Graphical Models 129 (2023) 101184

Contents lists available at ScienceDirect

Graphical Models

journal homepage: www.elsevier.com/locate/gmod

Non-homogeneous denoising for virtual reality in real-time path tracing rendering

Victor Peres [a,*], Esteban Clua [a], Thiago Porcino [b], Anselmo Montenegro [a]

[a] Universidade Federal Fluminense, Av. Gal. Milton Tavares de Souza, s/n, Niterói, Rio de Janeiro, 24210-346, Brazil
[b] Dalhousie University, 6299 South St, Halifax, Nova Scotia, B3H 4R2, Canada

ARTICLE INFO

Keywords:
Foveated rendering
Dual-screen systems
Head-mounted devices
Denoising

ABSTRACT

Real time Path-tracing is becoming an important approach for the future of games, digital entertainment, and virtual reality applications that require realism and immersive environments. Among different possible optimizations, denoising Monte Carlo rendered images is necessary in low sampling densities. When dealing with Virtual Reality devices, other possibilities can also be considered, such as foveated rendering techniques. Hence, this work proposes a novel and promising rendering pipeline for denoising a real-time path-traced application in a dual-screen system such as head-mounted display (HMD) devices. Therefore, we leverage characteristics of the foveal vision by computing G-Buffers with the features of the scene and a buffer with the foveated distribution for both left and right screens. Later, we path trace the image within the coordinates buffer generating only a few initial rays per selected pixel, and reconstruct the noisy image output with a novel non-homogeneous denoiser that accounts for the pixel distribution. Our experiments showed that this proposed rendering pipeline could achieve a speedup factor up to 1.35 compared to one without our optimizations.

1. Introduction

Path tracing is a consolidated rendering approach for achieving photo-realistic graphics, but currently not suitable for real-time performance. Shadows, global illumination, and reflection are essential graphical effects that are intrinsically achieved through it. Additionally, since its complexity is directly related to the number of pixels of the screen, using it on high-resolution devices is challenging, requiring non-trivial solutions to be used on head-mounted displays (HMDs).

Furthermore, path tracing is the current state-of-the-art in real-time rendering games, interactive rendering in graphical applications, and pre-rendering films. Recently, Nvidia launched RTX GPUs, embedded with hardware acceleration related to path tracing [12], enabling an increasing number of consumers and workplaces to leverage these optimizations for real-time applications.

Even so, it is still laborious and time-consuming for the graphical processors to render at high resolutions. Therefore, performance optimizations are required, such as hardware-based Bounding Volume Hierarchy (BVH) structures, ray-polygon intersections computed in hardware, and better sampling of the traced pixels. Denoising is one of the most important optimization methods. It is used to reduce the variance in a noisy image produced by the path-tracing rendering on a low ray sampling. In the literature, it is possible to find different denoising techniques such as wavelet filters [6], bilateral filters [18], machine learning algorithms [10] and sampling with spatio-temporal accumulation [21], to name a few.

Foveated rendering is an important topic and is related to the fact that the human eye can only distinguish important details of pigments in a central region of the retina, called fovea [20]. This paper presents a novel approach to reconstruct path-traced rendered environment images in real-time by leveraging the perception characteristics at the center of the human vision. The proposed rendering pipeline splits the path-traced noisy image in different concentric layers and apply denoising strategies on each layer differently, with parameter values that are adapted according to the different regions of the visual field.

We tested our solution using a collection of scenes with different triangles count, different display resolutions corresponding to the entirety of the dual screens of a regular HMD, different spatial sampling configurations for the path-traced step, and different reconstruction layer parameters. Our method achieved notable speedup improvements in all configurations when compared to non-optimized implementations.

* Corresponding author.
E-mail addresses: victorperes@id.uff.br (V. Peres), esteban@ic.uff.br (E. Clua), thiago@dal.ca (T. Porcino), anselmo@ic.uff.br (A. Montenegro).

https://doi.org/10.1016/j.gmod.2023.101184

Available online 8 August 2023

2. Related works

Among different methods available for path-tracing rendering, there are interactive rendering, offline rendering and real-time rendering. Besides that, we acknowledge the advancement of other optimization techniques that fall into different categories, such as machine learning-driven filters or sampling through spatio-temporal accumulation. This work focus on real-time reconstruction techniques, such as bilateral kernel filtering with geometric data in the form of G-Buffers.

Our work combines these features to improve path tracing within dual-screen systems, taking advantage of reduced density of pixel required in peripheral display regions. This section describes the basics and related works associated with its proposal.

2.1. Path tracing

Whitted [27] introduced the first ray-traced image with not only shadows directly projected from a light source but also including a series of additional graphical effects, such as specular reflection and refraction, simulating the light transport with optical properties from the Fresnel equations. Later, Cook [4] introduced a random distribution of the rays sampled in the hemisphere of a reflected surface.

Kajiya introduced a novel form of ray tracing, named path tracing, which solved the exponential problem that the nature of ray tracing methods introduced [9]. It combined concepts of Monte Carlo integration and, instead of shooting a set of rays for each ray bounce, it shot only one ray for each ray bounce. Given this property, it is possible to achieve a high level of graphical realism with all the mentioned visual effects and global illumination features in a much faster way. This is made through several samples per pixel, i.e., the lighting factor of an object that is perceived with the light rays bouncing from different surface sources.

Hybrid approaches such as Barre et al. [1] were introduced to further improve performance optimizations by leveraging the advantages of rasterization, ray-tracing, and compute shaders. It uses rasterization in G-Buffer, direct shadowing stages and ray-tracing in other steps such as direct, indirect lighting, and real-time reflections. It also uses compute shaders in the post-processing stages.

To guarantee that the path-tracing algorithm via Monte Carlo integration produces a result that converges into a photo-realistic image, it is necessary hundreds (if not thousands) of samples per pixel to achieve an almost perfect image. Today, the most demanding scenes can handle at most a few sample per pixel (spp) with global illumination. This generates an image with Monte Carlo variance noise and, due to the nature of Monte Carlo integration, this noise is decreased in a square-root proportion to the number of samples. Hence, there is a need to reconstruct this non-perfect noisy image using properties from the scene's geometry.

2.2. Denoising

Several algorithms and techniques are available for reconstructing images in real-time. Methods related to machine learning techniques and neural network training have recently gained attention. Some of these works use the concept of autoencoders, which are being popularized due to their adoption by the graphics industry, such as Intel Open Image Denoise [8] and Nvidia Optix Autoencoder [16]. There are also techniques that use concepts of image analysis and processing, such as convolution filters. These are constituted by bilateral filters, which can produce simple results with some graphical artifacts of brightness change or blur. They can have a bad computational performance when the size of the filters is big, becoming inefficient for the use in path tracing with sampling smaller than one spp and real-time requirements.

Filters guided by buffers progressively reduce the artifacts. More specifically, the filter introduced by Dammertz et al. [6] is known as Edge-Avoiding À-Trous. This filter satisfactorily fills the noisy image, being capable of avoiding almost any artifacts, which makes this suited for filters with big kernel size and spaced with smaller sampling size. Further on this, there is another Edge-Aware filter, proposed by Qi et al. [19], that improves this approach and is capable of decreasing the atmospheric fog and the haze of an image with filtering with a decreasing step in each iteration of the process. Although the original problem was not related to rendering, these results may also be applied to reducing the same artifacts present in a noisy image.

Other works use similar filtering concepts, such as the works produced by Schied et al. [21,22], which uses a guided filter with spatio-temporal variance and temporally accumulates the samples using accumulation buffers from previous frames in a way that moving objects are accounted for the global illumination.

Regression-based techniques with QR factorization are recently showing promising results due to their high performance for real-time denoising in path tracing pipelines [13]. Zwicker et al. [29], and Kaplanyan et al. [11] present more details and discussions related to the topic.

2.3. Foveated rendering

Foveated rendering divides the rendering areas into different regions, using specific and separated rendering parameters for each region. Guenter et al. [7] split the image into three distinct layers with varying rates of sampling following the user gaze, enabling lower selection in the layer far from the center of the fovea gaze due to its small retina cell cones density. Their work is relevant due to the introduction of the layers based on the user's gaze, even though it uses rasterization as its rendering method. It showed with benchmarking that foveation rendering achieves a speedup with a factor of 5-6x of a non-foveated rendering on desktop displays [7].

Weier et al. [26] introduced the idea that a linear falloff is more suitable when dealing with ray tracing for HMD, in comparison with the previous hyperbolic falloff, due to the motion perception in the periphery vision area [26]. Since it used only direct lighting with point lights, area lights, or ambient occlusion shading, it missed global illumination and reflection effects. Their proposal achieved different speedups ranging from 1.46 to 4.18 depending on the rendering configuration.

Our work improves real-time performance optimizations by introducing a novel concept of a non-homogeneous type of path-traced image reconstruction, using denoising filtering algorithms with different levels according to the linear falloff of the center of the user's gaze and rendering it in a path tracing environment with a pre-determined ray sample distribution.

3. Non-homogeneous denoising

This work proposes a novel rendering pipeline suitable for path-tracing techniques running on HMDs. Although the proposed solution is hardware-agnostic, we still need a GPU-based system due to denoising requirements. This section describes the rendering pipeline running on both GPU and CPU. The pipeline is illustrated in the Fig. 2 and involves four steps: (1) the pre-pass that computes feature buffers; (2) the foveation distribution step, that computes the pixels that will be sampled in the following passes; (3) the path tracing pass that computes the lighting for including graphical effects and (4) the non-homogeneous denoising applied onto a noisy path traced image being rendered in two screens, with its foveation adjustment.

3.1. Pre-pass

The first stage consists of a set of ray-tracing shaders that generates one sample of ray for each pixel (spp) in both sides of the screen (one for the left and one for the right eye). Once the traced ray hits a surface, the shading data of the point intersected is computed. The calculated

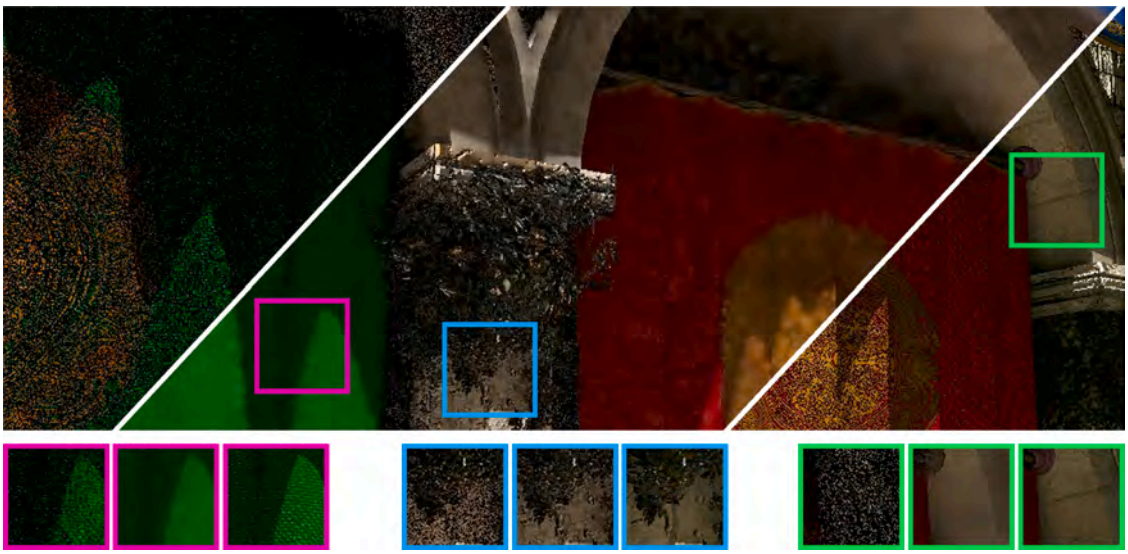

Fig. 1. In the top, from left to right: the foveated noisy input to the non-homogeneous denoiser, the denoised image output using our proposed three-layer foveated configuration, and the reference rendered image with 1024 samples per pixel. In the bottom, the cropped sections corresponds to the three distinct layers of the non-homogeneous denoiser in the same order as the top figure: noisy input, denoised output, and reference image.

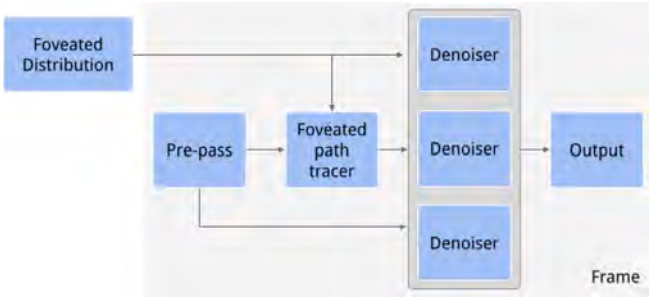

Fig. 2. Our proposed rendering pipeline: the foveated distribution is outside the frame because it does not change between frames and remains the same across the application execution.

shading data for each pixel can be seen in Fig. 3, with the diffuse component of the texture (a); the position in world coordinates (b); the normal of the surface in world coordinates and the position in world coordinates (c). The result of this computation is stored in buffers (G-Buffers), required for the subsequent passes, including direct lighting calculation, global illumination, and denoising.

3.2. Foveation distribution step

In order to take advantage of the fovea distribution, in this stage we compute a buffer with only the pixels that are being selected to be sampled with rays in the further step of ray generation. This

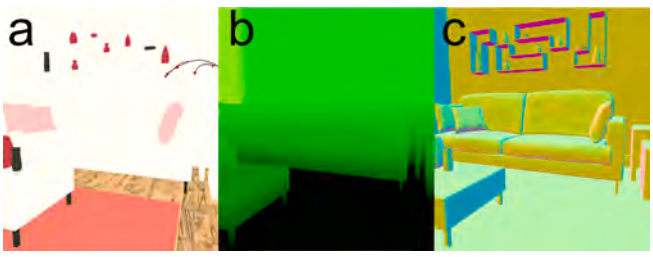

Fig. 3. Feature buffers rendered in the left screen in order of calculation, from left to right: diffuse, position and normal.

computation is constant for all frames and independent of the time or the scene. For this reason, we choose a CPU buffer instead of a GPU. In that sense, for coordinates distribution, we used only one buffer. Moreover, it was used in the left and right eyes, disregarding any possible mismatch of the cones' distribution between one eye and the other.

Using concepts similar to those proposed by Weier et al. [26], we build a three-layer distribution that is represented by concentric circles of pixels with different sampling decay, in a linear proportion in each layer. Even though previous studies have already shown that the decay of cones in the view falls in a hyperbolic distribution [7], the linear proportion is preferred for foveated rendering due to the movement in the scene around the periphery area of the vision.

The first layer represents the fovea and has a radius r_0. It is defined in such a way that the samples are taken at full resolution, i.e., each pixel will receive exactly 1 sample, meaning that it is the same spatial sampling that occurred in the G-Buffers computation in the pre-pass step. The inner radius of the second layer where the r_0 ends and is denoted as $r_0 + 1$. r_1 is its outer radius. Thus, it is the first layer outside the fovea layer. As in previous works [26], a linear proportion decay is applied in this layer, starting from $r_0 + 1$ and ending in r_1, parametrized with the probability p. This gives the sub-sampling effect of the middle layer, with each pixel having a probability p of being sampled according to the distance of the pixel to the center of the fovea. The third layer starts from radius $r_1 + 1$ and ends in the edge of the viewport, being the layer outside the second layer. It receives even less sampling than the second layer values, with a constant probability of $1 - p$ of a pixel being sampled by a ray.

Once the computation of foveation distribution is complete, the screen coordinates of the pixels are stored in the coordinates buffer. We also store this computation in a binary mask for the denoising pass to access the painted coordinates in constant time, as shown in Fig. 4. Thus, the output of this step are both the coordinates buffer with the size of the number of pixels that are being sampled, and the binary mask, containing the positions of the sampled pixels decided by the foveated distribution with linear decay, through the probability p. This step is necessary for the posterior path tracing pass in each one of the screens of the HMD and for the denoising pass.

3.3. Path tracing pass

The following step computes the direct shadows and the global

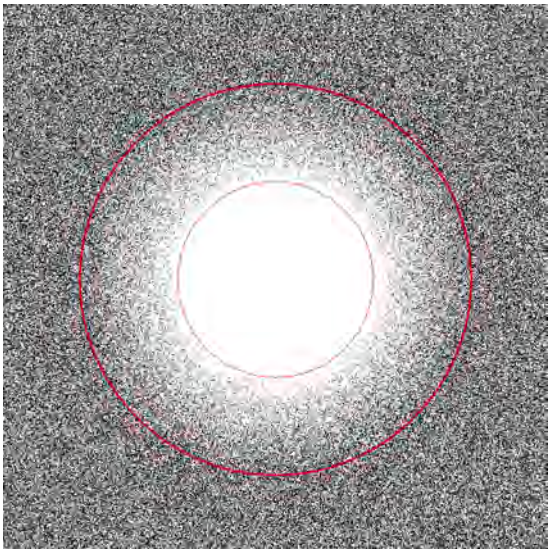

Fig. 4. The foveated distribution output, represented in a binary mask, rendered with the concentric circles delimiting the range of each of the three layers, with the outer layer being delimited by the borders of the screen.

illumination. Using the previously computed coordinates buffer, the set of path-tracing shaders generates rays with the coordinates defined by this buffer. This is used for generating rays that will pass through them. Thus, when taking into account the rendering based on visual acuity advantage, we maintain the sub-sampling property of this render pass, with a density smaller than the full sampling less than 1 spp for the middle and outer layers. Through this foveated rendering technique, it is possible to decrease the number of generated rays through a configurable proportion given by the size of the coordinates buffer in the previous step.

For each pixel, we compute the global lighting, direct lighting, and shadow effects, using two groups of shaders: the shadow group and the indirect lighting group. For the shadow group, when the generated ray hits a point, it shoots a visibility ray to a random light in the scene. For the indirect lighting group, when the generated ray hits a point, it shoots an indirect ray to compute its color, along the standard secondary shadow ray for its bouncing factor. For lighting and the BRDF, we used the Lambertian material [17].

It is worth noting that instead of two ray-tracing programs or instance of programs running, there is only one program that runs for both the screens. This means that the same program and shader groups are executed for the screen representing the left and right eyes. The difference between the shading of the screens is that, when generating a ray and defining where it is on screen coordinates, it must account for an offset, given by the width of the leftmost screen. By designing the path-tracing step in this way, there is a possibility for increasing the ray coherence from both cameras and, thus, an improvement in the memory locality of the BVH computations. With this render pass, the result is a noisy output texture representing the path traced algorithm applied in the scene accounting for the foveation distribution, as shown in Fig. 5.

3.4. Non-homogeneous denoising pass

The non-homogeneous denoising pass is responsible for the image reconstruction. This stage receives the output from the path-tracing pass with the noisy image, along with the G-Buffers and the binary mask as an input. After the pass finishes, the output is the denoised image with less variance than the path-traced image, as shown in the Fig. 6, which is in fact a reconstruction of the noisy output of the path-traced image in the Fig. 5. Note that the denoised image is lighter because the ”darkness” is created by a higher variance in the noisy input.

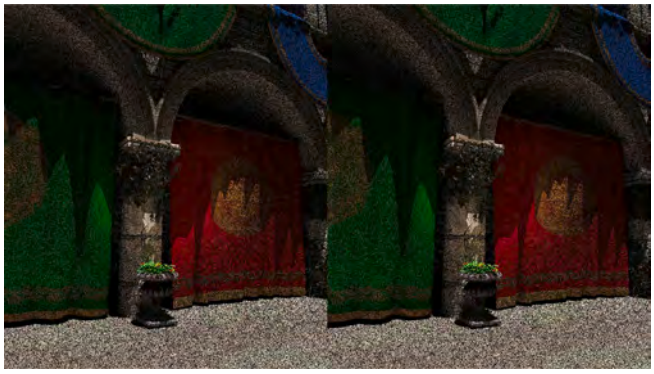

Fig. 5. Generated image after the path-tracing pass in the Sponza scene experiment, for the left and the right sides of the screen. The path tracing pass generates a noisy output for both screens using the G-Buffers and the coordinates buffer as an input, accounting for the foveated distribution.

Fig. 6. The reconstructed image from Fig. 5 after the non-homogeneous denoising pass, in the Sponza scene experiment.

Since the foveated noisy image has different sampling densities for every layer with varying amount of noise, there is a need to split the denoising process in accordance to this property. This way, we differentiate the image in three layers, using the same layers used in the foveation distributed step, as shown in Fig. 4. The pass applies the denoising process for each layer using the Edge-Avoiding Á-Trous filter [6] adapted to account for the foveation distribution. Since the filter works by weighting the noisy image and both the world normal and world position G-Buffers input against the neighbors' pixels, we need to filter with the binary mask as input. This mask works like a selection filter that decides which neighbor pixel is going to be accumulated for the final color of the current pixel being iterated. This works in a way that the sub-sampling does not darken and further increases the variance of the shaded pixel. Thus, from the original weight function

$$w(i,j) = w_{rt} * w_n * w_x \tag{1}$$

with pixel positions i and j, where w_rt is the weight of the path-traced color, w_n is the weight of the world normal, w_x is the weight of the world position. The modified version is

$$w(i,j) = w_{rt} * w_n * w_x * b(i,j) \tag{2}$$

with the added binary mask $b(i,j)$ at the pixel.

By processing the denoise pass only once on both sides of the screen, it is not possible to achieve a satisfactory noise reduction in the inner layer and let alone the middle and outer layers, so that each one requires a different number of iterations in proportion to the respective sampling. Among the three layers foveation distribution and denoising configuration, the inner layer starts from the center of the screen to the radius r_0

and requires M levels of iteration, the middle layer starts from radius r_0 +1 to the r_1 and requires N levels, and the outer layer starts from radius $r_1 + 1$ to the border of the viewport and requires P levels.

The outer layer needs more denoising iterations in comparison to the other inner layers, as it has fewer samples than these layers. Likewise, the middle layer requires more denoising than the inner layer for the same reason. Thus, the iterations of denoising must follow the inequality $M < N < P$ in our non-homogeneous denoising.

Each iteration in a layer reconstructs only the area corresponding to that layer. It does so with a 5 × 5 kernel using a convolution mask based on the same cubic B-spline as described in [6]: $(\frac{1}{16}, \frac{1}{4}, \frac{3}{8}, \frac{1}{4}, \frac{1}{16})$, with the difference that it is extended to a two-dimensional kernel. Thus, in the borders of each layer, the neighbor pixels from other layers may be selected for the accumulation of the final shading color. Except in two cases: the outer layer, where the neighbor pixels could be out of screen, and the pixels that were not selected in according to the binary mask, that are not supposed to be accumulated due to the fact that they were not sampled. With this, we can optimize the filter to be more efficient by leveraging the sub-sampling configuration from the foveated distribution.

To reduce the artifacts from the Edge-Avoiding À-Trous filter, we use the modified version with a decreasing step-width [19], instead of an increasing step-width in subsequent iterations as illustrated in Fig. 7.

The output of this pass is the final image that the user will see in each frame being rendered. This pass is described by Algorithm 1.

Input: M, N, P, binary mask B, path-traced texture rt, world normal texture n, world position texture x.
Output: Denoised texture c
Function `computeDenoise`$(step, r_a, r_b)$:
  foreach *pixel* (i, j) *in the texture* rt *from* r_a *to* r_b **do**
    $sum \leftarrow 0$
    $cum_w \leftarrow 0$
    foreach *selected neighbour* (u, v) *in the texture* rt *at the distance* 2^{step} **do**
      $w(u, v) \leftarrow w_{rt} * w_n * w_x * B(i, j)$
      $sum \leftarrow sum + c(u, v) * w(u, v) * kernel(u, v)$
      $cum_w \leftarrow cum_w + w(u, v) * kernel(u, v)$
    end
    $c(p, q) \leftarrow sum/cum_w$
  end
Function `Main`:
  for $i \leftarrow M - 1$ **to** 0 **do** computeDenoise(i, 0, r_0);
  for $j \leftarrow N - 1$ **to** 0 **do** computeDenoise(j,r_0, r_1);
  for $k \leftarrow P - 1$ **to** 0 **do** computeDenoise(k, r_1, r_{max});
  return c

ALGORITHM 1. Non-homogeneous denoising.

4. Implementation and results

This section details our implementation and denoising configuration. We also explore our experiments, based on different configurations of input parameters. Finally, a brief discussion of the benchmarks is made, with a speedup analysis.

4.1. Implementation

This work implemented the proposed rendering pipeline and its ray-tracing programs using the Microsoft DirectX APIs for the graphics shaders, the Valve OpenVR APIs [24] for the integration with the HMD, and imGui [5] for the GUI integration for the experiments. All of these libraries and APIs are inside the Falcor framework, developed by NVIDIA [2].

The G-Buffers are computed based on [28] implementation, with modifications in order to support two screens. This modification is done to support each one with a different view matrix representing the left and the right eye. With these two different matrices, the ray-generation shader traces a ray for each side of the screen, with reference to the pixel in each screen and its corresponding view matrix. It enables the computation of the G-Buffers of both screens by the end of a single rendering pass of the ray-tracing shaders, i.e., with a single run of the ray-tracing program instead of running the same program for each side of the screens separately. This improves the memory utilization of the triangle-ray intersection calculations in the RTX GPU, since there is ray coherence in both of the different cameras.

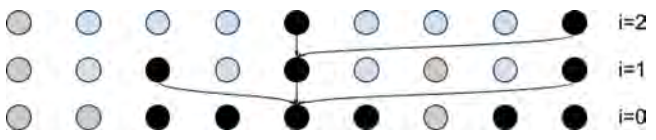

Fig. 7. Illustration depicting an example of the À-Trous filter with 3 levels of iterations with decreasing step-width. It starts by taking into account the pixels at distance 2^i (black dots) from the center pixel in the accumulation process and decreases for each subsequent iteration. The blue dots are skipped by the À-Trous filter and the gray dots are skipped by the binary mask modification. (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

The computation of the foveation distribution, is made in a CPU function before loading the scene, which is passed to the GPU via a constant buffer. This way, we can guarantee that there is no impact in the total performance of the rendering pipeline. The computation consists of the iteration over every pixel on the left side of the screen, verification in which region one pixel belongs to, and then selection of the pixel. In the inner layer, every pixel is selected. In the middle and outer layer, a random toss with the probability p set to $p = .5$ define which pixel is going to be selected, following the linear decay mentioned in the previous section. In the current implementation, we set this probability for both Full HD (960 × 1080 per screen) and near 4K Quest 2 (1832 × 1920 per screen) resolutions. This results in the sample distribution cited in Table 1. For the lack of eye-tracking devices in our experiments, we also consider that the user's gaze is in the center of the screen. After the computation of the foveated distribution process, we fill both the coordinate buffer and the binary mask.

The path-tracing pass is a modified version based on an existing implementation [28] of the diffuse illumination using a Lambertian model of reflection for the objects of the scene and also used for global illumination. In this case, the modification was in order to generate the initial rays based on the pixel buffer size and to trace the rays only through the coordinates given by the buffer. With this modification, we were able to generate only a controlled amount of rays in the initial ray generation shader, given the foveated distribution calculated in the previous step.

The implementation was tested with different spp densities, including the configuration that achieved satisfactory performance in the current hardware, fixed in 4 spp density for the inner layer. These path tracing shaders, such as the G-Buffers shaders, were also modified so that they could compute the lighting in both screens in a single pass. Since the G-Buffers already account for both sides of the screens, this modification in the path tracing step was simplified, and it resumes to a

Table 1
Region distribution used in the experiment for comparison, with its resolution, respective area size, samples, and spp density.

Resolution	Region	Radius	Area	Samples	spp
FHD	Inner	144	65144	260576	4
	Middle	288	195432	586296	3
	Outer	-	776224	1552448	2
Quest 2	Inner	265	220618	882472	4
	Middle	530	661855	1985565	3
	Outer	-	2634967	5269934	2

mapping of the coordinates buffer to the pixel launch coordinates in the ray-generation shader. For the right side of the screen, this mapping also accounts for an offset related to the width of the left screen so that the pixel launch coordinates are correctly shifted to dispatch the rays in the GPU shader.

Our denoising step consists of iterating over the levels of denoising and dispatching a compute task to the GPU. As the compute shader is the same for all three layers, we only pass the required arguments, such as inner radius, outer radius, and the current step width. This implementation also includes an adaptation and translation of the GLSL shader provided by [6] with some modifications. We changed the loop to a version introduced by Qi et al.[19], with a decreasing step-width in each subsequent iteration of the levels of the execution of the shader, in order to reduce the artifacts near the edge of the scene objects. Here, we also included the modification in the path-tracing pass to account for both sides of the screen. Besides that, since this reconstruction was done in a compute shader, we can dispatch a separate task for every region. The computation of the denoising is done so that each thread calculates the weighted sum of its neighboring pixels in both sides of the screen, by implementing an offset of the screen width. We avoid the pixels that should not be selected in either of the regions by selecting through our binary mask from the foveated distribution.

4.2. Results

To present and discuss the results of this work, a series of experiments using different configurations was performed in a single hardware platform. We ran our experiments through an implementation with a machine with the specifications: Intel Core i7-3770S CPU @ 3.10GHz, 8.00 GB RAM, NVIDIA GeForce RTX 2080.

We choose to measure the performance of our rendered pipeline using the metric of time, in milliseconds, that a frame is rendered in the GPU. This is so that we can diminish any possible interference from the varying OS system load or other background processes running on the machine. We render the scenes with the camera traversing a pre-defined path so that we can observe beyond a static frame of a single image. We measure the results over 1000 subsequent frames and take the average metric for each scene. Our target resolutions were set to Full HD (960–×–1080 per screen, totaling 1920 × 1080) and "near-4K" from the Quest 2 HMD (1832 × 1920 per screen, totaling 3664 × 1920).

Three different scenes were used in our experiments, with varying specifications of triangles and light count, as detailed in Table 2.

The first experiment is referenced as *Base* for our benchmark comparison, since it is our baseline for speedup calculations. This configuration was chosen as a reference for the performance analysis because it does not include neither a foveation distribution nor our non-homogeneous denoising. Instead, the pipeline in these *Base* experiments renders a full-screen denoising pass to a path-tracing with sampling of 4 spp across the entirety of the dual-screen viewport.

In the subsequent experiments referenced as Non-Homogeneous 3-Layer Denoising (*NH3LD*), we applied the proposed rendering pipeline optimizations. The foveation distribution has three layers as concentric circles, and also coincides with the regions split for the non-homogeneous denoising pass. Following the split in Table 1, for the sake of comparison, we also set the inner layer with a path-tracing sampling density of 4 spp, decreasing the sampling in the middle and outer layers.

We also consider that the outer layer in the *NH3LD* experiment has the greater number of denoising levels. Analyzing the averaged millisecond per frame performance in our experiments, we can compare it to the *Base* experiment. In the same level of denoising, the proposed solution with the non-homogeneous denoising optimization applied to three layers shows a speedup of up to 1.35.

A third experiment tested a different split in the non-homogeneous denoising pass, using two layers with different levels of denoising instead of the three-layer design used in the previous experiment. We reference it as Non-Homogeneous 2-Layer Denoising (*NH2LD*). Note that the foveated distribution still has the three-layer as described in Table 1. For the denoising split, the inner layer coincides with the foveated distribution as well. The outer layer is a join region with the middle and outer layer from the foveated distribution. In other words, the outer layer in the denoising split begins from the circles with radius $r_0 + 1$ and goes until the edge of the viewport.

We again consider that the outer layer in the *NH2LD* experiment has the greater number of denoising levels. Analyzing the performance in our experiments with the configuration of the same level of denoising in the first experiment, it is possible to see that the proposed solution with the non-homogeneous denoising optimization applied to two layers achieves a speedup of up to 1.33. The detailed performance measurements with the total averaged milliseconds per frame and the speedup are in Table 3.

For an objective quality analysis, we rendered the same scenes without the foveation distribution step optimization. We also increased the number of samples for 1024 spp temporally accumulated across several frames, where no denoising was applied to these reference images. In order to measure the error between the reference rendered images and the optimized by our rendering pipeline, we used Root-mean-square error (RMSE), Structural Similarity (SSIM) [25], and Peak signal-to-noise ratio (PSNR) metrics.

We also measured the error between the *Base* experiment and the referenced image. This is made in order to figure how our non-homogeneous denoising is performing quality-wise in comparison to a full-screen denoising.

As show in Table 4, we see a minor variance in the metrics of both our experiments of non-homogeneous denoising in comparison to the full-screen denoise in the *Base* experiment.

5. Conclusion

This work presented a novel real-time rendering pipeline for virtual reality devices, using non-homogeneous denoising scaled according to foveated regions. Our proposed pipeline was able to achieve the same

Table 2
Scenes tested and its characteristics.

Scene	Triangle Count	Light Count
Pink Room	786056	1 directional; 2 point lights
Sponza	262267	1 directional; 1 point lights
Forest	198541	80 point lights

Table 3
Performance metrics of Base, NH2LD and NH3LD experiments, in averaged milliseconds per frame for our GPU implementation.

Resolution	Scene	Renderer	Total time (ms)	Speedup factor
FHD	Pink Room	Base	16.26	
		NH2LD	13.99	1.16
		NH3LD	12.81	1.26
	Sponza	Base	31.01	
		NH2LD	23.28	1.33
		NH3LD	22.95	1.35
	Forest	Base	22.84	
		NH2LD	18.46	1.23
		NH3LD	17.54	1.30
Quest 2	Pink Room	Base	45.32	
		NH2LD	40.58	1.11
		NH3LD	39.19	1.15
	Sponza	Base	99.27	
		NH2LD	75.52	1.31
		NH3LD	74.62	1.33
	Forest	Base	73.66	
		NH2LD	60.24	1.22
		NH3LD	59.03	1.24

Table 4
Objective quality comparison with RMSE, SSIM and PSNR metrics.

Scenes	Rendering Pipeline	RMSE (%)	SSIM	PSNR (dB)
Pink Room	Base	3.2862	0.986438	28.659183
	NH2LD	3.4374	0.985774	28.030232
	NH3LD	3.5997	0.987343	27.619445
Sponza	Base	5.7633	0.918678	21.879093
	NH2LD	5.9252	0.913988	21.420517
	NH3LD	5.9029	0.913633	21.468767
Forest	Base	3.7991	0.813191	22.339938
	NH2LD	3.8649	0.795324	22.37879
	NH3LD	4.1787	0.778536	21.69204

effects of visual realism achieved when using regular denoising, but reducing the number of rays and increasing performance. We leveraged the foveation distribution created in the CPU, stored and passed to the GPU in a coordinates buffer with different spatial sampling in each layer of the visual field. This was important to decrease the initial ray generation in the GPU shader and its bounces. Adding non-homogeneous denoising also enabled a decrease in the load of work for the reconstruction steps, with different levels to apply the Edge-Avoiding À Trous [6] algorithm for the corresponding denoising layers.

Benchmarks were run with this optimized pipeline in several configurations against a non-optimized one. These configurations used different implementation details such as the size of layers, number of layers, and levels of denoising. Experiments were able to show a speedup in rendering time performance of up to 1.35.

In future works, we intend to explore how to apply the non-homogeneous denoising with other algorithms. Also extend it by conducting a user study to test the human perception of the graphical quality through different implementation settings, to further optimize our denoising and path-tracing parameters. There is the possibility to use another type of rendering system such as mapping coordinates to log-polar [15] or Visual-Polar [14] space before the start of the rendering pipeline. While Lambertian is a good approach to diffuse surfaces, we would like to include other types of materials that encompass specular surfaces, such as GGX [3,23].

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the data in the files attached.

References

[1] C. Barré-Brisebois, H. Halén, G. Wihlidal, A. Lauritzen, J. Bekkers, T. Stachowiak, J. Andersson, Hybrid rendering for real-time ray tracing. Ray Tracing Gems, Springer, 2019, pp. 437–473.

[2] N. Benty, K.-H. Yao, T. Foley, M. Oakes, C. Lavelle, C. Wyman, The Falcor rendering framework, 2018, https://github.com/NVIDIAGameWorks/Falcor.

[3] B. Burley, W.D.A. Studios, Physically-based shading at disney. ACM SIGGRAPH volume 2012, vol. 2012, 2012, pp. 1–7.

[4] R.L. Cook, T. Porter, L. Carpenter, Distributed ray tracing. Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques, 1984, pp. 137–145.

[5] O. Cornut, Dear imgui, 2014. https://github.com/ocornut/imgui.

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

[7] B. Guenter, M. Finch, S. Drucker, D. Tan, J. Snyder, Foveated 3d graphics, ACM Transactions on Graphics (TOG) 31 (6) (2012) 1–10.

[8] Intel®, Intel®open image denoise, 2019. https://www.openimagedenoise.org/.

[9] J.T. Kajiya, The rendering equation. Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques, 1986, pp. 143–150.

[10] N.K. Kalantari, S. Pradeep, Removing the noise in monte carlo rendering with general image denoising algorithms. Computer Graphics Forum volume 32, Wiley Online Library, 2013, pp. 93–102.

[11] A.S. Kaplanyan, A. Sochenov, T. Leimkühler, M. Okunev, T. Goodall, G. Rufo, Deepfovea: neural reconstruction for foveated rendering and video compression using learned statistics of natural videos, ACM Transactions on Graphics (TOG) 38 (6) (2019) 1–13.

[12] E. Kilgariff, H. Moreton, N. Stam, B. Bell, Nvidia turing architecture in-depth, 2018. https://developer.nvidia.com/blog/nvidia-turing-architecture-in-depth/.

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

[14] M. Koskela, A. Lotvonen, M. Mäkitalo, P. Kivi, T. Viitanen, P. Jääskeläinen, Foveated real-time path tracing in visual-polar space. Proceedings of 30th Eurographics Symposium on Rendering, The Eurographics Association, 2019.

[15] X. Meng, R. Du, M. Zwicker, A. Varshney, Kernel foveated rendering, Proceedings of the ACM on Computer Graphics and Interactive Techniques 1 (1) (2018) 1–20.

[16] NVIDIA, Nvidia optix-ai-accelerated denoiser, 2017, https://developer.nvidia.com/optix-denoiser.

[17] M. Oren, S.K. Nayar, Generalization of lambert's reflectance model. Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 1994, pp. 239–246.

[18] S. Paris, F. Durand, A fast approximation of the bilateral filter using a signal processing approach. European Conference on Computer Vision, Springer, 2006, pp. 568–580.

[19] B. Qi, T. Wu, H. He, A novel edge-aware à-trous filter for single image dehazing. 2012 IEEE International Conference on Information Science and Technology, IEEE, 2012, pp. 861–865.

[20] M. Reddy, M. Reddy, The Development and Evaluation of a Model of Visual Acuity for Computer-Generated Imagery. Technical Report, 1997.

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

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

[23] B. Walter, S.R. Marschner, H. Li, K.E. Torrance, Microfacet models for refraction through rough surfaces, Rendering techniques 2007 (2007) 18th.

[24] J. Selan, J. Ludwig, A. Leiby, Valvesoftware/openvr: Openvr SDK, 2015, https://github.com/ValveSoftware/openvr.

[25] Z. Wang, A.C. Bovik, H.R. Sheikh, E.P. Simoncelli, Image quality assessment: from error visibility to structural similarity, IEEE Trans. Image Process. 13 (4) (2004) 600–612.

[26] M. Weier, T. Roth, E. Kruijff, A. Hinkenjann, A. Pérard-Gayot, P. Slusallek, Y. Li, Foveated real-time ray tracing for head-mounted displays. Computer Graphics Forum volume 35, Wiley Online Library, 2016, pp. 289–298.

[27] T. Whitted, An improved illumination model for shaded display. Proceedings of the 6th Annual Conference on Computer Graphics and Interactive Techniques, 1979, p. 14.

[28] C. Wyman, S. Hargreaves, P. Shirley, C. Barré-Brisebois, Introduction to directx raytracing. ACM SIGGRAPH 2018 Courses, 2018.

[29] M. Zwicker, W. Jarosz, J. Lehtinen, B. Moon, R. Ramamoorthi, F. Rousselle, S. Pradeep, C. Soler, S.-E. Yoon, Recent advances in adaptive sampling and reconstruction for monte carlo rendering. Computer Graphics Forum volume 34, Wiley Online Library, 2015, pp. 667–681.