UCLouvain

École polytechnique de Louvain

Real-time Path-Tracing Denoising

Author: **Vincent Higginson**
Supervisor: **Christophe De Vleeschouwer**
Readers: **Christophe De Vleeschouwer, Antoine Legrand, Benoît Macq**
Academic year 2022–2023
Master [120] in Computer Science

Abstract

Path-Tracing techniques are at the heart of many tools and frameworks used to generate images, thereby enabling unified physically-based rendering pipelines. But given the current computing power and real-time requirements, the result is often noisy, due to the low number of samples evaluated. A new pass, which we will define as denoising, must be developed to allow the production of a high-quality image. This master thesis will evaluate and compare different techniques based on deep learning, filtering and regression. With predefined metrics, we consider both their effectiveness and computational efficiency. Additionally, this work will explore the integration of these denoising techniques in a more general rendering engine, identifying the dependencies required for their effective performance.

Special Thanks

Jake Ryan for his invaluable help regarding some parts of the temporal accumulation pass. Hugo Devillers, for giving me the first spark of this all-consuming passion. My family and their patience, in times when nothing was going right. My friends, for listening to my rants about computer graphics in the most awkward moments. Clemence and Marie-Astrid, for their grammatical insights.

Contents

Chapter 0

Introduction

Nowadays, path tracing is mainly used in the entertainment industry to generate visually appealing images. Thanks to its unified rendering pipeline of physically based materials, it can work for a wide range of different experiences such as games, movies and scientific interactions. However, path-tracing rendering comes with a larger computational cost than traditional rasterization methods. The power needed to ensure a perfect image, a noise-free image, at a real-time rate is simply inexistent for today's hardware. Some shortcuts have to be made to ensure that the frame to display is ready under strict time constraints. One of the major differences between offline and real-time applications is the number of samples per pixel evaluated. Namely, while offline applications can afford thousand of samples per pixel, the current situation for interactive experience is more stressful. As we'll see in the next section, only one sample (1spp) can be evaluated per pixel, thus generating noisy images.

So, to still benefit from the advantages of the unified path-tracing pipeline and the visual realism it can give, one must mitigate the noise to a level that fits the requirements of the end-user. The goal of this work is to develop a renderer capable of providing a real-time experience to the end user, by specifying each step needed to ensure the end image still benefits from the inner advantage of path-tracing, while being rendered at an interactive, or better, real-time rate. In our case, experimentations demonstrate that results are accurately reconstructed from 1spp image. Moreover, techniques are also temporally stable throughout multiple consecutive frames. A thorough comparison between four different denoising methods is presented, alongside their advantages and caveats.

In the first chapter, the whole pipeline of our renderer is presented with an emphasis on each step that comes before the denoising itself. The rendering equation driving the image generation is briefly explained, allowing the reader to understand precisely where the noise comes from. After that, we develop the intuition and the

limitation of temporal accumulation that helps to start the reconstruction of the image. Alongside the description of the features that are retrieved from the scene, the denoising can start.

The first part of the second chapter is a study of three different denoising techniques that follow different approaches to reconstruct the end image. They are explained in-depth with a strong focus on the mathematical parts. Our implementation is largely based on what is explained in this chapter. And the last part of the second chapter is the initiation of a novel method based on cellular automata, the potential advantages of the new method are explained as well as the possible improvements that could be made to put its performance at the state of the art level.

The third chapter explores the results produced by the proposed methods. With objective metrics, they are challenged against diverse 3D scenes and the parameters that were explained in the second chapter are explored. Rendered scenes are different and exploit crucial facets of illumination, shadows, and reflection to ensure that the methods presented can effectively be used in a general-purpose path-tracing renderer.

Chapter 1

General pipeline

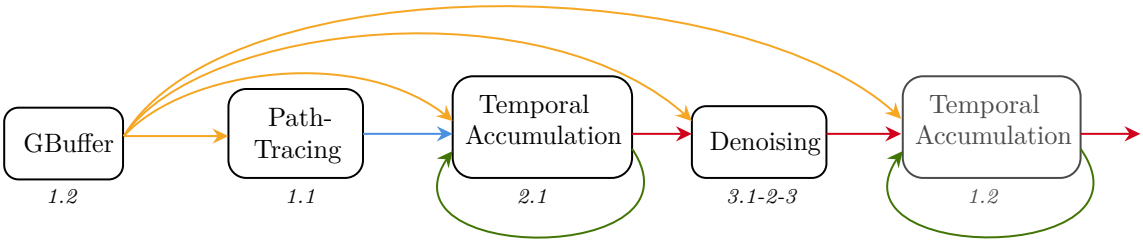


Figure 1.1

A denoiser itself is not the sole component inside a real-time path-tracing renderer. In this section, some steps that have to be taken before, and after denoising are explained to better understand the different takes regarding the implementation of said methods or decisions regarding their parameters.

In section 1.2.1, we discuss the inner content of the G-Buffer, which allows the casting of different rays in section 1.5, it also allows to generate a first image with the support of the rendering equation in 1.2. Before the actual denoising of the produced image, a first temporal accumulation happens in 1.3. Concerning the denoising itself, the subject of our work, it happens only after these 3 steps and is followed by an optional second temporal accumulation step.

1.1 The Rendering Equation

To better understand how to correct the noise of an image generated by a ray-tracing renderer, it is necessary to comprehend the nature of the renderer, and its

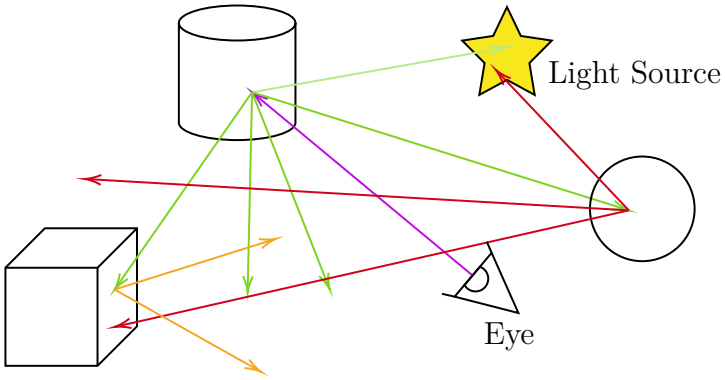


Figure 1.2: A typical path-tracing scenario, involving ray-casting.

underlying principle that resides in its mathematical origin. At the time of writing, most renderers that aim at visual fidelity are using the rendering equation[16] that is solved using the Monte Carlo method. This first chapter aims to have a brief understanding of such concepts, and why they are related to the undesired noise.

1.1.1 Definition

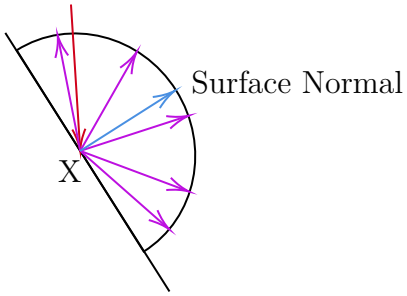


Figure 1.3: Ω is the unit hemisphere containing all possible directions at point X with normal $\hat{n}$

Following the notation from[9], the rendering equation can be expressed as:

$$L_0(X, \hat{\omega}_0) = L_e(X, \hat{\omega}_0) + \left[\int_\Omega L_i(X, \hat{\omega}_i) f(X, \hat{\omega}_i, \hat{\omega}_0) |\hat{\omega}_i \cdot \hat{n}| \, d\hat{\omega}_i\right] \tag{1.1}$$

where:

X	is a point in the scene
$\hat{\omega}_0$	is an outgoing direction
Ω	is the union of all possible directions $\hat{\omega}_i$ around the surface normal $\hat{n}$, the unit hemisphere at that point, see 1.3
$\hat{n}$	is a surface normal
$\hat{\omega}_i$	is an incoming direction

$L_0(X, \hat{\omega}_0)$	is the intensity of light passing from a point X with direction $\hat{\omega}_0$, the *outgoing light* (the one hitting the eye)
$L_e(X, \hat{\omega}_0)$	is the intensity of emitted light from a point X with direction $\hat{\omega}_0$
$L_i(X, \hat{\omega}_i)$	is the intensity of incoming light passing from a point X with a direction $\hat{\omega}_i$, the *incoming light* (the contributing light hitting the point X)
$f(X, \hat{\omega}_i, \hat{\omega}_0)$	is the bidirectional reflectance distribution function (BRDF) describing the proportion of reflected light from $\hat{\omega}_i$ to $\hat{\omega}_0$ at point X, it is often called the material function
$\lvert\hat{\omega}_i \cdot \hat{n}\rvert$	is the geometric term, a weakening factor due to incident angle

The bidirectional reflectance distribution function (BRDF) described the proportion of reflected light given a point, an incoming direction and an outgoing direction[27]. Therefore, its underlying definition depends on the nature of the medium, see 1.4.

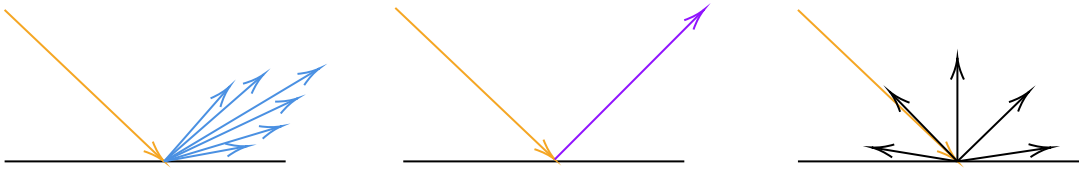


Figure 1.4: Example of the proportion of reflected light per type of medium.

1.1.2 Solving the equation

Solving this equation allows obtaining the intensity of energy at a given point ($L_0(X, \hat{w}_0)$) with a given direction towards the eye. It is worth noting that the function is recursive, as the intensity of light passing at a given point is function of the intensity of light passing at other points reachable with a direction $\hat{w}_i \in \Omega$. These intensities coming from other points are also computed using the rendering equation. A typical situation may help the reader understand the equation as depicted in figure 1.2.

A naïve attempt at solving the equation may result in taking into account all

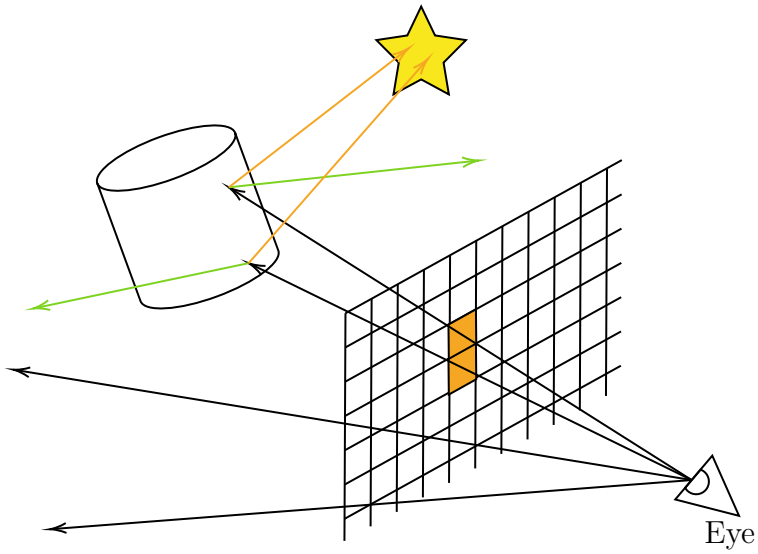


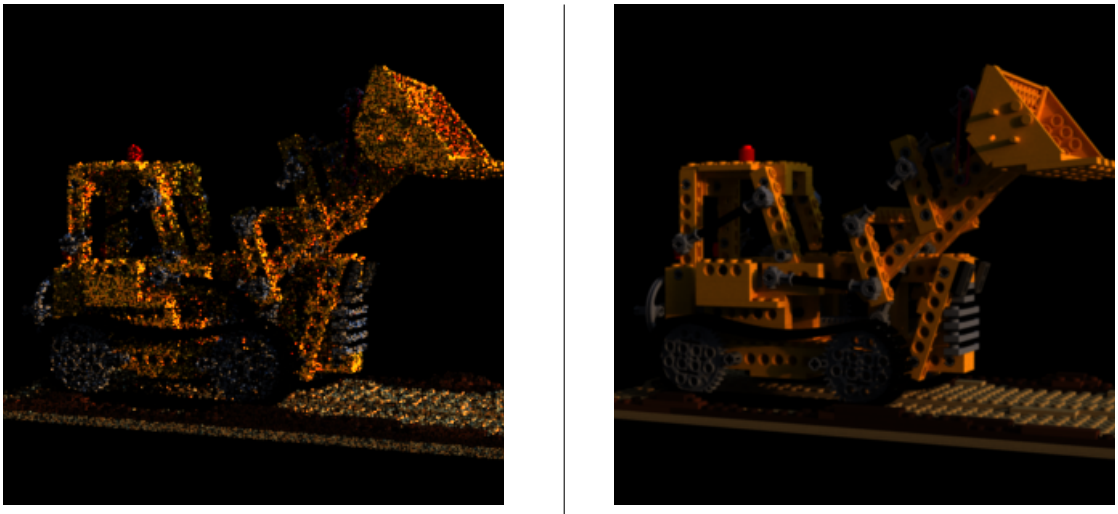
Figure 1.5: Primary ray in **black**. And two subsequent secondary rays in **green**, and **orange**.

possible directions in the unit hemisphere, which is computationally prohibitive. Instead of relying on the whole set of incoming directions, only a specific number of them are considered. The Monte Carlo method allows approximating the result of the rendering equation by picking random samples of the integral[10]. By increasing the number of random samples used to solve the integral for a given point, the approximated solution is expected to eventually converge.

1.1.3 Appearance of noise

The purpose of the described rendering engine is to generate a color image by retrieving the energy emitted by the scene on the camera plane. Real-time renderers often limit the number of cast rays drastically to ensure low computation time. Thus, reducing the number of effective samples of the rendering equation, limiting the quality of the produced result. As described in[19],[29], the approach is often the following:

- One **primary** ray is cast per pixel from the eye of the scene.
- One **secondary** ray, called the shadow ray, is cast from the intersection to a random point of a random light source.
- One **secondary** ray is cast from the intersection point with a random direction.



(a) 1spp (b) 1024spp

Figure 1.6: The image on the left is noisy due to the extremely low number of samples.
3D model published by Heinzelnisse on `https://www.blendswap.com/blend/11490` with the CC-BY-NY license.

Pixels from the generated image have a high variance due to the random nature of the ray casting, and the low sampling of the original rendering equation integral. This high variance is called noise, and that is what we aim to correct in this work.

How the primary rays are generated is related to the inner knowledge of the rendered scene. Section 1.2.1 describes the tight relation between the scene and the generated rays before they interact with the scene.

1.2 Ray generation and Feature buffers

As said before, a typical renderer must first cast a series of primary rays that originate from the eye, effectively traversing each pixel, to determine the intersection within the scene. Due to the complexity of such a task, and to keep the real-time nature of the renderer which involves the limitation of expensive routines, the first intersection is computed using rasterization. Rasterization takes a list of triangles of a specific mesh and outputs a list of highlighted pixels that are effectively contained in original triangles. Nowadays, GPUs are optimized for this workload and performed it in less time than invoking ray intersection routines. This characteristic allows the appearance of a Hybrid Renderer[1] which determines the primary rays and their intersection within the scene using that method. Moreover,

additional information regarding the intersection comes with a little cost. In this section, we define this additional information (which is stored in *feature buffers*) as they'll be needed in subsequent parts of this work.

1.2.1 G-Buffer

The G-Buffer approach is the one used in our implementation, it is comparable to the first occurrence of deferred shading[7]. During the rasterization pass, six textures are generated. Each sample of each texture contains data regarding the nature of the intersection within a given mesh.

1. The *depth buffer*: which contains the distance of the intersection from the eye of the scene.

   $B_{D_x} = \frac{(T \times P)_z}{(T \times P)_w}$, with T being the transform matrix, and P the vertex position.

2. The *normal buffer*: which contains the normal of the plane with which the ray intersected with respect to geometry transform. That means each normal is multiplied by the inverse of the transform matrix.

   $B_{N_{x,y,z}} = (T^T_{1:3,1:3})^{-1} \times N$, with T being the transform matrix, and N the vertex normal.

   Certain materials can be characterized with low roughness, making them almost like a mirror. In this kind of situation, incoming rays are reflected with respect to the surface normal, and intersect with another point of the scene. The surface normal of the newly intersected geometries is used instead of the primary hit point (1.2.1). That allows reflections to be correctly denoised, as we'll see in the next sections. In our implementation, only materials with a roughness < 0.4 behave like this.

3. The *geometry normal buffer*: which contains the normal of the triangle with which the ray intersected. In that case, normals aren't modified, and correspond to the original normals without any transform (rotation, translation, scale) applied to the corresponding geometry instance.

   $B_{GN_{x,y,z}} = N$, with N being the vertex normal.

4. The *position buffer*: which contains the position of the intersection.

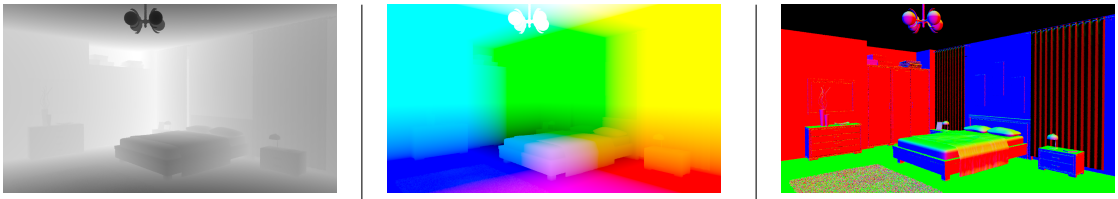
   $B_{P_{x,y,z}} = \frac{(T \times P)_{x,y,z}}{(T \times P)_w}$, with T being the transform matrix, and P the vertex position.

5. The *motion buffer*: which contains the difference between the previous position of that sample in the previous frame and the position of that sample in the current frame. This allows an easier reprojection.

   $B_{M_{x,y,z}} = (T_{i-1} \times P)_{x,y,z} - (T_i \times P)_{x,y,z}$, with T_i being the transform matrix for the current frame, T_{i-1} for the previous frame and P the vertex position.

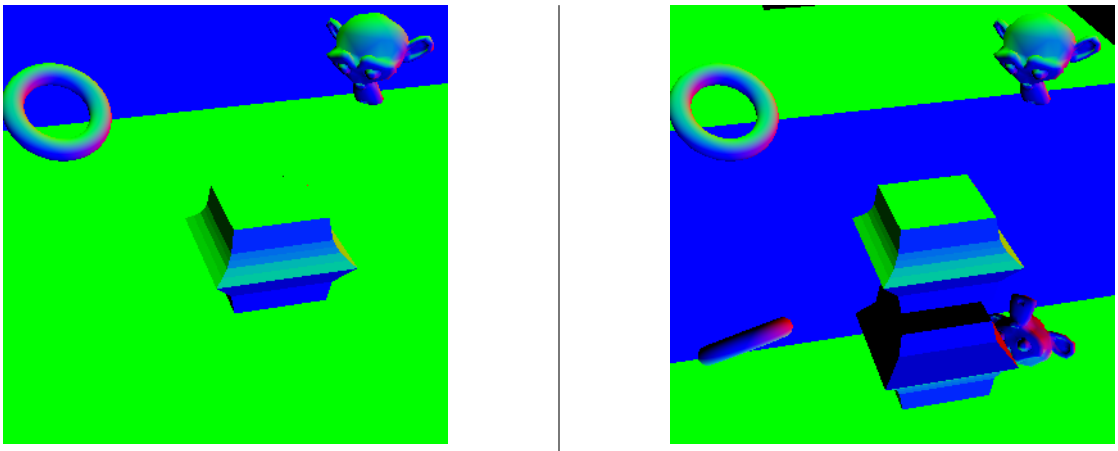
6. The *albedo buffer*: which contains the color without any shading applied to the pixel. Namely, a sample of the texture or the plain color linked to the material at that point.

   $B_{A_{r,g,b}}$



(a) Depth buffer (b) Position buffer (c) Normal buffer

Figure 1.7: Visualization of some of the feature buffers.



(a) (b)

Figure 1.8: (a) Surface normal of the first hit point is used to fill the *normal buffer*. (b) Surface normal of the second hit point (after reflection) is used to fill the *normal buffer*.

1.2.2 Visibility Buffer

Still using rasterization, some renderers use another technique to fetch information for each sample[4]. First described in[2], the Visibility Buffer approach aims to reduce the memory load induced by having several textures containing attributes. Moreover, these attributes could be straightforward to compute inside a pass that would make use of these feature buffers. Instead of storing, and accessing multiple attributes for each sample, the visibility pass only writes the relevant IDs necessary to fetch the triangle that was visible in the given sample. Ultimately, if features are fetched from a texture or computed from the drawn triangle, all algorithms described in the next sections would remain valid. The technique wasn't implemented, but it is worth noting that, to compute the primary rays and the relevant intersection details, multiple techniques exist and are used.

1.3 Temporal Accumulation

The description of the appearance of noise we gave before states that the number of rays we resolve per frame is not enough to have a correct idea of the solution of the rendering equation. However, it is possible to recover the previous frame's result to complete the current solution. If this is done per frame, and by keeping a reliable history, the visual result could eventually converge to an appreciable state. Intuitively, the current color value C_i for a given pixel p could be represented as an accumulation of previous color values over N frames. And thus, obtaining the following first step:

$$C'_i(p) = (C_i(p) + C_{i-1}(p) + C_{i-2}(p) + C_{i-3}(p) + C_{i-4}(p) + \dots)/N$$

As it is not possible to conserve each individual frames $C_j(p)$ throughout a long-running program, the renderer works with a history composed of two frames. The first one is the current frame $C_i(p)$ (result of the ray-tracing), and the previous frame $C'_{i-1}(p)$ (result of the previous accumulation). The previously defined approach becomes:

$$C'_i(p) = (C_i(p) + N \times C'_{i-1}(p))/(N+1)$$

The scenario presented here assumed the camera wasn't moving, as the pixel coordinates in the previous frame remain the same in the current frame. In a more realistic scenario, the camera may move according to user input or a given action, corrupting the coordinates of the pixel we wanted to accumulate. That is where the notion of reprojection inside the overall concept of temporal accumulation arises. it is worth noting that it is a central piece of certain denoising algorithms that we're

about to explain. In the next section, the implemented reprojection mechanism, as well as other alternatives are explained.

By adding the fact that the importance of the new sample may be a parameter and that the previous sample may be located at another pixel coordinates, we obtain the following well-known formula defining the temporal accumulation problem:

$$C'_i(p) = \alpha \times C_i(p) + (1 - \alpha) \times C'_{i-1}(\overleftarrow{p}) \tag{1.2}$$

where:

p	is a pixel coordinate
$\overleftarrow{p}$	is the reprojected pixel coordinate
$C_i(p)$	is the noisy data
α	is a scalar, a weight blending two pixel values, in the present section, it is $\frac{1}{N}$
$C'_{i-1}(\overleftarrow{p})$	is the previously accumulated color

The first part of this section presented the accumulation related to the color of the pixel. However, some denoising methods need some statistical information regarding pixel values. So along the textures C_i, C'_i, C'_{i-1}, and N (the length of the history for a given pixel) that were previously defined and used, the textures M'_i and M'_{i-1} are introduced. Each texture holds two values, namely the first, and second raw moments of a pixel luminance. Once again, the moments of the previous frame are kept in memory. The accumulation for the moment is identical to the accumulation for the color.

$$M'_i(p) = \alpha \times m + (1 - \alpha) \times M'_{i-1}(\overleftarrow{p}) \tag{1.3}$$

where:

p	is a pixel coordinate
$\overleftarrow{p}$	is the reprojected pixel coordinate
m	$= [luminance(C_i(p)), luminance(C_i(p))^2]$ the first and second raw moments for the current pixel value
$M'_i(p)$	is the current accumulated moments
α	is a scalar, a weight blending two pixel values, in the present section, it is $\frac{1}{N}$
$M'_{i-1}(\overleftarrow{p})$	is the previous accumulated moments

1.3.1 Reprojection

Figure 1.9: Example of a situation where a given sample may be accumulated, but has to be reprojected with respect to camera changes.

Naive Reprojection

Using the position feature buffer, the world position of the current pixel is fetched. With the previous camera matrix, it is possible to compute the coordinates of the pixel. And so the reprojection is done as follows:

$$\overleftarrow{uv} := \frac{T_{i-1} \times B_{i,P}(p)}{(T_{i-1} \times B_{i,P}(p))_w} \times 0.5 + 0.5$$

$$\overleftarrow{p} := uv \times dim - 0.5$$

Some tests comparing the value of $B_{i-1,D}(\overleftarrow{p})$, $B_{i,D}(p)$ and $B_{i-1,N}(\overleftarrow{p})$, $B_{i,N}(p)$ are then performed. It allows us to discard samples that were recently (dis)occluded and so, that the accumulation is not applied to an unrelated sample. With this $\overleftarrow{p}$, the previously accumulated color value $C'_{i-1}(\overleftarrow{p})$ can be used to pursue the accumulation over $C'_i(p)$ with the equation 1.2.

Advanced Reprojection

However, the naive reprojection suffers from a lack of robustness at the sub-pixel level. In the proposed implementation, we follow the reprojection approach of[29], and one implementation of it[28] with some changes. The overall approach is divided into two sampling methods, and both methods are explained in this section.

Basically, it starts with a bilinear filter, and if the filter fails to gather enough valid samples, it fallbacks to a bilinear grid. And in the case where the bilinear grid fails too (in the case of a (dis)occlusion for example), the new data coming from the path tracer is used instead of an accumulated value.

As explained before, the first step of the implementation introduces a bilinear filter that starts around $\overleftarrow{p}$ and tries to find valid samples from the previously accumulated texture in the surrounding area, at a distance of one pixel, effectively scanning a 2 by 2 square. A "valid" pixel means that they're related to each other in terms of spatial disposition. With the number of valid samples found c, an array describing if they're valid v_{ij} and their corresponding color values s_{ij}, the following equations find $C'_{i-1}(\overleftarrow{p})$.

As $\overleftarrow{p} \in R$, it is possible to extract its fractional part to be used as a weight in the following interpolation. Intuitively, w allows weighting the interpolation to give more importance to samples that are closer to $\overleftarrow{p}$.

$$w := fract(\overleftarrow{p})^1$$

Using the array describing the validity of the different samples around $\overleftarrow{p}$, the factor f is computed, allowing the filtering of the previous accumulated value. The *accum* value is used in place of $C'_{i-1}(\overleftarrow{p})$ in the accumulation formula 1.2.

$$f := mix(mix(\boldsymbol{v}_{00}, \boldsymbol{v}_{01}, w_x), mix(\boldsymbol{v}_{10}, \boldsymbol{v}_{11}, w_x), w)^2$$

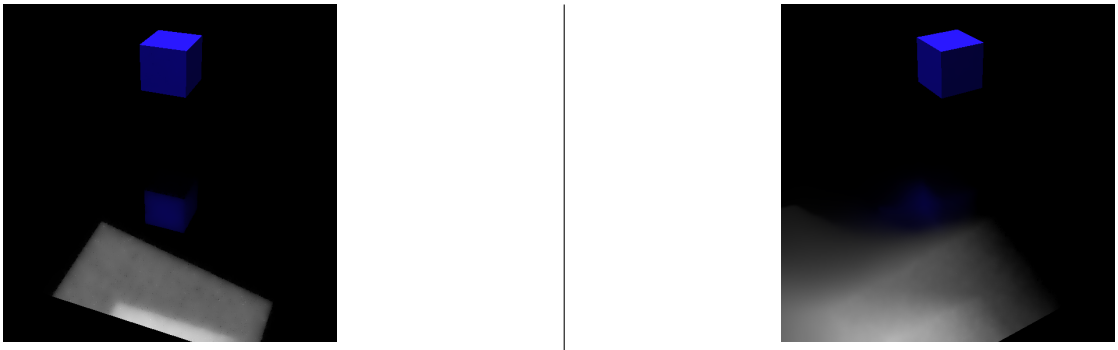
$$accum := \frac{mix(mix(\boldsymbol{s}_{00}, \boldsymbol{s}_{01}, w_x), mix(\boldsymbol{s}_{10}, \boldsymbol{s}_{11}, w_x), w)}{f}$$

However, some early conditions have to be met in order to compute the bilinear filter. They were described in the naive reprojection section, and they remain valid. Namely, samples have to be related to each other. In some cases, the processing fails to find at least one valid sample. When that happens, the initialization of the bilinear filter is discarded, and a bilinear grid which is effectively larger is started. In a more practical manner, it means that if a valid sample wasn't found around $\overleftarrow{p}$, the area of research is extended to a 3 by 3 square.

The principle remains the same as the bilinear filter, neighbor pixels are sampled and their corresponding normals and depths are analyzed to compute a given weight expressing whether or not they belong to the original pixel. The kernel of the bilinear grid is defined as:

[1] $fract(x) := x - floor(x)$ is equivalent to taking the fractional part of x

[2] $mix(\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{w}) = a \times (1 - w) + b \times w$



(a) Static scene. (b) Moving scene.

Figure 1.10: Lags and Ghosting artifacts.

$$k = \begin{bmatrix} 0.0778 & 0.1233 & 0.0778 \\ 0.1233 & 0.1953 & 0.1233 \\ 0.0778 & 0.1233 & 0.0778 \end{bmatrix}$$

w is the weight defining if the two currently compared samples are similar in terms of spatial disposition (regarding normals and depths). Once again, *accum* is used in place of $C'_{i-1}(\overleftarrow{p})$ in the accumulation formula 1.2.

$$accum := \sum_{x=0}^{x=2} \sum_{y=0}^{y=2} k[y][x] \times w(p, \overleftarrow{p} + (x, y)) \times C'_{i-1}(\overleftarrow{p} + (x, y)) \tag{1.4}$$

If the two previous sampling methods failed to grab enough neighboring pixels to perform the accumulation, the new pixel value generated by the path tracer is simply output into $C'_i(p)$. In this scenario, 1.2 is not computed.

$$C'_i(p) := C_i(p)$$

1.3.2 Lags and Ghosting

If the temporal accumulation is tested against a dynamic scene featuring moving shadows or specific highly reflective materials, one should notice certain types of artifacts. To better underline them, a scene with a complete mirror as the ground, with the camera rotating over a blue cube is introduced. As the camera is moving, the reflection of the plane emitting light, and the cube itself, moves as well. However, as the accumulation is tightly linked to the idea of a history of previous values, two major types of artifacts can occur.

If a pixel sees its illumination changing rapidly over the course of a limited number of frames, the accumulated pixel still "holds" old values. That means, only

new values that are being added to the accumulation are valid. In the example, that is the case for pixels on the left of the image. In the beginning, they were reflecting the plane. The phenomenon will see a method to mitigate.

Chapter 2

Related Works

2.1 Machine-Learning with a Recurrent Auto-Encoder

With the recent success in computer vision and image manipulation, the graphics programming field saw new methods of denoising that rely on deep learning. We chose to study one of these types of methods. In this section, the official Open Image Denoise SDK from Intel[12] is used to conduct the denoising, and the generation of results. An implementation inspired by[3] is also presented, alongside its training and the related dataset. The inspiration will be called "Optix Denoiser", as the model is featured in NVIDIA SDK and the related literature under this name.

2.1.1 Algorithm and Implementation

Inputs

As with other denoising techniques, generated images from the G-Buffer are needed to conduct the image reconstruction. In our case, we are interested in the albedo, and in the normal feature buffers. They are used to construct a *deep image* containing 7 channels in total. Namely, the noisy input, the normals in view-space, and the albedo. The 6 last channels aren't noisy as presented previously in the list 1.2.1.

Depending on the situation the model is facing, the image generated by the path-tracer can have different resolutions. With the presented approach, as no layers are resolution dependant, the inputs can be arbitrarily dimensioned.

Architecture

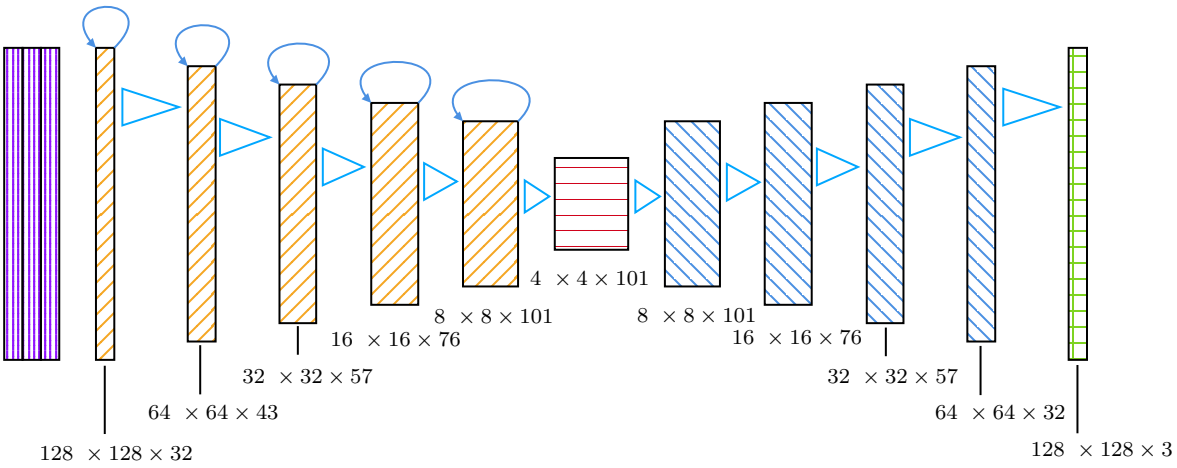


Figure 2.1: Architecture of the neural network, as presented in the Optix Denoiser[3].

As the title may suggest, the neural network can be apparent to an autoencoder. That means, the network can be divided into two parts: encoder and decoder. Namely, the encoder manages feature extraction by being forced to store inputs in a specific layer. This layer is actually much smaller in size compared to the inputs or the outputs[21]. This specific layer is called the bottleneck and is represented in red in the visual. In a typical encoder, layers progressively decrease in size until the bottleneck is reached. From that bottleneck, the image is then reconstructed.

Layers of the encoder can be defined as a series of convolutional layers followed by a max pooling. The max pooling kernel size is 2, and the convolutional layer has a kernel of 3×3, with padding same.

Blocks of the decoder can be defined as a series of linear upsampling, followed by two convolutional layers. Skip connections are introduced at the beginning of each block, in the first convolutional layer. They consist of a concatenation between the output of the previous layer, and the 5 last channels of the original input, namely the albedo and normal features. The last block is completed by a max pooling, to correct the resolution at the end of the pipeline.

Skip connections re-introduce the intrinsic scene knowledge later in the inference, to speed up the convergence time and preserve edges in the end image. They are a major part of the proposed model, and by implementing these, we also noticed a positive difference in the result quality, and in the training time.

2.1.2 Training

Loss Function

The loss function itself can be quite interesting to develop. It comes from the characteristics the final image is expected to possess. Namely, temporal coherence, sharp edges, and visual correspondence to the ground truth.

To ensure sharp edges, the paper is featuring the High Frequency Error Norm[24]. The loss function computes the spatial gradient for the ground truth, and the generated one. And then, it applies a L_1 norm on it. Intuitively, the differences in terms of edges are accentuated, something that a single MSE loss is not able to express. A minor detail resides in the fact that a Gaussian blur has to be applied to avoid noise while computing the spatial gradient. The following loss function was implemented:

$$\mathcal{L}_g = \frac{1}{N}\sum_{i}^{N} |\nabla g_{3\times3}(T_i) - \nabla g_{3\times3}(P_i)| \tag{2.1}$$

One of the major points was related to the temporal coherence, the fact that two consecutive frames weren't provoking visual discomfort such as drastic changes in terms of colors, or illumination. They introduce a function loss based on temporal derivatives, forcing the recurrent parts of the network to effectively activate to ensure temporal coherence. Temporal derivative can be apparent to an operation computing the velocity of the brightness change at the pixel level[11]. Given the fact that we expect the end image to not contain flickering patterns, the loss function computes the difference between velocities of the ground truth and the generated image:

$$\mathcal{L}_t = \frac{1}{N}\sum_{i}^{N} \left| \frac{\delta T_i}{\delta t} - \frac{\delta P_i}{\delta t} \right| \tag{2.2}$$

where:

$\frac{\delta T_i}{\delta t}$ is the temporal derivative computed from the truth image T_i, and the previous truth image T_{i-1}.

$\frac{\delta P_i}{\delta t}$ is the temporal derivative computed from the generated image P_i, and the previous generated image P_{i-1}.

Alongside the more basic $\mathcal{L}_1$ loss, it is now possible to construct the weighted combination of previous losses.

$$\mathcal{L} = w_1\mathcal{L}_1 + w_g\mathcal{L}_g + w_t\mathcal{L}_t \tag{2.3}$$

Dataset

Concerning the dataset,[25] was used, alongside various data augmentation techniques such as horizontal and vertical mirroring which were applied on the deep image itself. The number of different scenes was rather small: 6 different one including illumination changes. So using Mitsuba 3.0[13], the number of different scenes was effectively increased to 12 by producing high-resolution rendering results at 1024 or more spp. As the training is applied on 128×128px chunks, the dataset is composed of 15384 entries. 80% of them is used as the training set, while the rest is used as the validation set. The separation into these two sets is applied after shuffling the entirety of the entries. The discussion regarding the training of our model and its performance is in the experimentation section 3.4.

2.2 Blockwise Multi-Order Feature Regression for Real-Time Path Tracing Reconstruction

Following the use of feature buffers to better describe the underlying properties of each pixel in the scene (the G-Buffer), and the use of the previously described temporally accumulated rendering to add a certain stability and temporal coherence in the noise, it is, therefore, possible to extract information from the scene to conduct a certain type of regression. That is what the Block-wise Multi-Order Feature Regression method[19] used as its starting point.

This section aims to describe the algorithm itself, and more precisely, how it manages to denoise the image using a least-square solution.

2.2.1 Algorithm and Implementation

Definition of the Problem

A mathematical approach alongside the shader implementation is described in this section. They are derived from the original paper[19], and from some related implementations[20][15]. As stated before, the G-Buffer and the noisy texture are the starting point of this method. A set F of N feature value is defined as follows:

$$F = [F_1, ..., F_N] \tag{2.4}$$

Note that $F_i \in R$ is different from the previously described feature buffer. In most configurations of F, it's in fact the value of each channel of each value of the original feature buffers. For example, F_1 could be the channel value of B_{D_x}.

From this set F, an extended set T can be derived by rising to a specific power certain feature buffers:

$$T = [F_{n_1}^{a_1}, ..., F_{n_m}^{a_m} ..., F_{n_M}^{a_M}] \tag{2.5}$$

where:

m	$= 1, ..., M$
n_m	$\in \{1, ..., N\}$
a_m	$\forall m \neq 1 : a_m \in \mathbb{R}_0^+$
a_1	$= 0$
	the first element of the extended set must be a constant buffer such that $F_{n_1}^0 = 1$

The best configuration of this extended set is obtained by experimentation. One possible configuration given as an example could be the *normal* and the *position* buffers raised to the power of 1 and the position feature buffer raised to the power

of 2, which gives $T = [1, N_x, N_y, N_z, P_x, P_y, P_z, P_x^2, P_y^2, P_z^2]$. Some possible results obtained in different configurations are discussed in the experimentation section.

The *Blockwise Multiple Feature Regression* problem defined in the related paper[19] involves finding a series of weights $\hat{\alpha}^{(c)} \in \mathbb{R}^M$ that allows reconstructing a denoised block via a weighted sum of the elements of the extended set T. These weights should be computed such that they minimize the squared error between the noisy input $Z^{(c)}$ and the weighted sum. Therefore, the regression problem is defined as:

$$\hat{\alpha}^{(c)} = \arg\min_{\alpha^{(c)}} \sum_{p \in \Omega_j} (Z^{(c)}(p) - \sum_{m=1}^{M} \alpha_m^{(c)} F_{n_m}^{a_m}(p))^2 \tag{2.6}$$

where:

Ω_j	set of absolute image coordinates within a specific block (i, j) it is worth noting that Ω_j is not a set with an exhaustive number of elements, but a limited number of coordinates, that we call "samples" in the experimentation section
$Z^{(c)}(p)$	the accumulated noisy data at coordinates p and at the channel c

And the denoised image can be reconstructed with:

$$\hat{Y}^{(c)}(p) = \sum_{m=1}^{M} \hat{\alpha}_m^{(c)} F_{n_m}^{a_m}(p) \tag{2.7}$$

Solution to the Problem

The written expression 2.6 is actually a least square problem. The novelty of the BMFR paper[19][26] resides in the way in which they solve it. Specifically, the Householder QR factorization using matrix-vector notation is used allowing a parallel and fast computation of the said equation. Multiple works demonstrated the available ways of computing it efficiently and parallelly[17][26].

The annex contains more information regarding the deployment of the Householder QR factorization, and the novelty regarding its computation in the annex A.1. Generally speaking, $\hat{x}$ should be the approximated solution of $Ax = b$. Such that the distance between $A\hat{x}$ and b is minimized.

$$A\hat{x} \cong b$$

With $r = \|b - A \times x\|$ being the residue to minimize:

$$\hat{x} = \arg\min_{x} \|b - A \times x\|$$

A matrix T is defined. T is $W \times M$ where W is the number of coordinates in Ω_j and M is the number of feature in F. T is obtained by concatenating specific

column-vectors obtained by extracing $F_{n_i}^{a_i}(p_j)$ ($\left[F_{n_i}^{a_i}(p_0)\dots F_{n_i}^{a_i}(p_W)\right]^T$) for column i.

$$T = \begin{bmatrix} F_{n_1}^{a_1}(p_1) & F_{n_2}^{a_2}(p_1) & F_{n_3}^{a_3}(p_1) & \dots \\ F_{n_1}^{a_1}(p_2) & F_{n_2}^{a_2}(p_2) & F_{n_3}^{a_3}(p_2) & \dots \\ F_{n_1}^{a_1}(p_3) & F_{n_2}^{a_2}(p_3) & F_{n_3}^{a_3}(p_3) & \dots \\ \dots & \dots & \dots & \dots \end{bmatrix}_{W\times M}$$

Said in another way, all features are extracted for each pixel coordinates in Ω_j, and placed as rows in the T matrix. The matrix T is then augmented by adding the vector column $z^{(c)}$ which is $\left[Z^{(c)}(p_1)\dots Z^{(c)}(p_W)\right]^T$ to obtain $\widetilde{T}$:

$$\widetilde{T}^{(c)} = \left[T, z^{(c)}\right] = \begin{bmatrix} F_{n_1}^{a_1}(p_1) & F_{n_2}^{a_2}(p_1) & \dots & Z^{(c)}(p_1) \\ F_{n_1}^{a_1}(p_2) & F_{n_2}^{a_2}(p_2) & \dots & Z^{(c)}(p_2) \\ F_{n_1}^{a_1}(p_3) & F_{n_2}^{a_2}(p_3) & \dots & Z^{(c)}(p_3) \\ \dots & \dots & \dots & \dots \end{bmatrix}_{W\times(M+1)}$$

From now, some assumptions have to be made to ensure the next steps are properly defined. First of all, $W \gg M$ ensures that a block of pixel coordinates Ω_j contains a lot more pixels than F contains features. Then, $\widetilde{T}^{(c)}$ has to have full rank[1]. Sometimes, $\widetilde{T}^{(c)}$ may be rank deficient which provokes errors within the QR factorization that we'll describe in the next part. To counter this phenomenon,[19] employs a form of stochastic regularization consisting of the addition of a random value to each element of the matrix. These values are uniformly distributed over a specific interval, and so have a zero-mean value. And so, we obtain an updated definition for $\widetilde{T}^{(c)}$ if we consider the matrix N with each element being the random value to be added:

$$\widetilde{T}^{(c)} = \left[T + N, z^{(c)}\right]$$

The Householder QR factorization is applied to yield $\widetilde{Q}^{(c)}$ and $\widetilde{R}^{(c)}$ such that $\widetilde{T}^{(c)} = \widetilde{Q}^{(c)} \times \widetilde{R}^{(R)}$. As expressed in the annex, computation complexity is reduced by omitting the computation of $\widetilde{Q}^{(c)}$. Therefore, $\alpha^{(c)}$ is found by solving the following system, where R is the upper left $M \times M$ matrix, and $r^{(c)}$ is the upper right $M \times 1$.

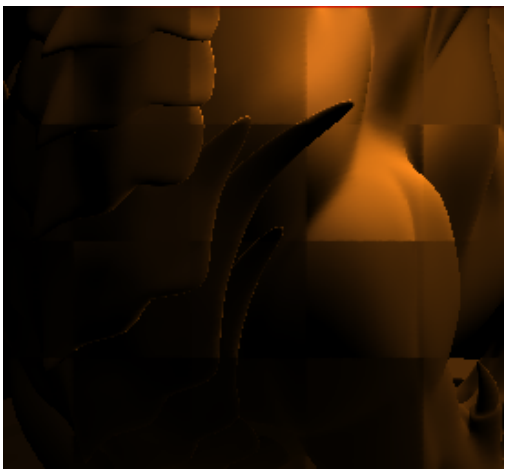
$$R\hat{\alpha}^{(c)} = r^{(c)}$$

Once the system has been solved, each individual pixel can be denoised. A dot product between $\alpha^{(c)}$ is computed for the current block, and the vector of feature fetched for the current pixel is operated.

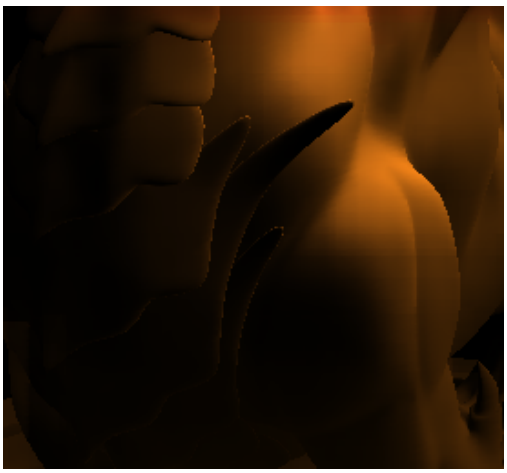
[1]The rank of the matrix $\widetilde{T}^{(c)}$ has to be equal to $(M+1)$.

$$\hat{y}^{(c)} = T\hat{\alpha}^{(c)}$$

Notice that the matrix T used to execute the weighted sum with the factors $\hat{\alpha}^{(c)}$ is not the one where stochastic regularization was applied. Moreover, the R matrix can be reused to filter other channels, as long as the z^c vector is multiplied by the corresponding H_i, resulting from the Householder factorization. This single fact of being able to reuse R instead of doing the same calculations for the three channels allows for decreasing the computational complexity of the denoising pass.



(a) Blocky look without offset being applied. Rendered with our implementation of BMFR.



(b) Accumulation being applied after offsetting the grid. Rendered with our implementation of BMFR.

Figure 2.2

Denoising by block introduces a noticeable blocky look to the final processed frame. To oppose it, each frame will see a different offset being applied to the grid, allowing each block to effectively move around. By adding a new temporal accumulation step 1.3, and by adding this moving offset, the blocky look quickly disappears after the first frame. This is the optional temporal accumulation pass displayed in gray in the general pipeline visual 1.1.

2.3 (Adaptive) Spatio-temporal Variance Guided Filtering

The intuition between[30],[29], and[6] resides in the fact that even if a given image region is noisy, the overall visual aspect of the given region can be obtained by locally "blurring". Blurring allows a decrease in the variance related to pixel values and effectively suppresses the noise. However, naively applying a blur may remove sharp edges and illumination details. This section aims to describe and study the algorithm presented in[30] (which is itself inspired by[29] and[6]).

Moreover, the following method aims to solve the different caveats of temporal accumulation. These problems were described in the previous chapter and were the eventual ghost and lag artifacts that usually appear when a light source or a shadow caster moves rapidly within the scene, leaving an unpleasant trail due to pixels still holding the accumulated values of the previous states. Previous approaches involved giving more importance to new data, but at the cost of stability, rendering some denoising methods less able to effectively filter the image. Intuitively, at the pixel level, drastic changes in terms of color are tracked which are typically provoked by light changes (shadows or illumination). It allows, on one hand, to discard the accumulation history for the given pixel, and on the other hand, to increase the filtering for the given region that was recently influenced.

The pipeline is divided into multiple steps. For easier reading, the mathematical formulas are described in the order they appear within the said pipeline along with their usage.

2.3.1 Algorithm and Implementation

Gradient Estimation

The purpose of this step is to actually mark the different pixels that have seen noticeable changes in terms of illumination. This is done by computing a gradient between the previous luminance, and the current luminance within a 3×3 block. We say within a 3×3 block, because not all pixels interfere with the gradient computation, but the value of the pixel with the highest luminance within the said block. Given the fact that subsequent frames may have seen a camera change, the gradient is computed for the luminance found at the reprojection pixel coordinates. The reprojected pixel coordinates are noted $\overleftarrow{p}$. The formula to compute the gradient is taken from[4]:

$$G_p = \left[\frac{|L_{i-1,\overleftarrow{p}} - L_{i,p}|}{\max(L_{i-1,\overleftarrow{p}}, L_{i,p})} \right]^2$$

Note: if $\max(L_{i-1,\overleftarrow{p}}, L_{i,p})$ is null, $G_p = 0$.

This gradient step fills a texture that has a 3 times lower resolution. An à-trous filtering is applied on the result of the computation to avoid outliers and follows the fact that an illumination effect is never seen alone on a single pixel, but on a larger zone.

Temporal Accumulation

Temporal accumulation was already discussed previously 1.3, and it was effectively stated that lags and ghosting were frequent in the case of a dynamic scene where objects may reflect or cast shadows differently throughout different frames. The cause of these artifacts was the accumulation part of the global pipeline, that is why it is also there that a solution to this problem is presented.

Previously the α parameters in equation 1.2 ended up being $\frac{1}{N}$, some workarounds that can be considered "hacks" exist, one of them resides in giving new samples more importance, so that if they bring new information regarding shading, old history disappear faster. With[30] however, α is computed with respect to the gradient found in the previous step. Intuitively, α is expected to be really close to 1.0 when there was a drastic change in illumination (when G_p is high).

$$N'_p = \min(N_p \times (1.0 - G_p)^{10} + 1.0, 256)$$

Here, we can consider some parts of the pixel history discarded. A high G_p value renders the left part close to zero, effectively shrinking the number of frames in the said pixel history. Notice, we now consider a "per-pixel" history, and a "per-pixel" α.

$$\alpha_p = \text{mix}(\frac{1}{N'_p}, 1.0, G_p)$$

The α_p part gets close to 1.0 when G_p is high, meaning that in equation 1.2, the new pixel value generated by the path-tracer $C_i(p)$ is almost copied as is in $C'_i(p)$. In the other case, if G_p is low (meaning no drastic change), α_p is closer to the original α, that is $\frac{1}{N}$. To better illustrate the behavior of these snippets consider the following two situations:

Given	
$G_p = 0.01$	$G_p = 0.95$
$N_p = 45$	$N_p = 45$
Computed	
$N'_p = 45.552$	$N'_p = 1.0$
$\alpha_p = 0.023$	$\alpha_p = 0.999$

This explains well how the previous code can discard pixel history when the gradient of a specific pixel is high. Some hyper-parameters were omitted, they're supposed to cover edge-case such as division by zero, or to scale specific parts of the presented formula. As the reader may see, they have been omitted to avoid overloading the explanations.

À-trous Wavelet Filtering

After properly accumulating the results produced in the previous frames, and in the current scene, it is time to actually denoise it. As described before, a smart blur has to be driven on the whole picture to remove noise, while still keeping sharp edges and illumination details. To achieve it, some edge-avoiding functions taken from[29] have to be declared beforehand. We'll see how they interact at a higher level later.

Edge-avoiding functions allow the differentiation of two pixels at coordinates i and j. If they have similar depth, normal, and luminance, it is correct to blur them together. On the contrary, the blur shouldn't be applied, the importance of the value at j should be diminished. A typical situation can be at the edge of a wall, where depth can easily allow discarding of j.

$$w_n(i,j) := \max(0, N_{xyz}(i) \cdot N_{xyz}(j))^{\sigma_n} \tag{2.8}$$

The weight is linked to the normal features of two pixels with coordinates i and j respectively. When $N_{xyz}(i)$, and $N_{xyz}(j)$ are significantly different, they may not be related, and so the dot product referenced in the weight ends up being a little value. The more similar the two normals are, the greater the weight will be. σ_n is a hyperparameter that can be tweaked depending on the needs of the end user.

$$w_d(i,j) := \exp\left(-\frac{|D(i) - D(j)|}{\sigma_z \times |\nabla D(i) \cdot (i-j)|}\right) \tag{2.9}$$

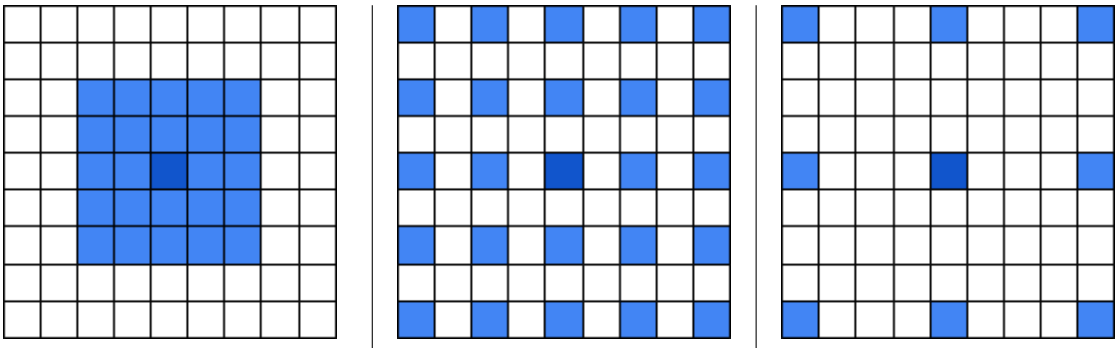
In the same manner, when two pixels see their depths being really different, they are unrelated. Here, the more $D(i)$, and $D(j)$ are different, the greater the difference will be, and in the end, e^{-x} will get nullified. The denominator is meant to account for oblique surface using the screen-derivative related to depth. Two related samples along an oblique surface may see a radical change in depth. A naive edge-function discarding sample based on a depth offset may be incorrect if discarding these. That is why the normal is included. As it detects if the detailed samples are belonging to the same surface.

$$w_l^{(k)}(i,j) := \exp\left(-\frac{|\hat{l}^{(k)}(i) - \hat{l}^{(k)}(j)|}{\sigma_l \times \sqrt{g_{3\times3}(\mathrm{Var}(l^{(k)}(i)))}}\right) \tag{2.10}$$

As defined in the pipeline section 1.3, the G-Buffer also contains the raw moments of the current pixel value. By computing the variance of the value hold by the pixel, the edge-function can detect outliers. Some kind of outliers are fireflies or samples with radically different luminosity. The first kind are samples produced by the path-tracer that are often really high values, with very low probability density[35]. Even if they are clamped in our implementation, they are still disturbing. The second kind of outliers can simply be explained by the eventual shadow or reflection that can affect one sample an not the other, it is mandatory to conserve sharp illumination effects.

$$w^{(k)}(i,j) := w_n(i,j) \times w_d(i,j) \times w_l^{(k)}(i,j) \tag{2.11}$$

And here is the weight function as a whole, the product of previously described weights. We now describe the wavelet operations that effectively use it.



(a) $i = 0$ (b) $i = 2$ (c) $i = 3$

Figure 2.3: Different steps of the à-trous filter.

$$A_{i+1}(p) := \frac{\sum_{o\in\Omega} A_i(p + 2^i \cdot o) \cdot w^{(k)}(p, p + 2^i \cdot o) \cdot h_i(o)}{\sum_{o\in\Omega} w^{(k)}(p, p + 2^i \cdot o) \cdot h_i(o)} \tag{2.12}$$

where:

Ω	is the list of relative offsets, with a fixed distance of 1 around the center of the kernel
k	is the number of offsets within Ω
$A_i(p)$	is result of the ith iteration of the à-trous filter
$A_0(p)$	is equivalent to the texture containing the temporal accumulation result of the current frame ($C_i'(p)$ in the previous section)
$w^{(k)}$	is the previously described weight function
$h_i(o)$	is a filter kernel value, where $h = (\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16})$

Seed Reprojection

As expressed in the introduction of this work, rays bounce randomly on the hit surface, generating a random result in each frame. One can simply not use these values that can be completely incoherent from frame to frame to construct a gradient (see 2.3.1) in the temporal accumulation phase. Remember that we're using the pixel value with the highest luminance within a 3×3 block.

To generate a random number, a random number generator associated with a specific seed is needed. With our implementation, random numbers are generated using blue noise[8]. In a naive renderer, each ray cast from a specific pixel has a different seed for each frame, allowing to discover new information. Actually, if the seed was identical, subsequent random numbers would be identical, and so the result of the path-tracing.

In this context of wanting to preserve the highest luminance value to not corrupt the gradient computation, we want to reproject the seed that generated this value across multiple frames. That means if the pixel value drastically changes, even if the seed was kept, that is surely related to a change in shadow/illumination. The pixel undergoing the reprojection is only the one holding the maximum luminance value, other pixels within the block receive a new seed, and so, will find another value in the next path-tracing step.

2.4 Tiny Neural Network Denoiser

Machine learning methods, such as the optix denoiser 2.1.1, suffer from their large number of parameters. Their training requires a large dataset to avoid over-fitting and noticeable hardware to do the inference in a reasonable time. Following the recent research in neural cellular automata[22], works have demonstrated that simple networks are able to generate specific textures, and more importantly, to reconstruct it after degradation[23]. The intuition of the presented method is to apply this architecture to denoising by considering that the noisy image needs to be reconstructed. In this section, we develop the underlying of a novel method based on cellular automata, as we didn't find any existing algorithm using cellular automata to reconstruct a noisy image generated by a path-tracing renderer. With as little as 351 parameters for the complete neural network, it contrasts drastically with current machine learning methods which favor a certain over-parameterization. Conclusions and the different caveats encountered during the testing are developed in the experimentation section.

2.4.1 Algorithm and Implementation

The explanation considers the model as a cell. The image is now a grid of cells and each cell has a specific value, the value vector of a single pixel of the deep image to be more precise. The different cells are expected to update their values with respect to the values of the different other cells that are adjacent. With a specific set of rules encoded in the weights of the neural network, the value of the different cells are expected to converge, after multiple iterations, to a denoised state. Intuitively, an iteration is composed of two steps in our implementation. The first step involves the *perception* of the surroundings, that is, the values of the adjacent cells. It constructs a perception vector that is larger than the depth of the deep image. The second step is the update, which we can call the *growing* step. The tiny neural network takes the perception vector as input and returns an intermediate state value that is added to the previous state value. The following section details more formally the different operations that are applied.

The architecture accepts as input the concatenation of the channels of the noisy image, the normal and albedo feature buffers. This concatenation forms a deep image comparable to the input of 2.1.1. The formal procedure follows the nomenclature of[22]. With the deep image s defined as:

$$s = [s^0 = R, s^1 = G, s^2 = B, \\ s^3 = N_x, s^4 = N_y, s^5 = N_z, s^6 = A_x, s^7 = A_y, s^8 = A_z] \quad (2.13)$$

The perception step is defined as a series of convolutions using pre-defined filters. Namely, the cell perceives the surroundings with:

- A Gaussian filter applied to the color channels.
- A Laplace filter applied to the color, normal and albedo channels.
- A vertical and a horizontal Sobel filter applied to the color, normal and albedo channels.

The perception vector is the concatenation of the results of this filter with the original deep image values so that the cell can keep track of its current state. Which gives:

The initial state:

$$s = [s^0 = R, s^1 = G, s^2 = B, \dots, s^8] \tag{2.14}$$

Note that the presence or not of a specific channel or the application of a specific filter is one parameter that has to be experimentally defined.

The filters are applied as defined previously for their respective channel, and their results are concatenated to form the perception vector. Its size is 39 elements. Notice that the first deviation from the original paper is here. Not all filters are necessarily applied to all channels. The Gaussian filter, for example, is only applied to the RGB channels. While the Laplace, and Sobel filters are applied to all. They aren't listed exhaustively in the following equation, but the reader is expected to refer to the previous list 2.4.1.

$$p = concat(s, K_g * s^0, K_g * s^1, K_g * s^2, \dots) \tag{2.15}$$

The set of rules is applied to the perception vector. Rules are expressed in a 39×9 matrix, comparable to a one dimension convolution which decreases the dimensionality of the vector to the original depth of the deep image.

The activation function introducing non-linearity is the sigmoid function. That is the second deviation from the original paper. While the authors chose to apply the set of rules to $concat(s, abs(s))$, we omit using the activation function before the rule set. Moreover, applying *abs* to a feature vector that contains normal vectors is inappropriate given the fact that negative numbers have a strong meaning in this case. Sigmoid gives good results and its image domain is in the range of our exposed data.

$$s_{update} = sigmoid(pW_{39\times 9}) \tag{2.16}$$

Moreover, exposing the perception vector containing the blurred color channels, the normal channels and/or the albedo creates a local minimum that the neural

network always falls into. The culprit was the original absolute value activation function. In the sense that during training, the neural network only updates the different weights to always output a blurred version of the noisy image or an absolute value of the normal or a grey version of the albedo. While it effectively produces a coherent image, the first case was breaking the intention of reconstructing the image, second and third ones were simply laking any form of illumination. Using the sigmoid function prevents this case.

Applying the ruleset on the perception vector gives us a new vector, that we apply to the previous cell state, giving the new state that is used in the next iteration. At each iteration, regardless of their number, the values contained in the first three channels are coherent images.

$$s_{next} = s + s_{update} + bias_{1\times 9} \tag{2.17}$$

2.4.2 Training

Dataset

The inner advantage of having a little number of parameters is the reduced dataset that you need to train the network. And still with this little number, the risk of overfitting on specific scenes or disposition is reduced as well. The dataset we used to train the network is composed of three different scenes that are different from the scene used in the experimentation section, totalizing a thousand 128×128 chunks. One-third was generated with our renderer, which features a pixel-perfect series of $(noisy, normal, albedo, reference)$ entries. Images are pixel-perfect in the sense that no anti-aliasing or ray-jittering was performed, which ended up being a crucial part, knowing that each cell is only able to see the adjacent cells. That is not the case with[25] for example.

Loss function

We first evaluated the loss function presented in[33] specially tailored for denoising of photos and pictures. In their case, the end image suffers from a very different kind of noise. Therfore, the model was decreted as not applicable in our case. And so, the loss function presented in the autoencoder[3] that we previously described is re-introduced. Given the fact that the end image was blurry, the loss function ended up very similar to the one defined to[3].

$$\mathcal{L} = w_1\mathcal{L}_{mse} + w_2\mathcal{L}_g \tag{2.18}$$

With $\mathcal{L}_g$ being exactly 2.1.2. And $\mathcal{L}_{mse}$ being the mean squared error loss.

To decide whether the loss function is to be applied after each iteration or after a certain number of iterations, we chose to stick to the original training process of[22]. That is, each batch of the dataset used for training sees a random number of iterations. The random number allows the neural network to stay stable given an arbitrary amount of iterations, and to effectively reconstruct the image. And so, the loss function and the back-propagation happen at the very end of the last invocation of the presented model.

Chapter 3

Details of implementations

3.1 Introduction

The implementation of the 3 presented methods of denoising strictly follows the original papers and the state of the art, the main differences are explained in this section. All tests are conducted within the 3 different scenes (namely the *Sponza*, *DeleteMe* and the *Dragon*). Moreover, the resolution of the end images is 1280×758, and stays that way throughout this section, the content of the feature buffers is kept the same. Physical hardware that was used to benchmark is also kept the same (namely an AMD ATI Radeon RX 570 as the GPU, and an AMD Ryzen 7 2700X (16) @ 3.700GHz as the CPU).

Each time a frame is used to compute the objective quality, the frame of the denoised pipeline is compared against an image generated after accumulating 1024 samples. If a movement was applied to the camera, it is specified.

Method	Sponza	DeleteMe	Dragon	(ms)	*Device*
A-SVGF	0.940	0.914	0.989	5.46	GPU
BMFR	0.954	0.928	0.987	8.59	GPU
OIDN	0.941	0.924	0.983	∼600	CPU
TNND	0.906	0.838	0.848	∼200	CPU

Table 3.1: Side-by-side comparison of the objective image quality SSIM alongside the time needed to run the denoiser. Note the difference in running devices that prevents us from ranking the different methods in terms of pure performance between device types.

3.2 BMFR

3.2.1 Exploration of Parameters

The official implementation by the authors of the paper was made in OpenCL, and suffers some limitations preventing us from integrating it directly in our demo. Firstly, the initial implementation requires pre-rendered noisy inputs, and pre-rendered feature buffers. Secondly, it was clearly stated that some parameters were fixed, preventing us from exploring the possible performance. Some examples of fixed parameters were namely the dimension of the block on which the regression is computed. A new implementation is presented in the linked software. It strictly follows the presented ideas, while allowing the modification of the complete set of parameters. Objective quality is measured using our implementation, while execution time measurements are done using[15]. By default, all tests are using a 64×64 block size with 36 samples per block.

Number of samples per block

On a static scene, the number of coordinates inside Ω_j and its effect concerning the quality of the denoised image is studied. Intuitively, it is expected to see a noticeable improvement with the increasing of samples as it allows the least square approach to better handle the approximation of $\alpha^{(c)}$. Tests are made with the dragon scene with a static camera.

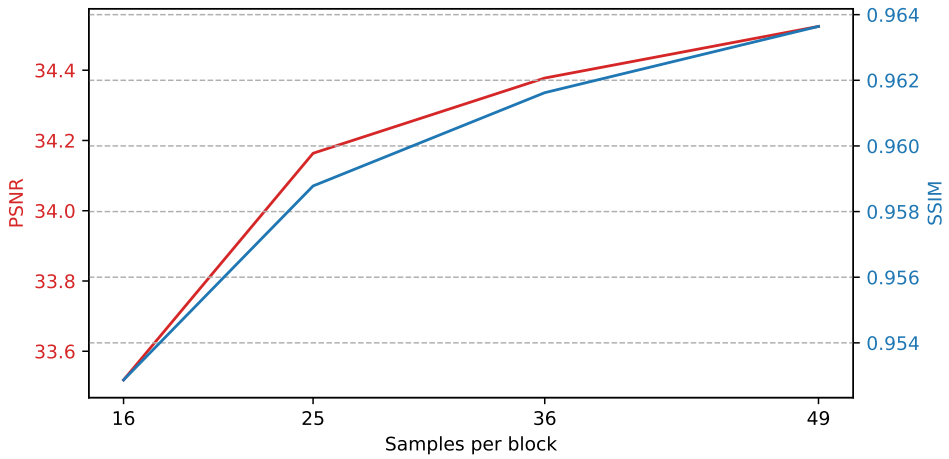
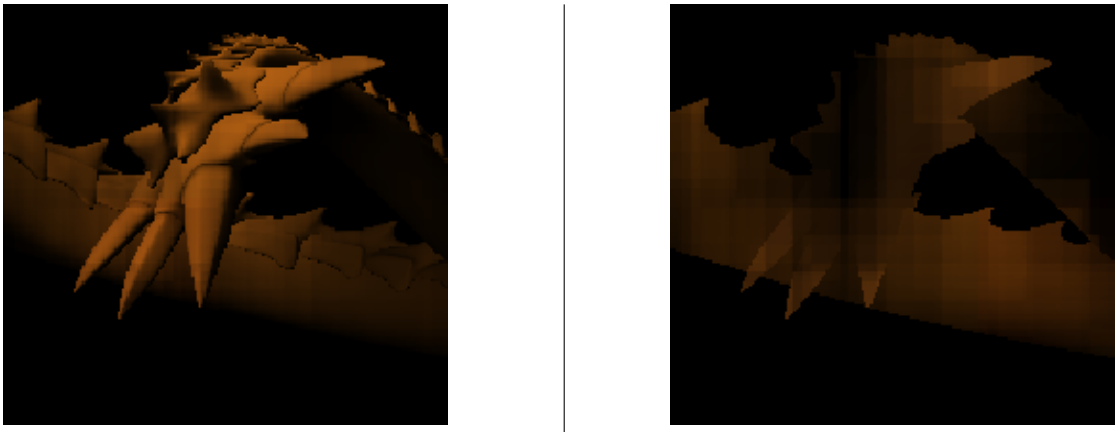


Figure 3.1: Image quality with different sizes of blocks.

Feature buffers used

The nature of the features used to fill $\tilde{T}^c$ greatly impacts the end result. It was noticed that using only normals was the smallest feature set sufficient to produce an acceptable image depicting hard changes in the scene geometry. Even if the computed PSNR and SSIM values are lesser with this configuration, details aren't blurred out and the overall aspect is more pleasant.



(a) Denoising using only normals.

(b) Denoising using only positions.

Figure 3.2

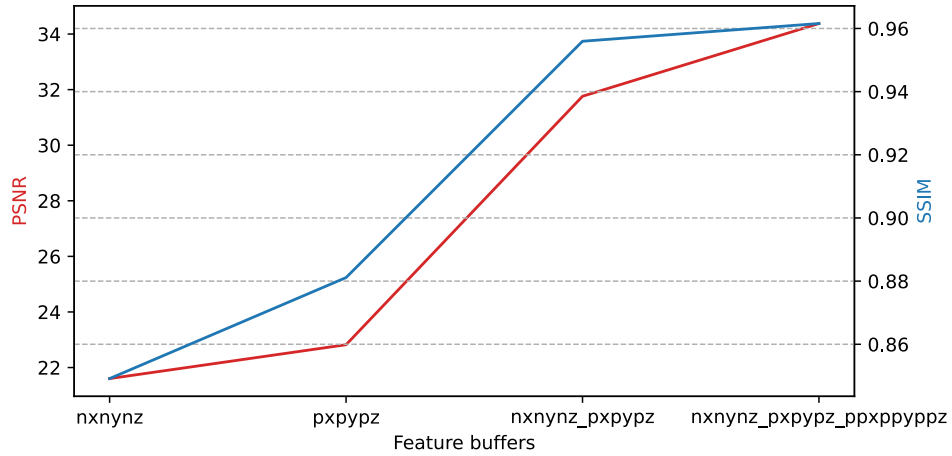


Figure 3.3: Image quality with different features. A noticeable quality increase is present with a higher variety of features.

Different feature sets that were studied are presented in the following order:

1. $[N_x, N_y, N_z]$
2. $[P_x, P_y, P_z]$
3. $[N_x, N_y, N_z, P_x, P_y, P_z]$
4. $\left[N_x, N_y, N_z, P_x, P_y, P_z, P_x^2, P_y^2, P_z^2\right]$

3.2.2 Size of blocks

Lastly, the size of the blocks is changed. Some configurations can be considered absurd given the fact that their dimension is not enough to grab the complexity of the geometry and shading behind the given area. Moreover, by keeping 36 samples per block, different rows of $\tilde{T}^{(c)}$ end up very similar, breaking the QR factorization. That is the case of the 8×8 configuration.

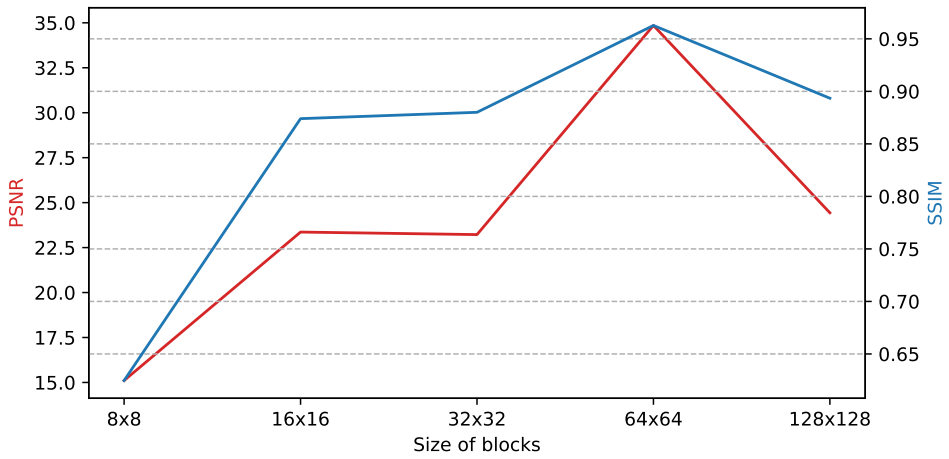


Figure 3.4: Image quality with different features. A noticeable quality increase goes along with a bigger block size. The maximum is reached at 64×64.

3.2.3 Computational Complexity

Denoising is applied on 15 frames for the two specified scenes, and the total time is averaged. Moreover, the size of the matrix needed to solve the least square problem strictly depends on the number of features. Therefore, the evolution of the computational complexity regarding the number of features is analyzed. And as an evident conclusion, and that joins the previous conclusion, increasing the number of feature buffers increases the image quality, but drastically increases the

execution times. Thus, it forces us to find the perfect place, which is situated in the third configuration.

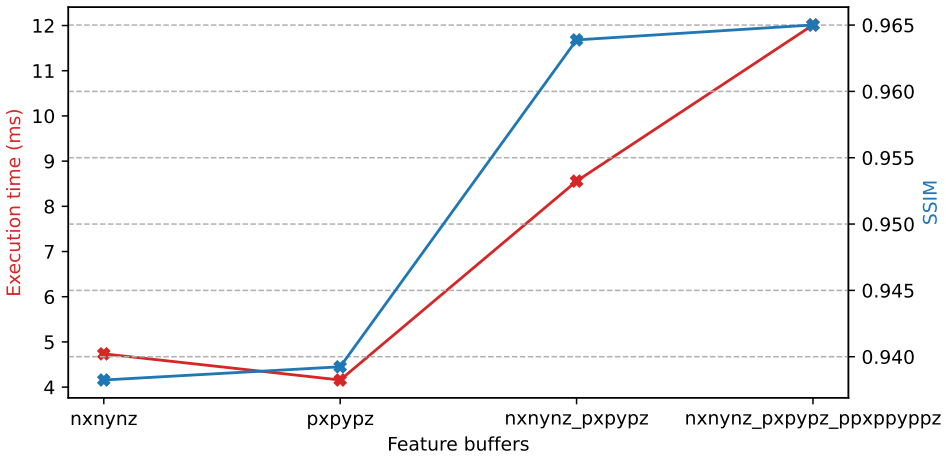


Figure 3.5: Execution times and their relation to the number of feature buffers used to denoise the *Sponza* scene, and the *Dragon* scene.

3.3 A-SVGF

The implementation of A-SVGF in our work follows the main principles of[30] and[4]. One of the main differences resides in the fact that, while[4] decided to denoise the shadows cast by the first secondary rays, see 1.1.3, and the illumination generated by the second secondary rays *separately*, our implementation does not. That is because results allowed us to be confident in the fact that it wasn't necessary for our use case. Moreover,[4] features some game-specific settings that were simply off-topic for a more general analysis.

Number of à-trous iterations

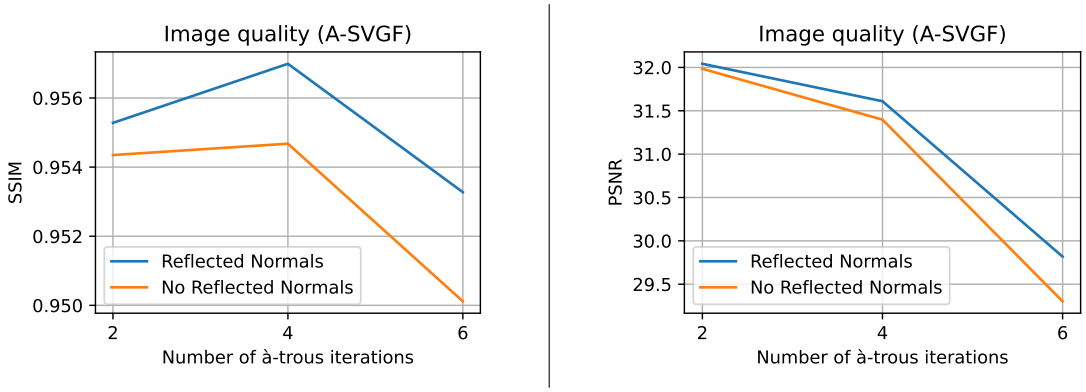


Figure 3.6

The presence of reflected normals, as described in figure 1.2.1, alongside the number of à-trous iterations is discussed with the *DeleteMe* scene. A slight increase in objective image quality is noticeable for all presented number of iterations. On the other hand, the subjective quality of the image suggests that reflected normals are better for allowing reflective surfaces not to lose detail. Adding new à-trous iterations is not cost-free. On our hardware, each iteration in the color step takes about $2ms$ms to be executed.

3.4 Auto-encoder

3.4.1 Training

The proof-of-concept inspired by the explained architecture depicted in figure 2.1.1 is slightly modified to match the performance of the original model. Therefore, in this section, we detail more precisely our take on the auto-encoder. Using 100 epochs and the original loss, dark parts of the image are not correctly denoised. They appear far more bright. By experimentation, the $\mathcal{L}_1$ loss is considered to be the culprit and replaced by a mean squared error loss. Moreover, the edges of the generated image do not display sharp angles and continuous lines. It's especially noticeable with windows or plants. We experimentally decrease the importance of $\mathcal{L}_{mse}$, and increase $\mathcal{L}_g$, to better account for this trend. The focal frequency loss is also introduced, helping to mitigate the blurry edges and the eventual artifacts that are more visible in the spectral domain[14].

The number of layers and their nature are kept, and it's worth noting that only the recurrent part, allowing each encoder layer to keep a history of the hidden state of the previous frame is omitted.

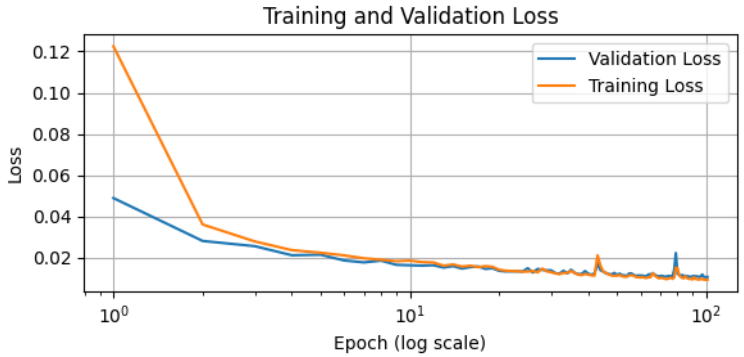


Figure 3.7: Training and validation loss with 100 epochs.

The inputs also slightly change. Instead of using the screen-space normals which is a $2D$ vector, the world normals are used. It effectively increases the number of channels in the input, a choice similar to[5]. Overall, the quality of the result is similar to the OIDN denoiser with some differences in dark environments. However, given the fact that the dataset was considered too small and that our pytorch model couldn't be easily ported to our renderer, objective image quality is computed using an already trained model featured in the mentioned framework.

3.4.2 Exploration of Parameters

Concerning OIDN[12], using the distributed weights from the official repository, two feature buffers can be used to conduct the denoising. As explained before, the features can be the normal map and the albedo. It can accept 3 different types of input sets. Namely , $Noisy, Albedo$ or just $Noisy$. Tests are expressed with different scene layouts, at different positions (same across different sets) and by taking the average of the results. Unexpectedly, the quality difference between the different input sets varies insignificantly. To be more precise, our tests have a difference of about 0.01 in terms of SSIM.

- $Noisy, Normal, Albedo$: 0.946455
- $Noisy, Albedo$: 0.9356857
- $Noisy$: 0.93522143

3.4.3 Computational Complexity

3.5 Side by side Comparison

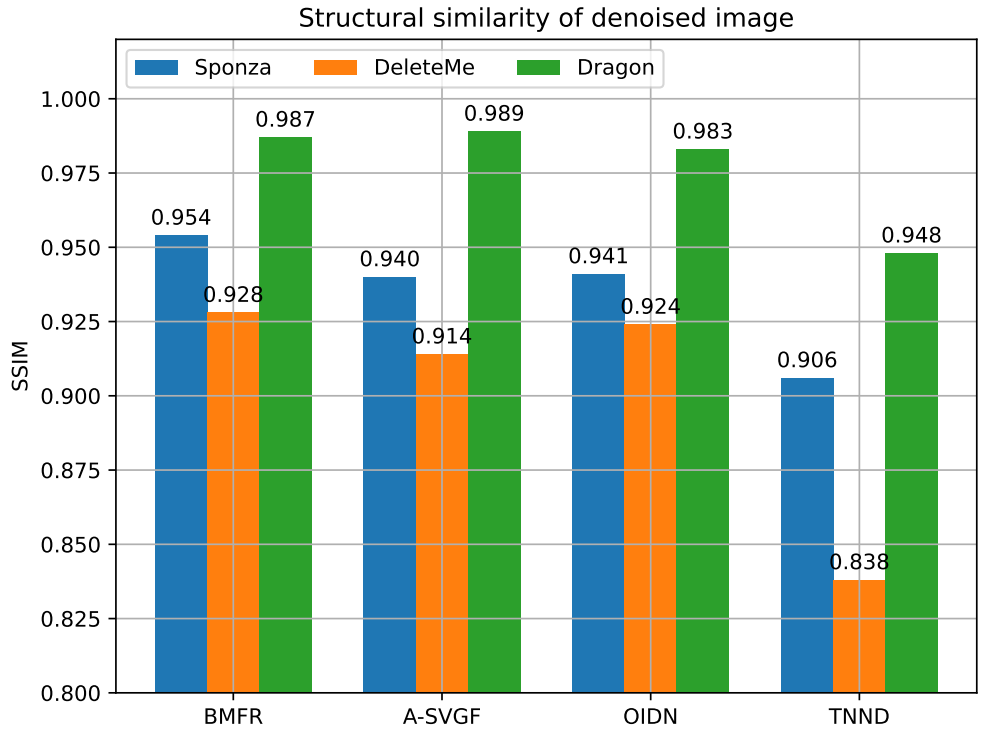


Figure 3.8: Side by side image quality comparison, for the same inputs, but with different denoised methods.

Using three different scenes featuring different difficulties that a typical production denoiser could encounter, we compare the objective quality of the generated results in the graph 3.8. The first scene is the *DeleteMe* scene, simple primitives in levitation in front of colored mirrors. The second scene is the usual *Sponza* scene, with less illumination than usual. And the last one is the yellow *Dragon* statue.

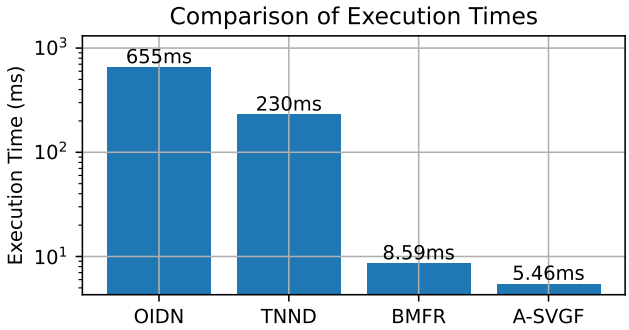


Figure 3.9: Execution time for the 3 presented denoising methods. Their configuration is the one that produces the image with the highest quality.

Figure 3.9 highlights that the three methods output images of similar qualities. However, as depicted in Figure 3.9, their execution significantly differs.

It is easy to notice that the three methods are quite even in terms of quality performance. What will separate them is the computational complexity (the actual time needed to generate the image), and the ease of implementation. The graph 3.9 displays the time needed to denoise the image. To measure the time needed to denoise an image, more than 10 frames were denoised in an image sequence, and using[18], the needed GPU time could be recorded.

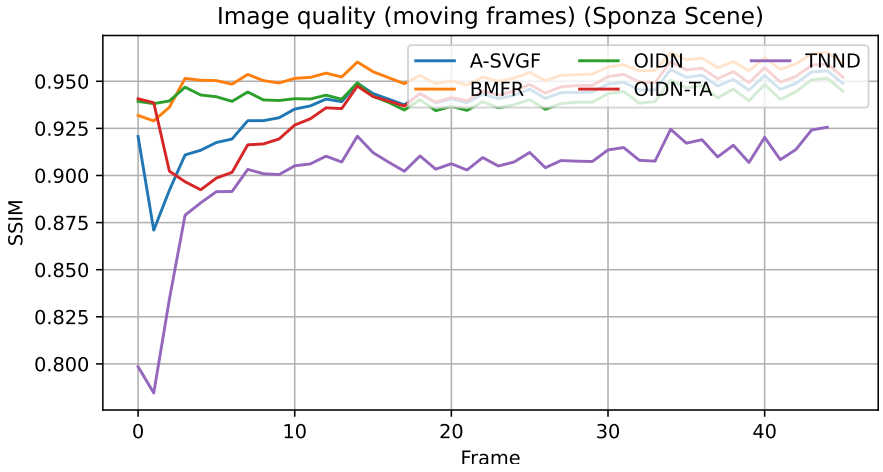


Figure 3.10

Concerning the temporal coherence of the produced images, previous scenes were rendered as an image sequence, with the camera rotating on itself or around the subject. Each image of the sequence is compared to the ground truth, an image rendered with the same camera configuration and 256 spp.

The sponza scene does not feature any kind of reflective material, and so, except for the first few frames, the results are pretty good for both presented methods, see 3.10. As they heavily rely on temporal accumulation to artificially augment the number of samples per pixel, it explains the somewhat slow start at the beginning.

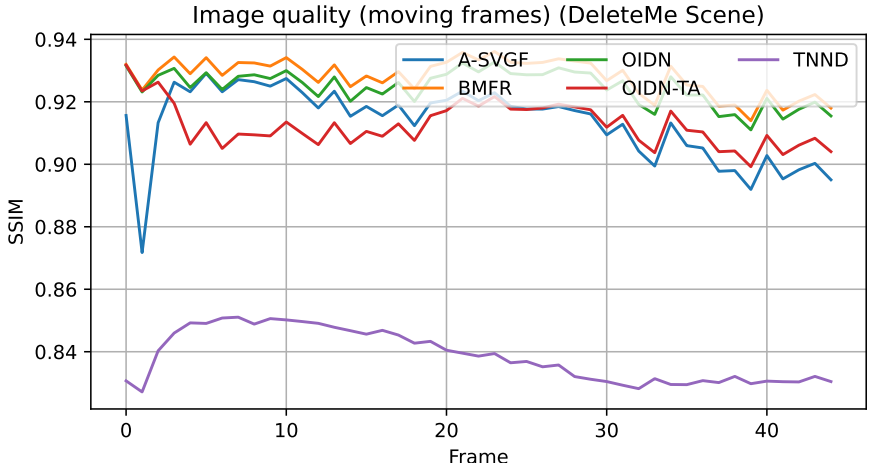


Figure 3.11

Using the *DeleteMe* scene, the scene with colored mirrors, the temporal stability, and the objective quality over time are analyzed, see 3.11. The slow start caused by the temporal accumulation is also noticeable.

As displayed in 3.11, TNND shows results that are not close to the state of art. It may be explained by the lack of scenes featuring hard reflection in the dataset used to train the TNND model. The results are still better than the noisy frame, but there is room for improvement.

Chapter 4

Conclusion

4.1 Consideration of the Results

The present work aimed at explaining, and discovering some reconstruction algorithms inside a carefully crafted path-tracing renderer. By exploring their results on multiple scenes, and modifying the different parameters, we were able to grasp their underlying working. By doing so, we explored the path towards a temporally stable, and high-fidelity implementation of a denoising solution. As expressed multiple times, the denoising step is not alone in a typical renderer. That is why we also developed and analyzed the needed dependencies that are inherent to a real-time path-tracer.

The adaptive part of A-SVGF was implemented, and the results were analyzed. Even if image quality is high, and that we are confident in the results, they're slightly lower than what was presented in the original works[29][4]. This is probably due to our different approaches in computing the actual gradient, reprojecting the actual seed necessary to keep the sample value across multiple frames, and the fact that we do not split the noisy frame into two components (specular and diffuse).

4.2 Future work

The equipment available, too old for the effective demands of real-time ray-tracing, did not allow testing the real conditions of the entire pipeline presented. This is why this master thesis only counts the time necessary for denoising as such. However, some more recent methods propose to move this crucial step further up the pipeline. Although interesting, including them in our work would make its "real-time" characteristic hardly verifiable due to the limitations encountered by our equipment. So one of the possible next steps would be to compare what we proposed with some alternative method.

By speaking of possible alternative methods, we first think of[32], and[34]. The first paper tries to track, at the path-tracer level, the cause of shadows, and reflections which are often the cause of visual artifacts. This would be a great improvement over the present accumulation that we presented. The second paper modifies slightly the rendering equation to extract two components (a noise-free, and a low-frequency) which facilitates greatly the denoising in itself.

Our renderer is built on top of ChameleonRT[31], using the *Embree* plug-in. Embree, in its current configuration, is exclusively capable of running on the CPU, forcing the renderer to transfer the diverse textures, and other resources back and forth, from the host memory to the GPU memory. It would have been smarter to keep everything on one side, to avoid the multiple data hazards we encountered throughout the implementation and the time needed to effectively copy the texture data.

The novel method, that we called TNND, inspired by a model capable of generating textures, presents mixed results and is somewhat below the current state of the art. We still believe that this approach explores a new way of approaching the problem of noisy image reconstruction and therefore, in this sense, still contributes something to the subject presented. Moreover, a real-time implementation on GPU is possible, given the very small number of parameters.

Appendix A

Mathematical Annex

Some notions were referenced multiple times in the present work and act as an important foundation for some of the conclusions or novelty of the different denoising methods. They are described in the present chapter to permit an easier understanding of complex mathematical concepts. Moreover, certain papers seem to consider them as "already known", which is probably not the case for a majority of the readers, and students who want to get a grasp of the subject related to denoising.

A.1 The Householder QR factorization

The BMFR denoising technique[19], which introduces a regression-based noise filtering approach, makes heavy use of the Householder QR factorization method to find a solution to the equation 2.6. Moreover, as written in the development, the Q matrix needed to realize the complete factorization, as in $T = QR$, is never computed, allowing an interesting speed-up. This section aims to explain why.

Remember that the least-square problem is defined as finding an approximate solution $\hat{x}$ that minimizes the error, the distance between $T\hat{x}$ and b. And so can be defined as:

$$\hat{x} = \min_{x \in R} ||Tx - b||^2$$

And with its solution using a QR factorization such that

$$R\hat{x} = Q^T b$$

For the sake of simplicity, $\hat{\alpha}^{(c)}$ has been renamed to b, and $Z^{(c)}$ to x. It is also assumed that T is full rank, and $W \times M$. In the development, we create an augmented matrix $\widetilde{T}$:

$$\widetilde{T} = [T, x]$$

As an example, this matrix T could be:

$$\widetilde{T} = \begin{bmatrix} 1 & 5.5 & 6.5 & 1.2 & 0.5 \\ 1 & 5.6 & 6.5 & 1.4 & 0.4 \\ 1 & 5.7 & 6.4 & 1.2 & 0.65 \\ 1 & 5.4 & 6.3 & 1.3 & 0.8 \\ 1 & 5.7 & 6.2 & 1.4 & 0.62 \\ 1 & 5.4 & 6.3 & 1.3 & 0.55 \end{bmatrix}$$

The Householder factorization is an iterative process consisting in computing a series of n matrices H_i such that $\hat{R} = H_n H_{n-1} \dots H_2 H_1 \widetilde{T}$ is an upper triangular matrix and $\hat{Q} = H_1 H_2 \dots H_{n-1} H_n$ is an orthogonal matrix. Each householder matrix H_i is constructed from the ith column of the matrix A. By definition, they are symmetrical, it is therefore valid to declare:

$$\hat{R}\hat{x} = \hat{Q}^T b$$

$$(H_n H_{n-1} \dots H_2 H_1 T)\hat{x} = (H_1 H_2 \dots H_{n-1} H_n)^T b$$

$$(H_n H_{n-1} \dots H_2 H_1 T)\hat{x} = (H_n^T H_{n-1}^T \dots H_2^T H_1)^T b$$

$$(H_n H_{n-1} \dots H_2 H_1 T)\hat{x} = (H_n H_{n-1} \dots H_2 H_1) b$$

If the augmented matrix $\widetilde{T} = [T, b]$ is used instead, the following assertions are valid too, as the Wth first columns remain the same between T and $\tilde{T}$.

$$\widetilde{T} = [T, b]$$

$$(H_n H_{n-1} \dots H_2 H_1)\widetilde{T} = [(H_n H_{n-1} \dots H_2 H_1)T, (H_n H_{n-1} \dots H_2 H_1) b]$$

The development ends up with an upper triangular matrix $\widetilde{R}$, of which it is possible to extract the necessary components to solve the original $\hat{x} = \min_x ||Tx - b||^2$. $\hat{R}$ being an upper triangular matrix $W \times (M+1)$ with zero padding: R a $W \times W$ matrix, and r a $W \times 1$ matrix are extracted from it.

$$\widetilde{R} = (H_n H_{n-1} \dots H_2 H_1)\widetilde{T}$$

$$\widetilde{R} = \begin{bmatrix} R & r \\ 0 & \end{bmatrix}$$

$$Rx = r$$

And that is how one can explain the claim "there is no need to compute $\widetilde{Q}^{(c)}$"[19]. It also explains "$r^{(c)}$ can be computed for different channels without recalculating R"[19], as $r^{(c)} = H_n H_{n-1} \dots H_2 H_1 z^{(c)}$ is computed from matrices already known and computed from previous channels.

Bibliography

[1] Barré-Brisebois, C., Halén, H., Wihlidal, G., Lauritzen, A., Bekkers, J., Stachowiak, T., and Andersson, J. Hybrid rendering for real-time ray tracing. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs* (2019), 437–473.

[2] Burns, C. A., and Hunt, W. A. The visibility buffer: a cache-friendly approach to deferred shading. *Journal of Computer Graphics Techniques (JCGT) 2*, 2 (2013), 55–69.

[3] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG) 36*, 4 (2017), 1–12.

[4] Christoph Schied, N. Q. T. Nvidia's implementation of rtx ray-tracing in quake ii. `https://github.com/NVIDIA/Q2RTX`, 2023.

[5] Dahlberg, H., Adler, D., and Newlin, J. Machine-learning denoising in feature film production. In *ACM SIGGRAPH 2019 Talks*. 2019, pp. 1–2.

[6] Dammertz, H., Sewtz, D., Hanika, J., and Lensch, H. P. Edge-avoiding a-trous wavelet transform for fast global illumination filtering. In *Proceedings of the Conference on High Performance Graphics* (2010), pp. 67–75.

[7] Deering, M., Winner, S., Schediwy, B., Duffy, C., and Hunt, N. The triangle processor and normal vector shader: a vlsi system for high performance graphics. *Acm siggraph computer graphics 22*, 4 (1988), 21–30.

[8] Georgiev, I., and Fajardo, M. Blue-noise dithered sampling. In *ACM SIGGRAPH 2016 Talks*. 2016, pp. 1–1.

[9] Haines, E., and Akenine-Möller, T. *Ray Tracing Gems: High-Quality and Real-Time Rendering with DXR and Other APIs*. Springer, 2019.

[10] Halton, J. H. A retrospective and prospective survey of the monte carlo method. *Siam review 12*, 1 (1970), 1–63.

[11] Horn, B. K., and Schunck, B. G. Determining optical flow. *Artificial intelligence 17*, 1-3 (1981), 185–203.

[12] Intel. Open image denoise. `https://github.com/OpenImageDenoise/oidn`, 2023.

[13] Jakob, W., Speierer, S., Roussel, N., Nimier-David, M., Vicini, D., Zeltner, T., Nicolet, B., Crespo, M., Leroy, V., and Zhang, Z. Mitsuba 3 renderer, 2022. https://mitsuba-renderer.org.

[14] Jiang, L., Dai, B., Wu, W., and Loy, C. C. Focal frequency loss for image reconstruction and synthesis. In *ICCV* (2021).

[15] Jiangping Xu, G. T. Bmfr-dxr-denoiser. `https://github.com/gztong/BMFR-DXR-Denoiser`, 2020.

[16] Kajiya, J. T. The rendering equation. In *Proceedings of the 13th annual conference on Computer graphics and interactive techniques* (1986), pp. 143–150.

[17] Kerr, A., Campbell, D., and Richards, M. Qr decomposition on gpus. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units* (2009), pp. 71–78.

[18] khronos group inc, T. Gl_arb_query_timer. `https://registry.khronos.org/OpenGL/extensions/ARB/ARB_timer_query.txt`, 2023.

[19] Koskela, M., Immonen, K., Mäkitalo, M., Foi, A., Viitanen, T., Jääskeläinen, P., Kultala, H., and Takala, J. Blockwise multi-order feature regression for real-time path-tracing reconstruction. *ACM Transactions on Graphics (TOG) 38*, 5 (2019), 1–14.

[20] Koskela, M., Immonen, K., Mäkitalo, M., Foi, A., Viitanen, T., Jääskeläinen, P., Kultala, H., and Takala, J. Blockwise multi-order feature regression for real-time path tracing reconstruction. `https://github.com/maZZZu/bmfr`, 2019.

[21] Kramer, M. A. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal 37*, 2 (1991), 233–243.

[22] Mordvintsev, A., and Niklasson, E. Nca: Texture generation with ultra-compact neural cellular automata. *arXiv preprint arXiv:2111.13545* (2021).

[23] Mordvintsev, A., Randazzo, E., Niklasson, E., and Levin, M. Growing neural cellular automata. *Distill 5*, 2 (2020), e23.

[24] Ravishankar, S., and Bresler, Y. Mr image reconstruction from highly undersampled k-space data by dictionary learning. *IEEE transactions on medical imaging 30*, 5 (2010), 1028–1041.

[25] reality, V., and group, G. A. V. Blockwise multi-order feature regression for real-time path tracing reconstruction: Dataset. `http://urn.fi/urn:nbn:fi:att:4439207b-764d-43b2-9bea-2dcc36f28256`, 7 2019.

[26] Rotella, F., and Zambettakis, I. Block householder transformation for parallel qr factorization. *Applied mathematics letters 12*, 4 (1999), 29–34.

[27] Rusinkiewicz, S. A survey of brdf representation for computer graphics, cs348c, 1997.

[28] Ryan, J. Froggy opengl engoodener. `https://github.com/JuanDiegoMontoya/Fwog`, 2023.

[29] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In *Proceedings of High Performance Graphics*. 2017, pp. 1–12.

[30] Schied, C., Peters, C., and Dachsbacher, C. Gradient estimation for real-time adaptive temporal filtering. *Proceedings of the ACM on Computer Graphics and Interactive Techniques 1*, 2 (2018), 1–16.

[31] Usher, W. ChameleonRT. `https://github.com/Twinklebear/ChameleonRT`, 2019.

[32] Zeng, Z., Liu, S., Yang, J., Wang, L., and Yan, L.-Q. Temporally reliable motion vectors for real-time ray tracing. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 79–90.

[33] Zhao, H., Gallo, O., Frosio, I., and Kautz, J. Loss functions for image restoration with neural networks. *IEEE Transactions on computational imaging 3*, 1 (2016), 47–57.

[34] Zhuang, T., Shen, P., Wang, B., and Liu, L. Real-time denoising using brdf pre-integration factorization. In *Computer Graphics Forum* (2021), vol. 40, Wiley Online Library, pp. 173–180.

[35] Zirr, T., Hanika, J., and Dachsbacher, C. Re-weighting firefly samples for improved finite-sample monte carlo estimates. In *Computer Graphics Forum* (2018), vol. 37, Wiley Online Library, pp. 410–421.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**