity buffer (screen-space offset between last frame and current frame fragment positions in the world). A fragment is discarded on disocclusion, this is determined by a big difference between predicted reprojected depth and actual depth in depth buffer. Only current frame light data is accumulated in a case of a disocclusion. The formula for the normal case is:

$$c = cr + p(1-r)$$

$$r = max(a, \frac{1}{age})$$

where current frame c and previous frame p is blended by a ratio r. age is a number of frames without disocclusion of a current fragment. $r = \frac{1}{age}$ alone would blend current light indefinitely, but any reprojection errors would be visible for a long time. We can start discarding old information at the expense of more noise by enforcing a minimum r by a parameter a. Values around 0.1 proved to be a good compromise.

Path tracing estimates the lighting by a Monte Carlo method: it varies ray directions randomly and that's why there is so much noise in the output when using too few samples. Different parts of the image can have different amount of noise: many of them may be completely lit, some of them might and might not be occluded. That's why the resulting light intensity can have a large variance: this is the variance we are trying to estimate to constraint our denoiser. Light intensities in consecutive frames can be understood as multiple samples, so the variance is acually both variance in time and variance of multiple samples. A term "variance" in this text can be understood more specifically as a variance estimation: we are estimating the true variance from a few frames we can work with.

0.0625	0.125	0.0625
0.125	0.25	0.125
0.0625	0.125	0.0625

Figure 4: 3×3 kernel used multiple times in the algorithm (many possible kernels can be used here)

A variance estimation is computed at this stage as well. We have only limited buffer memory (we can't calculate variance by summing samples from a few last frames) so we will use this formula:

$$var(x) = \sum x^2 - (\sum x)^2$$

We only need to store $\sum x$ and $\sum x^2$ by two scalars in buffer memory, we can blend them in the same way as our light buffer. x in this equation is a luminance of a current pixel: we convert RGB values to luminance before computing variance using a formula:

$$grayscale = 0.299R + 0.587G + 0.114B$$

This saves us valuable memory. Variance computation for each color component is possible as well, but that would help us in a narrow set of conditions (human vision focuses on detail).

4.3 Variance smoothing

Variance obtained in this way can be very noisy and can lead to some artifacts. We can alleviate this problem by blurring variance buffer by a small 3×3 kernel 4: we don't lose much spatial information this way and there are no isolated pixels, therefore results are much better 5.

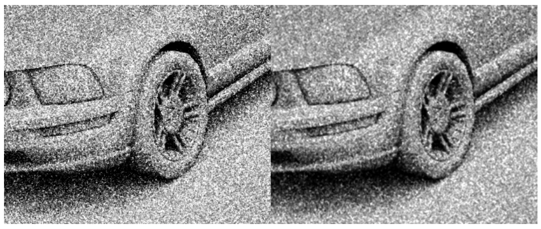


Figure 5: variance smoothing (smoothed on the right)

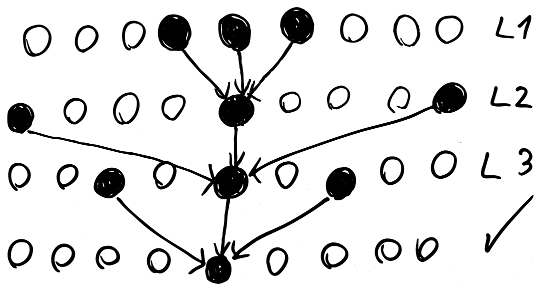


Figure 6: wavelet-based filtering visualization for one pixel and 3 leveld (instead of five), colored pixels have non-zero coefficients and rows are the filtering passes

4.4 Filtering for progressive rendering

We accumulated some samples from a few previous frames and computed their variance. Now we can filter them spatially. We will constrain the blurring by normal, depth and variance buffers from previous steps using a bilateral filter and three similarity weight functions defined later.

We still have a lot of noise in our light buffer; a relatively extensive bilateral filter is needed. Calculating this bilateral filter in a classical way would be very slow, that's why we used a hierarchical approach based on Edge-Avoiding À-Trous Wavelets[2].

This approach filters the input in multiple passes. Each pass uses the same kernel and processes the same, unresized buffer, but with different step size (2^N for the Nth pass, N = 0, 1, 2, etc.) 6. The passes filter large scale first and the small scales last, which filters any high-frequency artifacts caused by the earlier passes [3].

We use 5 filtering passes, each using 3×3 kernel4. Original paper used 5×5 kernel, but we found 3×3 kernel is much faster on lower performance hardware and the quality of results is still ok (6 passes could be used if smoother results are needed). Five passes of 3×3 kernels effectively simulate 61×61 bilateral filter with the a smooth kernel.

The order reversal wasn't in the original paper, and it isn't obvious that variance steering function still works correctly with it, but it does in practice. The output of the first level is used as a previous frame in reprojection phase (see Section 4.2) to gain some additional spatial smoothing of the temporal accumulation: it therefore needs to be done on the smallest scale. This could introduce artifacts in some cases but works well enough in practice 7 (4 more levels hide any residual artifacts).

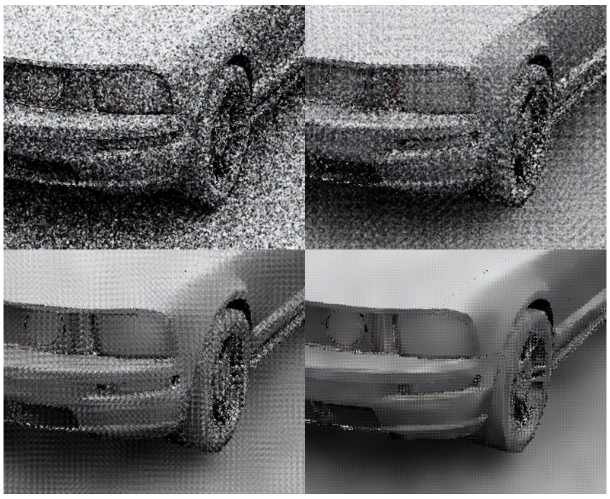


Figure 7: first four wavelet levels, in reading order

4.4.1 Normal weighting function

Normals are weighted by a dot product of their angles. Assuming N_0 to be the first normal and N_1 offset normal being blended:

$$w_N = max(0, N_0 \cdot N_1)^{\sigma_N}$$

Fragments with similar normals are blended this way. Exponent σ_N controls how similar normals need to be to be blended, values around 64 work well.

4.4.2 Depth weighting function

Depth information is a bit harder to weight: depth can differ wildly when sampled in screen space, a simple difference is not satisfactory. We approximate a local slope given by screen-space depth derivatives and define our weight function by a deviation from this slope:

$$w_D = \exp\left(\frac{-|D_0 - D_1|}{|\sigma_D(grad \cdot off)| + \varepsilon}\right)$$

D_0 is the first depth, D_1 is the offset depth being blended into it, grad is the gradient approximated from screen-space derivatives, off is screen-space offset of D_1 from D_0, ε is a small value to avoid division by zero (depends on the scale of depth values used, around 0.005 works fine for a normalized depth buffer) and exponent σ_D controls a threshold of quantization artifacts (leaving at 1 was god enough for us).

This depth weight function works fine in most of the cases except very steep and narrow sides. Very narrow

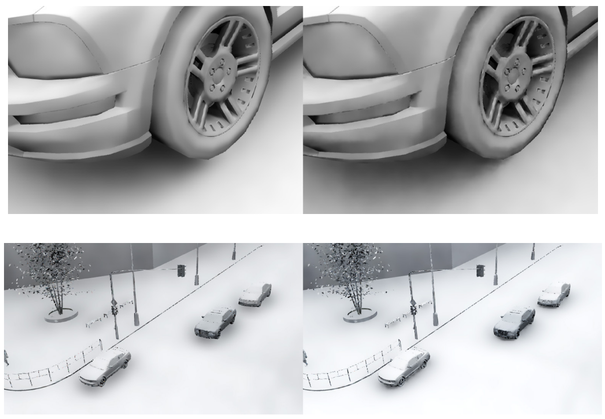


Figure 8: filtering without (left) and with (right) variance, sharp shadows and shadows far away get over blurred

and steep parts of the depth buffer have a very little information to filter through and result in a visible noise. The noise is blended into the following frames and can be noticeable for a long time. That's why we turn off the depth weight function for too steep gradients: normal weight is sufficient in this case.

4.4.3 Luminance weighting function

Light buffer filtered this way would erase all the valuable detail formed during path tracing even in scenes without direct lighting8. That's not what we want, and that's why there is a third, most interesting weighting function. We don't want to blend samples with different luminances, but luminance itself varies wildly in the noisy output. We have variance at our disposal: samples with high variance are noisier and need to be blurred more. This gives us our luminance weighting function:

$$w_L = \exp\left(\frac{-|L_0 - L_1|}{\sigma_L var + \varepsilon}\right)$$

L_0 is the first luminance value, L_1 is the offset value being blended, *var* is our accumulated and prefiltered variance, ε is a small value to avoid division by zero and offset minimal noise (depends on input noisiness and dynamic range, typically somewhere around 0.01 and 0.1) and exponent σ_L controls filtering strictness (4 works well).

Variance approximation can be unusable for a few frames after disocclusion. There are multiple possible ways to handle this: we can turn off the weight entirely or we can approximate spatial variance from the light buffer instead. The former option always blurs the lighting when there is not enough data and the latter can achieve better quality at the cost of additional calculation using e.g. another bilateral filter.

While the variance calculation in the first level of the À-Trous transform is accurate, we don't want to blend the light buffer in other levels as much. We can steer the variance weight function using weight computed in the previous level to avoid blurring light buffer too much:

$$var_{L+1} = w^2 var_L$$

var_{L+1} is a variance one level higher than var_L and w is the complete weight from the last level. All three weights are combined into it like this:

$$w = w_N \cdot w_D \cdot w_L$$

This resulting weight is then used to adjust the À-Trous transform kernel.

4.5 Blending

We have much smaller amounts of noise at this stage, but there's still one big problem: edges in the normal and depth constraints are not aliased and create jagged lines in the output. We can deal with this the usual real-time way: blending with reprojected last frame.

First, we modulate back albedo into the frame and apply tone mapping (if there's any), we will remove jagged lines from the output itself. Then we can reproject the previous frame the same way as we did in the reproject phase, but we will ignore our predetermined disocclusions (they are aliased) and always apply the same ratio r:

$$c = rc + (1 - r)p$$

We will look at the 3×3 neighborhood of current fragment, find the largest and the smallest value and push the previous fragment to this range if it's outside, then we blend them the standard way. A slight indiscernible noise can be added to the output to alleviate any shifting artifacts.

Some more advanced algorithms could be preferred for problematic cases, but this one works fine for most ordinary scenes. It adds one last step to the filtering as well (it filters the changes between frames during the wavelet filtering) and there is hopefully no noise left.

5 Fewer samples

The previously described algorithm works well, converges very quickly (visually around four frames) and handles reprojection well. It's, however, quite often a case that it isn't possible to path trace even single sample per pixel in real-time (that means the quality on 3 is achieved only after multiple frames). The original algorithm was extended so it works as expected even in the case of using less than one sample per pixel.

Missing normal and depth information: It can happen that we no longer have any information about normals or depths in some fragments (in case we're not rasterizing the GBuffer on GPU). We need to approximate missing values somehow.

Simple and efficient way to do this is interpolating values in 3×3 neighborhood hierarchically [10]: average local samples using a 3^N texture pyramid and then propagate them back, filling the holes on the way. This smooths out all edges, and a smarter filter could be used, but it doesn't make that significant difference in the end anyway (normal and depth buffer converge faster than path tracing output).

Reprojection phase: even new samples, not only previous, can have missing values. Values in the light buffer must be ignored when new missing value arrives: light accumulation, reproject ratio and variance all need to stay the same.

Variance filtering: missing values can't be overwritten from neighbors.

Wavelet filtering: missing values need to be skipped during weighing. This assumes at least some samples in every 61×61 rectangle (effective kernel size), but that's usually the case (it would be a very ineffective path tracing anyway).

Aliased edges: There are not enough samples on the object edges to provide any supersampling. Some blurring must be introduced at least initially: FXAA or a similar post process is a good choice.

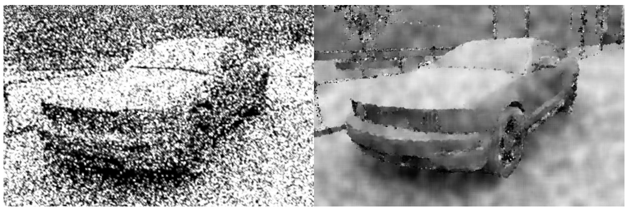
Reusing the first wavelet filtering level for reprojection is an issue with a small number of samples as well: there's blurring happening every step even on previously untouched samples which results in continuous over blurring. This issue doesn't disappear by weakening the wavelet filtering; new samples need special care. One solution is applying the reprojection from first wavelet levels only once full sample per pixel gets accumulated. This, however, introduces a slight smoothing visual skip in the movement as new wavelet smoothing is performed every N frames. That can be alleviated by using an additional buffer always one step behind and blending the real result with this buffer continually, so there is only gradual change.

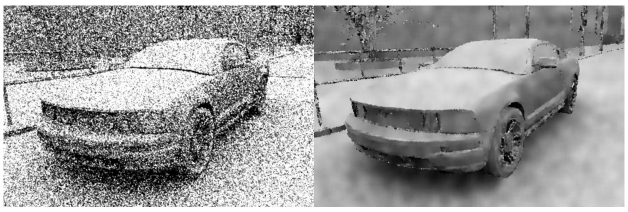
6 Implementation and results

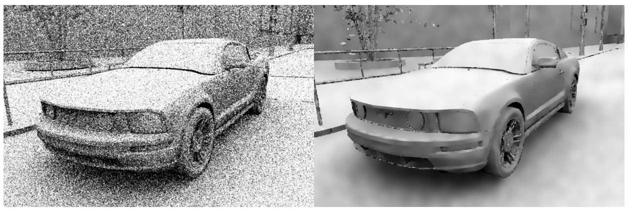
We implemented the algorithm into the existing high-performance CPU renderer in its using OpenGL in its progressive version: it can take about 2 seconds (50-100 frames) on a laptop CPU to generate one sample per pixel. There is more noticeable noise and smoothing during rendering and reprojection, but it quickly converges to the right result as more samples are accumulated.

I used 4-core 8-thread Intel Xeon E3-1240v3 CPU and NVidia 760 GPU when benchmarking. A single view of the car was tested using resolution close to 720p (1366×690). CPU took 20.360 ms (median value) to render one frame ($\frac{1}{128}$ samples per pixel on average) and GPU went trough all the filtering process in 4.666 ms (median value) per frame. CPU and GPU processes could run mostly in parallel.

The algorithm is very stable and practically content independent: shaders do exactly the same work every time.


frame 5, 0.08 seconds


frame 20, 0.32 seconds


frame 88, 1.4 seconds

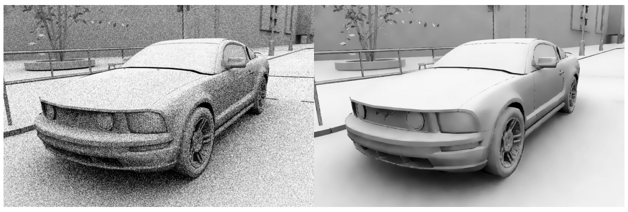

frame 190, 3.16 seconds

Figure 9: Output of CPU renderer (left) and our filter (right) in 720p with $\frac{1}{128}$ samples per pixel per frame

The only exception is reprojection during accumulation, which can cause a large number of cache misses if the reproject misses is too far. The perfomance dip is negligible.

Results from CPU implementation (with uniform sampling and a blur shader) are compared to the output of our filter 9. There are some tonemapping issues and aliased edges: the algorithm is still being developed. There are some advantages of our algorithm that are not visible on this comparison: the solution is stable under motion and new information is blended in on the fly. There is not much detail in the image until the samples start to overlap: accurate variance estimation is the most important requirment for our algorithm. Albedo in primary rays is replaced with white color for clarity.

7 Conclusion and future work

The filtered results converge to an image similar to ground truth after a few frames, with some amount of oversmoothing present. It is obvious that the filter is fast enough for real-time application due to its simplicity (even using not that powerful GPU). Higher performance of the filter compared to the original paper [8] is achieved mainly by using less demanding 3×3 wavelet filter kernel (instead of original 5×5) and ignoring the optional spatial variance estimate (this could enhlance the quality a little bit). The algorithm is practically data-independent and the execution time scales linearly with number of pixels (so it runs around 10 ms at fullHD).

It can significantly filter path tracing in a real-time setting to the point the noise is gone in most of the scene after a just a handful of frames (when using one sample per pixel) and works well even with reprojection. There are some limitations to the current approach, some by design and some can be fixed by a bit different implementation.

The algorithm always needs at least incomplete normals and depth values (or similar metrics) to constrain the edge-avoiding wavelets, so visibility needs to be implicitly known right away and can't be stochastically sampled. But simple effects such as depth of field, motion blur etc. can be easily approximated in screen space. It wouldn't be hard to implement more GBuffer layers for partly transparent content; one more layer might even fit in the same pass using some clever memory management.

The algorithm currently filters diffuse light component only. It partially works with reflections as well, but reprojects them wrongly and blurs them in the process. Another pass with better reflection reprojection (or rejection) would be needed, which isn't complicated to implement. More passes for hard and soft diffuse lighting is also possible (if one prefers noisier input for certain parts of an image), but not necessary; hard shadows are handled well (quicker when variance converges sooner).

Some noise can be visible in large dimly lit areas of the image even after all filtering due to too little data available. This problem could be solved by tweaking the smoothing parameter for dark parts of the picture.

If light moves or part of a scene moves, new parts of the scene can appear slowly, they are blended into the old ones. That's because there is no rejection registered. This can be fixed by allowing previous frames are blending only to the pixels with a similar neighborhood and rejecting them otherwise; special scene-dependent handling seems unnecessary. I am experimenting with this algorithm as my diploma thesis: I would like to implement this feature in it in time, that would make this algorithm genuinely real-time even for dynamic scenes. The other, more straightforward and a bit hacky option is doing more path tracing calculations with a shorter blending time. Error would be less obvious this way.

The algorithm currently handles cases with one or fewer samples per pixel. The straightforward way of implementing more samples per pixel (just blending them) would waste some detail in variance: the correct way would be summing the squared terms both inside one multisample and then between samples to arrive at an accurate variance estimate. I would like to implement this as well if there will be time.

References

[1] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4):98:1–98:12, July 2017.

[2] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch. Edge-avoiding À-trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics*, HPG '10, pages 67–75, Aire-la-Ville, Switzerland, Switzerland, 2010. Eurographics Association.

[3] Johannes Hanika, Holger Dammertz, and Hendrik Lensch. Edge-optimized a-trous wavelets for local contrast enhancement with robust denoising. 30:1879–1886, 09 2011.

[4] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.

[5] M. Mara, M. McGuire, D. Nowrouzezahrai, and D. Luebke. Deep g-buffers for stable global illumination approximation. In *Proceedings of High Performance Graphics*, HPG '16, pages 87–98, Aire-la-Ville, Switzerland, Switzerland, 2016. Eurographics Association.

[6] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz. An efficient denoising algorithm for global illumination. In *Proceedings of High Performance Graphics*, New York, NY, USA, July 2017. ACM.

[7] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. The state of the art in interactive global illumination. *Comput. Graph. Forum*, 31(1):160–188, February 2012.

[8] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*, HPG '17, pages 2:1–2:12, New York, NY, USA, 2017. ACM.

[9] Ari Silvennoinen and Jaakko Lehtinen. Real-time global illumination by precomputed local reconstruction from sparse radiance probes. *ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia)*, 36(6):230:1–230:13, November 2017.

[10] M. Solh and G. AlRegib. Hierarchical hole-filling for depth-based view synthesis in ftv and 3d video. *IEEE Journal of Selected Topics in Signal Processing*, 6(5):495–504, Sept 2012.