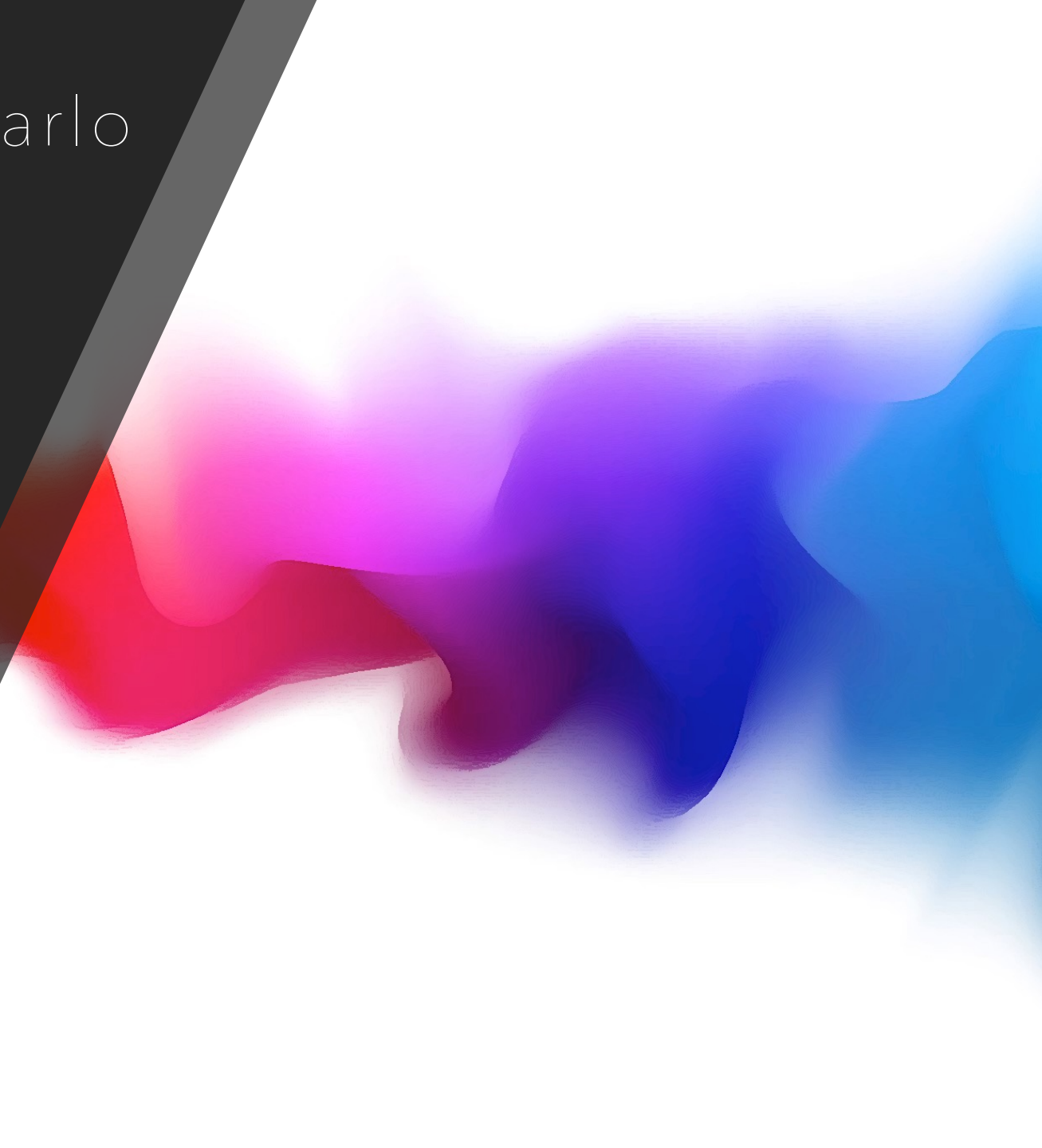


# Denoising Deep Monte Carlo Renderings

Presented by *Maxime Raafat*

*Delio Vicini, David Adler, Jan Novak, Fabrice  
Rousselle, Brent Burley (Disney Research)*

August 27, 2018  
EU Computer Graphics Forum 2018



# Deep Monte Carlo Renderings

Pixels contain multiple colour values for different depth ranges (stored in depth bins)

More expressive representation of the scene than conventional flat images



Kakamora , sequence taken from *Moana*. Disney 2016





Why Deep Images?

# Deep Images

Cheap re-rendering of scenes from novel viewpoints

Easy compositing, no prerequired order (unlike flat images) :  
inserting atmosphere or surfaces, performing lighting  
adjustments to particular depths



# Challenges of Denoising Deep Images

Noise is inevitable (Quasi-Monte Carlo integration convergence rate :  $O(1/N)$ )

All available denoising algorithms operate exclusively on flat data

Denoising a flattened image after compositing yields undesired artefacts

Deep images have an unstructured depth dimension (typically no correspondences between depth bins of neighbouring pixels)

Goal : preserve full expressiveness of deep image structure through denoising, while matching the robustness of filters operating on flattened data



# Denoising Flat Images

# Joint NL-Means Filter

State-of-the-art denoising algorithms : combination of noisy colour and feature information

This work builds on the joint NL-means filter used in previous works

$$\hat{O}_p = \frac{1}{C_p} \sum_{q \in \mathcal{N}_p} w(p, q) O_q$$

$\hat{O}_p$  = filtered value of pixel  $p$

$O_q$  = noisy input value of pixel  $q$

$\mathcal{N}_p$  = square neighbourhood centred on  $p$

$w(p, q)$  = weight of  $q$  contributing to  $p$

$C_p$  = weight normalisation factor



# NL-Means Weights

$$w(p, q) = \min(w_O, w_F)$$

$w_O$  : NL-means weight computed on the data

$w_F$  : cross-bilateral weight computed on a set of auxiliary features (surface albedo, normal and depth)

$$w_O(p, q) = e^{-D_O(O_{\mathcal{P}_p}, O_{\mathcal{P}_q})}$$

$$w_F(p, q) = e^{-D_F(F_p, F_q)}$$

$D_O(O_{\mathcal{P}_p}, O_{\mathcal{P}_q})$  = distance between square patches centred on  $p$  and  $q$ , according to the noisy input values

$D_F(F_p, F_q)$  = distance computed using the auxiliary feature  $f$  that maximizes the dissimilarity between  $p$  and  $q$



# Denoising Deep Images

# Deep Joint NL-Means Filter

The deep filter operates on images where the depth samples in each pixel are clustered into discrete bins. Recall the general joint NL-Means filter

$$\hat{O}_p = \frac{1}{C_p} \sum_{q \in \mathcal{N}_p} w(p, q) O_q$$

$$\hat{O}_{p_b} = \frac{1}{C_{p_b}} \sum_{q \in \mathcal{N}_p} \sum_{q_d \in B_q} w(p_b, q_d) O_{q_d}$$

# Deep Joint NL-Means Filter

We now denoise depth bins instead of pixels, as in the flat case

$\hat{O}_{p_b}$  = denoised colour of bin  $b$  in pixel  $p$

$O_{q_d}$  = input colour of bin  $d$  in pixel  $q$

$\mathcal{N}_p$  = square neighbourhood centred on  $p$

$B_q$  = list of bins of pixel  $q$

$w(p_b, q_d)$  = weight of bin  $q_d$  contributing to bin  $p_b$

$C_{p_b}$  = weight normalisation factor

$$\hat{O}_{p_b} = \frac{1}{C_{p_b}} \sum_{q \in \mathcal{N}_p} \sum_{q_d \in B_q} w(p_b, q_d) O_{q_d}$$

# Deep NL-Means Weights


$$w(p_b, q_d) = \min(w_O, w_F)$$

$w_O$  : NL-means weight computed on the data

$w_F$  : cross-bilateral weight computed on a set of auxiliary features (as before)

$$w_O(p_b, q_d) = w_O(p, q) \cdot w_\alpha(q_d)$$

$w_\alpha$  : effective alpha of neighbour bin  $q_d$  that measures its relative contribution to pixel  $q$



# Results & Outlook



# Implementation Details

The open-source Mitsuba renderer was modified to output deep images

We first do a **depth-only rendering pre-pass**, in which the depth and alpha value for each bin of every pixel is calculated

The pre-pass allows us to pre-allocate a fixed-size frame buffer and avoids the need to capture and sort all samples

We then **render the deep images** with the same random seed

The deep filter is implemented on CPU using a mixture of Python and C++

Performance measured on an intel i7-4930K processor, running 6 cores (12 threads) at 3.4Ghz

# Performance

Partially multithreaded filtering ~12 minutes per megapixel

Structure preservation of the input data is maintained with the deep filter



Noisy



Deep

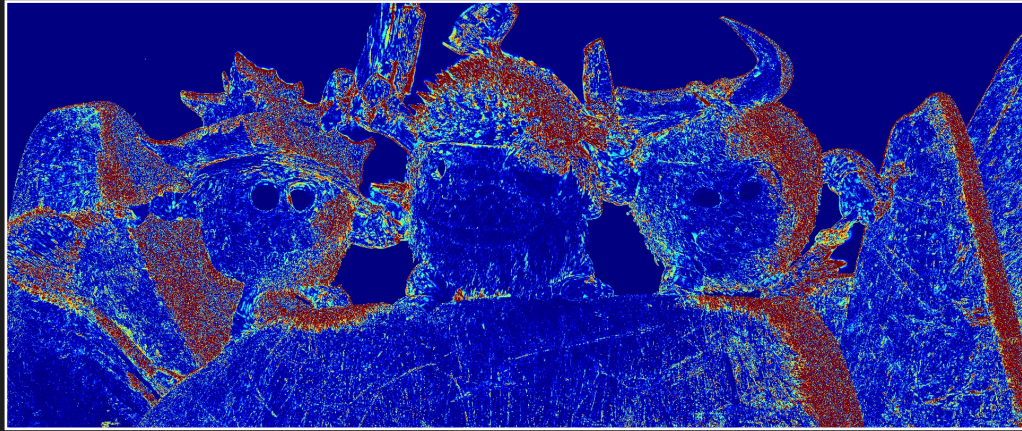


Reference

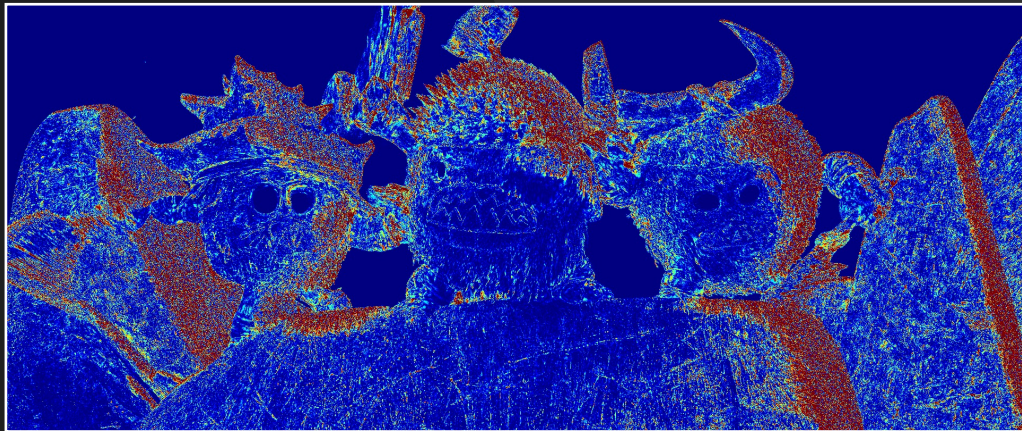




# Performance (Mean Square Error)



Flat (RDFC) MSE



Deep (Ours) MSE



Noisy MSE  
4.9134

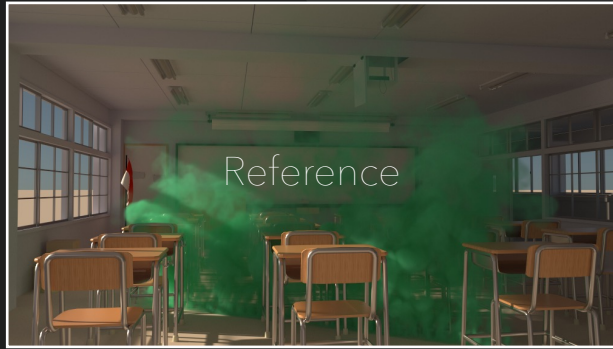


Deep MSE  
0.6875

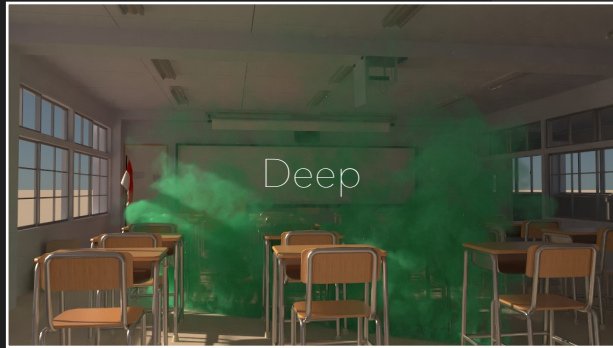


Flat MSE  
0.6852

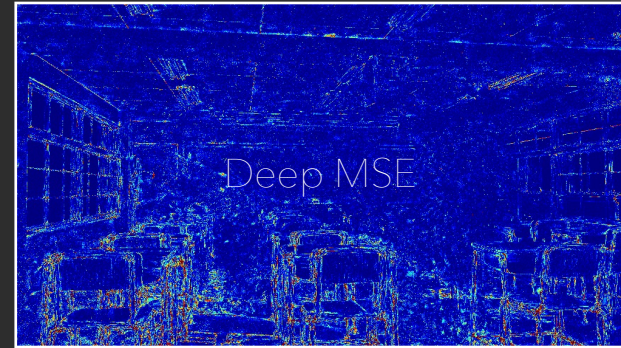




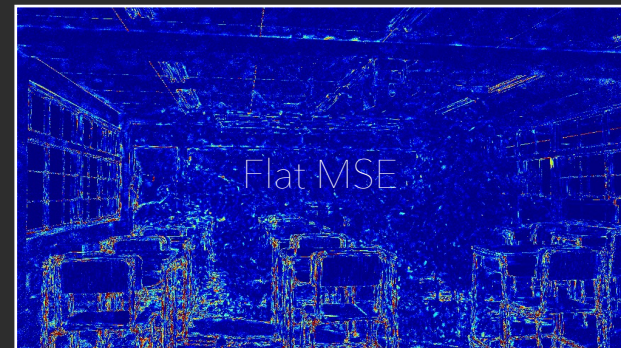
Noisy MSE  
2.0569



Deep MSE  
0.1316



Flat MSE  
0.1293



# Limitations and Future Work

**Computational overhead** : the deep filter sums over a varying number of bins, depending on the scene depth at each pixel (unlike the flat case for which the neighbourhood size is fixed)

Clever bounding of the neighbourhood size or adaptively fine-tuning the sampling scheme could improve runtime

**Noisy alphas** : the alpha values computed in the pre-pass are not denoised by the deep filter, which might harm the denoised image by assigning pixels too high or too low values

If denoised, the alpha values will yield no sharp compositing boundaries anymore



Questions?

