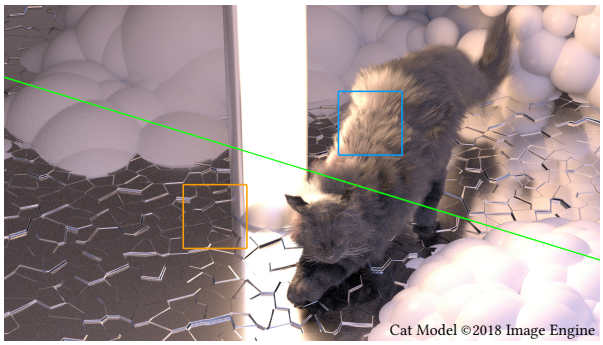
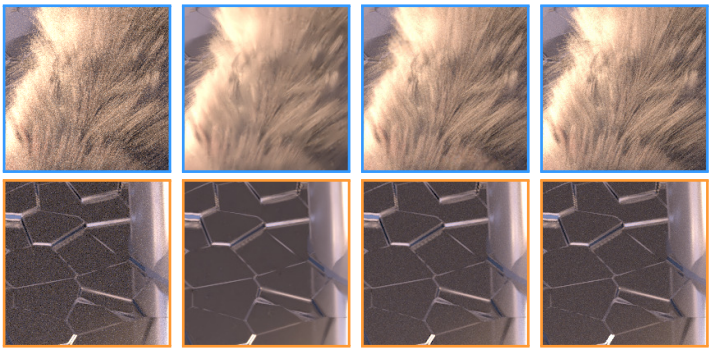
Practical Denoising for VFX Production Using Temporal Blur

Daniel Dresser
Image Engine
danield@image-engine.com



Raw Render (bottom) – Our Method (top)



Raw (36 spp) RenderMan Our Method Ref (144 spp)

Figure 1: A 36 spp render with hair, motion blur, and difficult illumination. The RenderMan denoiser produces very clean results but loses detail in tricky areas such as hair. Our method preserves details and provides a moderate noise reduction, with results similar to a 144 spp reference render. Both denoising methods use data from adjacent frames.

ABSTRACT

We present a simple, efficient, and reliable approach to denoising final ray traced renders during VFX production. Rather than seeking to remove all noise, we combine several simple steps that reduce noise dramatically. Our method has performed well on a wide variety of shows in Image Engine's recent portfolio, including Game of Thrones Season 7, Lost in Space, and Thor: Ragnarok.

CCS CONCEPTS

• **Computing methodologies** → **Ray tracing**;

KEYWORDS

denoising, Monte Carlo rendering

ACM Reference Format:
Daniel Dresser. 2018. Practical Denoising for VFX Production Using Temporal Blur . In *Proceedings of SIGGRAPH '18 Talks.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3214745.3214762

1 INTRODUCTION

Much recent denoising research has focused on removing as much noise as possible. Many papers have demonstrated dramatic results, starting with extremely noisy images rendered with very few samples per pixel (spp), and producing results that are clean, plausible, and appealing [Bitterli et al. 2016; Chaitanya et al. 2017]. Broadly, most of these methods are similar to joint non-local means filtering, blurring together all pixels that match some similarity criteria. This can achieve exceptionally good results, but it is difficult to achieve consistently good results in complex cases. There is risk of overblurring if a criteria is missed or artifacting if the similarity criteria is itself undersampled.

Feature film VFX companies have different requirements for denoising. They are prepared to spend substantial computing resources in order to reliably obtain a high-quality image. We start with ray traced images of reasonable quality — in the case of Image Engine, we render at a minimum of 36 spp for primary visibility and with enough secondary samples to clean up extreme noise from illumination. The denoiser must solve issues such as buzzing or sparkling on very thin objects, tight corners with sharp speculars, or lighting from difficult-to-sample lights. The RenderMan denoiser refers to this as removing the "long tail" of render time — stopping rendering before the point of diminishing returns [Pixar 2017]. Additionally, we do not remove all noise. In VFX, compositors add noise in order to match the characteristics of real cameras. Ideally, a clean render allows us freedom in choosing noise to add, but a small amount of leftover noise is less objectionable than other errors.

In this environment, we noticed that our compositing artists' approach to denoising was just to average together several adjacent frames using optical flow vectors. This provided surprisingly effective denoising on many production shots. Cinematic camera motion often involves minimal screen space motion of the area in focus. Aside from the obvious noise reduction from averaging more samples, noise that stays coherent over a couple of frames may be perceived as small details catching the light, rather than objectionable buzzing.

2 METHOD

We automated and improved the approach used by our compositors. Rather than incorporating information from adjacent frames into a more complex denoising algorithm [Goddard 2014; Pixar 2017],

SIGGRAPH '18 Talks, August 12-16, 2018, Vancouver, BC, Canada

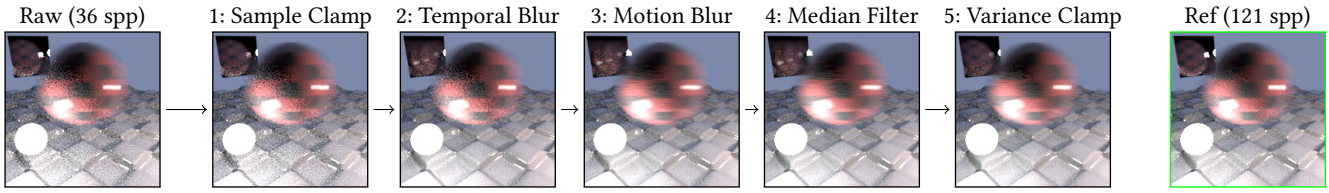


Figure 2: Our 5 steps. Note the errors introduced on the mirror during blurring, which are corrected by the Variance Clamp. For illustration, we start with extreme noise, resulting in some leftover noise and a specular caustic which gets clamped.

we blend per pixel between pure temporal blur, and some simple alternate steps with no dependency on adjacent frames.

Figure 2 demonstrates how our steps work together. Our implementation uses a custom pixel filter, followed by a series of image compositing operations run on sequences of images with adjacent frame data — we do not attempt to handle still images. All channels of both the main image and any supplemental images (arbitrary output variables, or AOVs) are filtered using the same filter kernel, which preserves additive compositing. Our supplemental material discusses a high-quality implementation of these steps.

Step 1 : Preprocess — Sample Clamp During ray tracing we remove intensity outliers that would be difficult to average out. A common approach is to clamp all camera samples to a fixed brightness threshold, but we do slightly better; we use a pixel filter that computes a global weight multiplier for each camera sample based on its total brightness. This preserves additive compositing and reduces clamping of desirable highlights.

Step 2 : Temporal Blur Our most important step is to average 5 frames: the current frame, and 2 frames before and after. We render AOVs for screen-space offsets to the previous and next frame. We sum the offsets forward and backward, and compare to the starting point, which yields an error representing how well an adjacent frame matches the current point. Thresholding this error gives us a weight for each pixel from the adjacent frames. If the sum of weights from adjacent frames is over 2, a pixel is replaced with a weighted average. This error calculation does not catch shading changes on stationary geometry — for example shadows and reflections from moving objects — but we partially address this with the final variance clamp step.

Step 3 : Motion Blur We render a motion vector pass to identify fast-moving pixels, which are not handled by temporal blur. We replace these pixels with the average of a set of samples taken along a short line segment aligned to the motion vector. Adding extra blur aligned with existing motion blur is crude, but generally not visually objectionable.

Step 4 : Median Filter At this point, almost all pixels have been denoised, but a few exceptional pixels will fall in between the categories handled previously. For pixels that were not affected by the previous two steps we use a median filter variant, which removes the worst noise. It blurs out some detail, but in practice very few pixels are affected.

Step 5 : Variance Clamp Finally, we mask the previous 3 steps to only affect pixels that are actually noisy. The loss of quality from the previous steps is generally low, but there could be some unnecessary blur or artifacts. We identify noisy areas of the image with an AOV which estimates the variance of each pixel as a whole, based on the variance of the subpixel samples. The final pixel value is the result from the previous steps, clamped to the original result of the preprocess, plus or minus this AOV. This results in blurring only where necessary to remove noise.

3 CONCLUSION

The biggest difference between our method and most recent denoising methods is that for any output pixel, the set of source pixels we consider is quite small. Many denoising techniques consider a large number of possible source pixels, narrowing down this large pool of information using similarity criteria. This allows for dramatic denoising but also for significant errors. By considering a small number of possible source pixels, we limit how much noise we can remove, but also the errors that can be introduced. The behaviors of median filters, motion blur, and frame averaging are all well-understood, so worst-case errors are usually small and predictable.

This compromise has worked well for us in production. Our method has been used without modification in projects such as Game of Thrones Season 7, Lost In Space, and Thor: Ragnarok (see supplemental video). It has minimal requirements: only 4 AOVs (variance, motion vectors, and 2 frame offsets) compared to the Renderman denoiser's 13. It does not require expensive computation: for the 1080p sequence in Figure 1, an unoptimized implementation of our denoiser takes 10 seconds per frame on an 8-core Xeon, compared to 100 seconds for the the RenderMan denoiser. Without a denoiser, our default camera sampling was 64 spp, but difficult scenes could require 144 or even 324 spp. With the denoiser, almost all scenes can be rendered acceptably at 36–64 spp.

REFERENCES

Benedikt Bitterli, Fabrice Rousselle, Bochang Moon, José A. Iglesias-Guitián, David Adler, Kenny Mitchell, Wojciech Jarosz, and Jan Novák. 2016. Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings. *Computer Graphics Forum* 35, 4 (2016), 107–117. https://doi.org/10.1111/cgf.12954

Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. 2017. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Trans. Graph.* 36, 4, Article 98 (July 2017), 12 pages. https://doi.org/10.1145/3072959.3073601

Luke Goddard. 2014. Silencing the Noise on Elysium. In *ACM SIGGRAPH 2014 Talks (SIGGRAPH '14)*. ACM, New York, NY, USA, Article 38, 1 pages. https://doi.org/10.1145/2614106.2614116

Pixar Animation Studios. 2017. RenderMan : Denoise Workflow. (2017). https://rmanwiki.pixar.com/display/REN/Denoise+Workflow

Supplemental Material

Practical Denoising for VFX Production Using Temporal Blur

Daniel Dresser
Image Engine
danield@image-engine.com

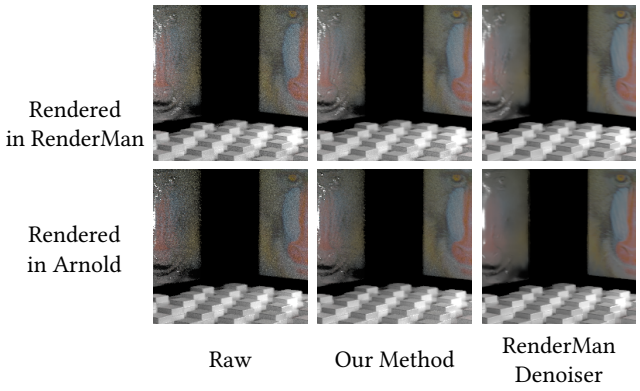


Figure 1: A small crop of our calibration scene, featuring texture, bump maps, reflections, and camera motion. It was rendered in both the Arnold and RenderMan renderers, and denoised using both our method and the RenderMan denoiser.

1 COMPARISON METHODOLOGY

We wanted to compare to a commercially available technique that is currently available for production use. We selected as the best option the denoiser included with RenderMan [Pixar 2018], which is widely available as part of a popular production renderer and appears to give results similar to some recent literature.

The initial implementation of our method was for the Arnold renderer, but the RenderMan denoiser is intended to work with the RenderMan renderer. In order to get comparable results, we set up a calibration scene where we created matching results in the two renderers, with all AOVs necessary to run both our method and the RenderMan denoiser (see Figure 1).

In both cases, this produced results consistent with our expectations. The RenderMan denoiser produces cleaner results (especially when high frequency details are correlated with the albedo), but in areas with insufficient information, it blurs excessively. The bump mapped mirror was chosen to highlight a weakness of using correlation with feature AOVs to drive denoising — a high frequency bump map on a shiny surface produces high frequency patterns with little correlation to the small differences in normal.

The cat image in the paper header was rendered in Arnold using the same method and denoised using both our method and the RenderMan denoiser. To avoid issues documented for the RenderMan denoiser when sharing samples between pixels, all images for this paper were rendered with a 1 pixel box filter.

The RenderMan denoiser was used with default settings and the default strength of 0.5. Modifying the default strength did not improve results, and there does not appear to be any obvious way to tune it to leave behind some noise (which would make it more directly comparable to our results). The results indicate that the RenderMan denoiser is a good option if you have still images or want totally clean results, but that our method is competitive when dealing with sequences of images in a VFX context.

2 IMPLEMENTATION DETAILS

Our current post-process implementation is a network of image compositing nodes in Gaffer [Haddon and Image Engine 2018]. It consists of standard image operations and could be set up in any image manipulation software. For a reference implementation, see http://gafferhq.org/resources/siggraph2018denoiser.

Our method requires some specific data from the renderer. We implemented two custom pixel filters for Arnold, for performing sample clamping and for outputting variance. We also extended our render exporter to include necessary information for outputting accurate frame offset AOVs.

2.1 Sample Clamp

We start from standard subsample clamping, as implemented by renderer features such as Arnold's `aa_sample_clamp` [Solid Angle 2018]. With a default pixel filter, the result for a particular pixel p in channel c is:

$$p_c = \sum_i w_i s_{c_i} \tag{1}$$

where s_{c_i} are a set of samples for each channel c and subpixel index i, and w_i are a set of weights for each subpixel determined by a pixel filter kernel, with $\sum_i w_i = 1$. The channels includes the main RGB image and also breakdowns of this total result into AOVs, such as just specular or diffuse results, or splitting into light groups.

Standard sample clamping introduces a threshold value t, which is set to some value bright enough to appear fully saturated. A common value is $t = 10$. The pixel value is then:

$$p_c = \sum_i w_i \min(s_{c_i}, t) \tag{2}$$

This will prevent any single badly sampled outlier from pushing the pixel above the threshold, but it will also clamp pixels where the true average is over this threshold. It also breaks "additive compositing" — compositing artists expect to be able to perform operations such as taking an AOV containing all specular light paths

SIGGRAPH '18 Talks, August 12-16, 2018, Vancouver, BC, Canada

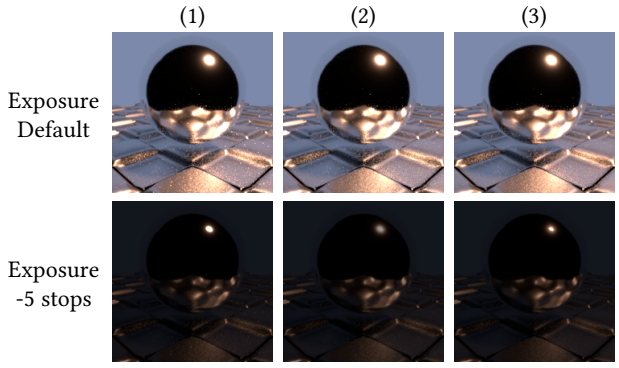


Figure 2: Pixel filtering using equations (1), (2), and (3). Note that both (2) and (3) remove "fireflies", but (3) does not crush the highlight as much and avoids tinting towards grey.

and subtracting it from the total image to get all non-specular light paths. This subtractive relationship is violated by a basic clamp.

Our improved sample clamping instead computes a new set of weights that achieve a similar effect but are applied as the same multiplier for all channels of a subpixel. The new weights are computed based on the value of the main RGB channels, (s_r, s_g, s_b):

$$v_i = w_i \max(1, \frac{t}{\max(s_{r_i}, s_{g_i}, s_{b_i})})$$

We then normalize these weights and compute the pixel values for all channels like this (note that by using the same weights for all channels, we preserve additivity):

$$p_c = \sum_i \frac{v_i}{\sum_j v_j} s_{c_i} \quad (3)$$

Note that due to the normalization, the pixel value p_c can exceed the threshold t if many subsamples all exceed the threshold. It is actually desirable that this threshold is no longer an absolute limit. If many subsamples are all bright, then the pixel represents an actual highlight, not a sampling error, and it should be allowed to exceed the threshold. This gives us the nicer falloff around the hottest part of the specular seen in Figure 2. This was a nice side effect of our original goal to preserve additive compositing.

We implemented this filtering as a custom pixel filter for the Arnold renderer. This is a small improvement over standard sample clamping, but it is also very easy to implement, so it would likely make sense for renderer implementers to include it as an option.

2.2 Temporal Blur

In order to average together 5 frames, we just need accurate offsets for each frame to the previous and next frame. This requires extra data to be available during rendering, to output two extra AOVs. Using Gaffer [Haddon and Image Engine 2018] to set up our renders allows us to insert a set of extra scene manipulation nodes at the end of our render setup scripts. These nodes automatically compute extra information to be passed to the renderer. This includes extra primitive variables and scene hierarchy attributes for the world space positions of deforming and translating objects in adjacent frames. Together with camera information from adjacent frames, these can be used to output precise offsets to screen space positions on adjacent frames.

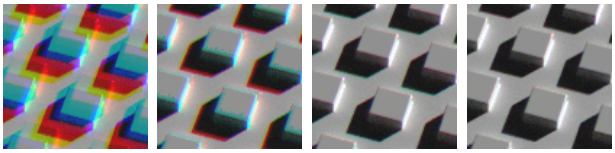


Figure 3: Superimposed in the red, green, and blue channels are frames 10, 12, and 14 of an animated sequence with camera movement and shake. Without alignment, the features shift substantially in these 3 frames. The forward and backward vectors computed by RenderMan for Maya approximately align features. Our method produces close alignment (but with a subpixel offset fringe) by sampling once per frame or exact alignment with 3 samples across the shutter.

The accuracy of these offsets is quite important because any small offsets could cause high frequency details to "wobble". The simplest way to compute these offsets would be taking the difference in position from the center of the shutter time for one frame to the center of the shutter time for the next. This is correct for objects moving in a straight line, but is not accurate enough during sudden direction changes.

The approximate position of a pixel on a previous frame is a small blurred region, and the position at the center time of the shutter may not be in the center of the blur. We found this discrepancy was sufficient to cause small texture details to be positioned incorrectly by up to a pixel. While a small amount of extra blur is not noticeable, this varying offset in the position of details can be objectionable.

To fix this, instead of comparing one sample at the center of the shutter, we take 3 samples for each position for each frame: at shutter open, shutter middle, and shutter. We weight these samples with the weights $0.25, 0.5, 0.25$. This is the exact value of the average position when rendering with two motion segments per frame and matches fairly closely when using > 2 motion segments.

Figure 3 shows the effect of using more accurate offsets to align 3 frames. In this example, a single sample in the center of the shutter gives a fairly good alignment, but in order to get subpixel accuracy, it is necessary to take 3 samples over the shutter. The offsets computed by RenderMan for Maya provide approximate alignment, but are not precise enough to be useful in our method. The RenderMan denoiser [Pixar 2018] is much less reliant on offset vectors, but by replacing these vectors with the ones we compute, we did observe slightly better results from the RenderMan denoiser (at least in sequences with some camera shake). It seems like it could be valuable for RenderMan for Maya to output more accurate offsets, both to improve the accuracy of their denoiser and to allow RenderMan for Maya users to employ simple temporal blur.

2.3 Motion Blur

Our current implementation of motion blur is incredibly simple. For each pixel, we compute a vector that is 50% of the motion vector for this pixel over the frame, clamped to be no more than 4 pixels long. We then take 10 samples arranged in a line along this

vector, centred on the pixel. A more sophisticated approach which attempts to resolve overlapping blurred objects and visibility occlusion is likely possible, but we found that any noticeable artifacts are usually cleaned up by the final variance clamp step, and a small amount of extra blur in motion blurred areas has not been an issue.

2.4 Median Filter

The specifics of the median filter step are not significant to the overall algorithm, since the majority of pixels are handled by the previous two steps. If additive compositing is not a concern, a standard 1 pixel median filter performs well.

We value preserving additive relationships between AOVs, so we must use the same filter kernel for all images — but a standard median filter would pick different source pixels for every channel. For this reason, we use a special median filter that chooses a source pixel based solely on luminance and then uses the same source pixel for all channels. This removes any severe luminance noise but may create some hue noise. We add a 1 pixel blur which corrects this, at the cost of some extra blur in pixels processed by this step.

2.5 Variance Clamp

Once we have a variance AOV, this step is extremely straightforward. The result for each pixel is:

$$\min(\max(d, o - 1.5\sqrt{\sigma^2}), o + 1.5\sqrt{\sigma^2})$$

where d is the denoised result after all filtering, o is the original result from the renderer (with only sample clamping applied), and σ^2 is the variance. The choice to clamp to 3 standard deviations was based on empirical testing but is intuitively reasonable — for hypothetical Gaussian distributed noise, this would allow us to remove 90% of noise. We only store variance for the 3 main RGB channels, and additional AOVs are filtered to match by computing a weighting between the denoised and original images that gives the same result as clamping the main image.

Computing variance is a bit tricky. We want a measure of how much overall error we think there is in the filtered pixel. As above, let s_{c_i} be subsample values for each channel, and w_i be the corresponding weights of the pixel filter, then the pixel value is $p_c = \sum_i w_i s_{c_i}$. Defining σ_c^2 as the variance of the filtered pixel for each channel gives us:

$$\sigma_c^2 = \mathrm{Var}(p_c) = \mathrm{Var}(\sum_i w_i s_{c_i})$$

If we make the fairly reasonable assumption that the values of s_{c_i} are independent, we get:

$$\mathrm{Var}(\sum_i w_i s_{c_i}) \approx \sum_i w_i^2 \mathrm{Var}(s_{c_i})$$

This appears difficult to accurately estimate. If we assumed that there is a single distribution for s_c which s_{c_i} all come from, there is a natural answer, but this does not perform well in practice. It puts too much importance on the variance of low weighted outliers on the edge of a pixel filter, which may contain features which do not actually influence the more heavily weighted samples.

If we don't assume the same distribution, we have only a single sample for each s_{c_i}, which is not enough to estimate variance. We therefore make a simplifying assumption: for the purposes of computing variance, we assume that the mean of every s_{c_i} is the filtered value of the pixel p_c (but that their variances differ). This is not true of real images where features partially cover a pixel, but yields reasonable results in practice. This gives us a single sample estimator of variance:

$$\mathrm{Var}(s_{c_i}) \approx (s_{c_i} - p_c)^2$$

Using this to estimate total pixel variance yields:

$$\mathrm{Var}(\sum_i w_i s_{c_i}) \approx \sum_i w_i^2 (s_{c_i} - p_c)^2$$

Note that this value differs from a standard weighted sample variance (assuming the samples come from the same distribution) only in that the weights are squared. Setting $\sigma_c^2 = \sum_i w_i^2 (s_{c_i} - p_c)^2$ gives us an AOV that is a nice intuitive representation of the amount of noise in an image. Any fireflies in the main image are visible in the variance AOV with matching filtering, and as you increase sampling, the noisiness drops towards 0.

If all values of w_i are the same (i.e. a box filter), this simplifies to just the mean squared error divided by the number of samples. This is the same as the "mse" value output by RenderMan [Pixar 2018]. When w_i differ, it can be viewed as a weighted generalization of mean squared error. In brief empirical testing, this variance result appears to perform well not only with our method, but also when substituted in as the "mse" input to the RenderMan denoiser (even when using filters with varying w_i).

As in Section 2.1, we also implement this variance output as a custom pixel filter for the Arnold renderer. Note that the weights we use for computing variance are the modified weights computed in Section 2.1, so we don't overestimate the amount of noise contributed by fireflies that get suppressed by clamping.

3 FUTURE WORK

Because our denoising time is currently insignificant compared to ray tracing, we have not put effort into optimizing our implementation. A GPU implementation could likely run close to realtime.

The most significant failure case for our method is when adjacent frames contain objects in the same position, but with dramatically different shading or lighting. This could be due to moving shadows, reflections, and lights, or animated shader properties. The final variance clamp will fix artifacts that happen in clean areas, but if these artifacts occur in areas that also contain substantial noise, ghosting artifacts may be visible in the final result.

This has not been an issue in the majority of production shots, but it does prevent using our method in some cases (particularly sequences lit by flashing lights).

We currently have no check for statistical similarity of the neighbourhood of a pixel in the current and adjacent images. Adding a simple check based on variance will hopefully avoid incorrectly using temporal blur when adjacent images are not similar. This would reduce the cases where our method cannot be used.

REFERENCES

John Haddon and Image Engine. 2018. Gaffer: An application framework for visual effects production. (2018). http://www.gafferhq.org/

Pixar Animation Studios. 2018. RenderMan : Users' Guide : RIS : Denoising. (2018). https://renderman.pixar.com/resources/RenderMan_20/risDenoise.html

Solid Angle. 2018. Arnold For Maya User Guide : Arnold Render Settings : Clamping. (2018). https://support.solidangle.com/display/A5AFMUG/Clamping