

Assignment 3

Deadline	23:59 on February 3, 2021	Students	Carlo Bredius	4132955
Handed in on	February 3, 2021		Brian Janssen	5661154

1 Differences since Assignment 2

General changes:

- Fixed the FPS counter
- Traversal speedup (before and after comparison maybe)
- Added Multi-threading (OpenMP)
- Added occlusion function (instead of only intersect) to the bvh
- Reworked material class to a basic Lambertian BRDF with added color and texture data.
- Renamed `sqrLentgh` to `sqrLength` in `vector.h`¹

Sampling changes:

- Changed uniform hemisphere diffuse bounce into Cosine weighted diffuse bounce
- Changed recursion for sampling into a while loop
- Added Next Even Estimation
- Added Russian Roulette
- Added Multiple Importance sampling

New Filtering additions:

- Split accumulated color into albedo and illumination
- Added Gaussian kernel creation with arbitrary size
- Added `PixelData` class, which stores:
 - First intersection position
 - First intersection normal
 - First intersection material index
 - First intersection albedo
 - Accumulated pixel illumination
 - Additional field for storing data after the first horizontal filtering pass
- Added cross bilateral filter on illumination using:
 - Illumination difference (sigma: 25.0)
 - First intersect location distance (sigma: 2.0)
 - First intersect normal angle (sigma: 0.5)
 - Material index (sigma: 0.5)
- Changed filtering into a kernel separated algorithm, see Section 4.1.
- Added an algorithm for weight re-usage, see Section 4.2.

¹The original `vector.h` file can be found in the template.

2 The code

The code can be found at <https://github.com/tosti007/AdvancedGraphics>.

3 precomp.h defines

Most of the names speak for themselves but we'll explain the new options either way:

- **USESTRATIFICATION**, gives the option to stratify the random numbers, to reduce variance and give a faster pleasant look
- **USERUSSIANROULETTE**, gives the option to use Russian roulette instead of having a set maximum number of bounces, which can be chosen with **MAX_NR_ITERATIONS**
- **USENEE**, Turns on/off Next Event Estimation
- **USEMIS**, Turns on/off Multiple Importance Sampling
- **KERNEL_SIZE**, if given an odd number higher than 0 turns on our bilateral filter. Most of our testing is done using the value 65. If 0 is given, no filtering will be done
- **SIGMA_ILLUMINATION**, Sigma used for the weight of the Gaussian kernel
- **SIGMA_FIREFLY**, The threshold above which a pixel is considered a firefly and will be scaled down with a factor of $1/\text{SIGMA_FIREFLY}$. Removing this also removes firefly suppression.
- **OPENCV2**, Turns on the bilateral filter implemented by OpenCV.

4 Speeding up cross bilateral filtering

Adding (cross-bilateral) filtering to a path-tracer impacts the frames per second (or after filtering rather seconds-per-frame) heavily. Hence we opted to change the naïve filtering algorithm in order to greatly improve performance. In this section we try to give an explanation of what we did and how it impacted performance.

4.1 Kernel separated algorithm

When applying a filtering technique a 2D kernel is applied on each pixel of the image. In cross-bilateral this kernel changes on a per-pixel basis. If ordinary filtering is applied (e.g. with a Gaussian kernel) the filtering process can be made much quicker by applying kernel separation. To obtain a fully equivalent result the rank of the filter kernel should be 1. Since we are doing cross-bilateral filtering the rank of our kernel is actually, in most cases, equal to the size of our matrix, this means our results will not be equivalent to applying it without speedup, but the results are still good looking.

The kernel separation algorithm works, as the name implies, with two passes; one in the horizontal direction and one in the vertical direction. Each pass uses the same 1D kernel, which is equal to the row (or column, as they are equal) from the 2D kernel that passes the center point. After using this 1D kernel for horizontal filtering, this horizontally filtered data is used as input for the second vertical pass. This second pass uses the same kernel as the first pass, but it possibly needs to be transposed, depending on implementation.

By applying the kernel separation the number of fields that needs to be visited per-pixel is reduced from $n * n$ to $n + n$ where n is the size of the kernel. For a kernel with size 65 this means we now only need to do 130 lookups instead of the prior 4225 lookups. For our implemented path tracer this resulted in great speedups. The initial (non-separated) filtering resulted in 0,22 frames per second, which was increased with separated filtering to 4,0 frames per second. This means we obtained a speedup of more than 18 times the initial frames per second, but the filtered results are a bit worse.

4.2 Weight re-usage algorithm

When applying filtering, weights are computed between the center pixel and a neighbouring pixel. The final value for the given pixel is defined as:

$$I^{filtered}(x) = \frac{1}{\sum_{y \in \Omega} W(x, y) D(x, y)} \sum_{y \in \Omega} I(y) W(x, y) D(x, y)$$

Where $I(x)$ is the pixel color at a certain location, $W(x, y)$ is the computed weight for the data differences between pixel x and pixel y , $D(x, y)$ is the weight for the spatial differences between pixels x and y , and Ω are the neighbouring pixels (including x) as defined by the kernel size.

The easiest way to implement this algorithm would be to loop over all pixels and re-compute each W and D value. However we know that:

$$W(x, y) = W(y, x)$$

$$D(x, y) = D(y, x)$$

because both W and D are computed using the Gaussian function. Since we are re-computing all weights for each pixel, we are essentially computing the same value twice. As these weights are computed with the Gaussian function, it is not a cheap operation and we should use the properties from above.

In order to do this we can, rather than doing the sum and normalizing in one pass, apply the formula in two passes. In the first pass we compute $w(x, y)D(x, y)$ for the neighbours of pixel x and store both the sum of just the weights and the weights multiplied by the neighbour pixel value $I(y)$. Then in the second pass we do the normalizing of the summed pixel color with it's total weight. Just splitting the algorithm in two passes does not add any speed, but we can now modify the first pass.

When adding to the total weight and the summed pixel color, we can also do the same for pixel y rather than just x . So in this first pass we compute:

$$I_x^{filteredsum}(y) = I(y)W(x, y)D(x, y)$$

$$I_x^{weightsum}(y) = W(x, y)D(x, y)$$

$$I_y^{filteredsum}(x) = I(x)W(x, y)D(x, y)$$

$$I_y^{weightsum}(x) = W(x, y)D(x, y)$$

and add these to the corresponding accumulators. Given this change, we now know that when filtering pixel y , the weight and color obtained from x has already been added to the corresponding accumulator. Hence, it does not need to be recomputed. In a simple iterative loop from top-left corner to bottem-right corner, this translates to only having to compute these values for all neighbours that are on the lower right quarter in respect to pixel x .

Applying this modified technique effectively reduced the amount of Gaussian function computations to $\frac{1}{2}$ for a 1D kernel and $\frac{1}{4}$ for a 2D kernel. Since we already applied the techniques as described in Section 4.1, only the 1D kernel is used with a size of 65. With the current path tracer setup the total frames per second went from 4.0 (without weight re-usage) to 5.3 (with weight re-usage), which means a speedup of 1.3x compared to Section 4.1 and a speedup of 24x compared to plain filtering.

Do note that we are using CPU only, the proposed algorithm works in an iterative manner and thus will yield different results on a GPU as it needs atomic additions to counter data races.

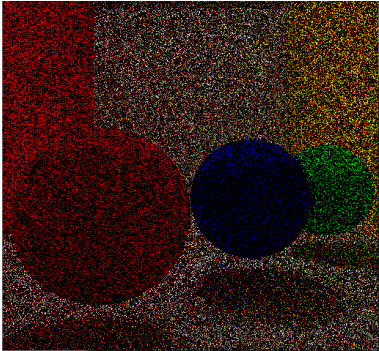
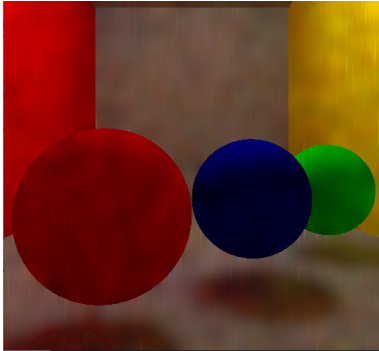
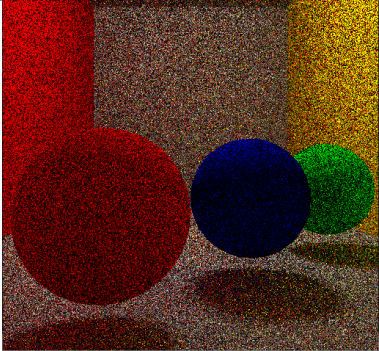
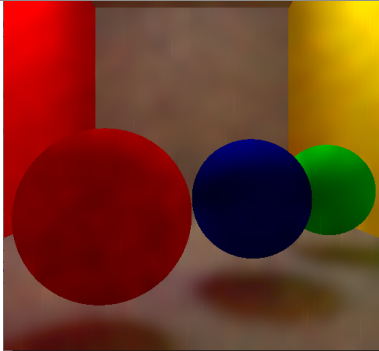
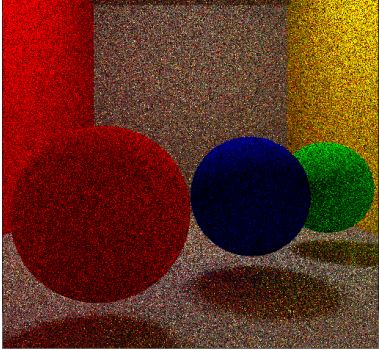
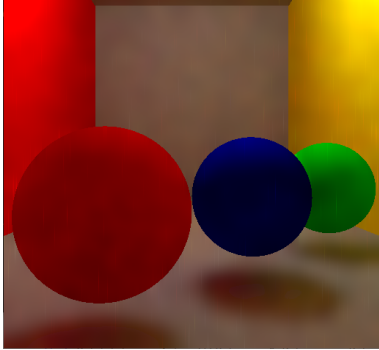
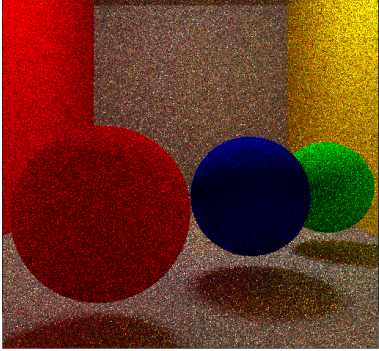
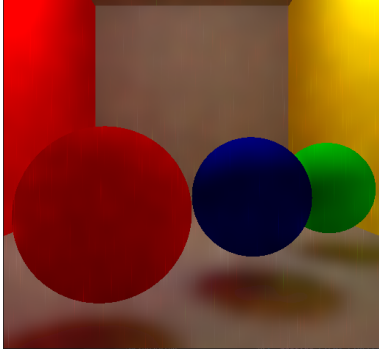
	Raw	Ours
1 spp		
5 spp		
10 spp		
20 spp		

Table 1: Comparison between using our (non-optimized) cross-bilateral filter and the raw image data.

5 Comparison

We analyze the cross-bilateral filter compared to not using it. We will show 1, 5, 10 and 20 samples per pixel (spp) of both to see the speed of convergence. For this we use a kernel size of 65x65. We can use this large kernel size, as the bilateral filter will ensure we will not blur over edges. A comparison is shown in Table 1.

Using the bilateral filter, only having a single spp sees a huge improvement over the raw image in terms of smooth surfaces due the use of a large blurring (Gaussian) kernel. This results in high frequency noise becoming low frequency noise. As spp gets higher this low frequency noise quickly fades away.

Especially in scenes which have a lot of different materials, the separation in material id can easily make a scene prettier in the first few frames (Figure 2). To render Sibenik cathedral, add its `.obj` as an argument (for us `assets/sibenik.obj`) when executing the compiled code.

OpenCV We have also added the option to see what the bilateral filter of OpenCV would do. For this you need to have OpenCV installed (on path C:

opencv). We did not include the library in the submission as it is huge in size. To prevent this trouble Figure 1 shows the result of using this filter. The quality of the image does not increase a lot as the bilateral filter in OpenCV can only use the pixel colors as provided in an image. And because the path tracer has a lot of noise, the function has a hard time figuring out what are real edges, or what is just noise. It sees some noise as actual edges, creating an even sharper cutoff, amplifying the fireflies. For this reason our own implementation, using the low-noise features, can better estimate where real edges are, creating a smooth surface.

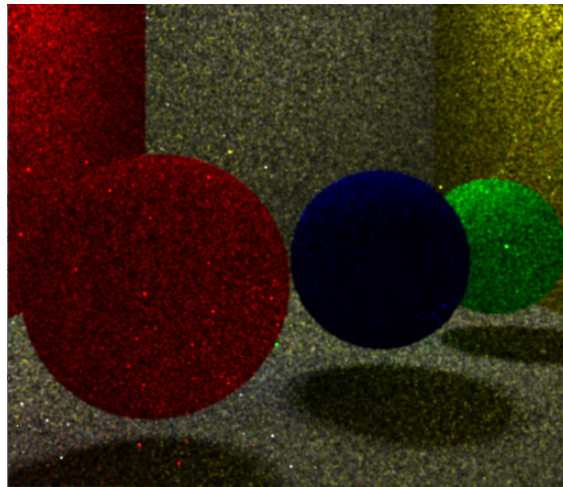


Figure 1: OpenCV's bilateral filter with 10 spp using the parameters $d = 65$, $\sigma_{Color} = 25$ and $\sigma_{Space} = 1$.

6 Additional notes

Filtering fireflies Due to kernel separation, the fireflies get blurred out horizontally before getting blurred vertically. The fireflies that do get through our chosen sigma show up as small vertical stripes.

Sigmas For the filter weight computation several features are needed. Each additional feature introduces a new sigma to be tuned. We tried to give sensible defaults for our sigmas, but they can be heavily scene depended and even vary within one single scene, e.g. whilst looking at a wall versus towards a light. Hence we added the two most sigma as a define.

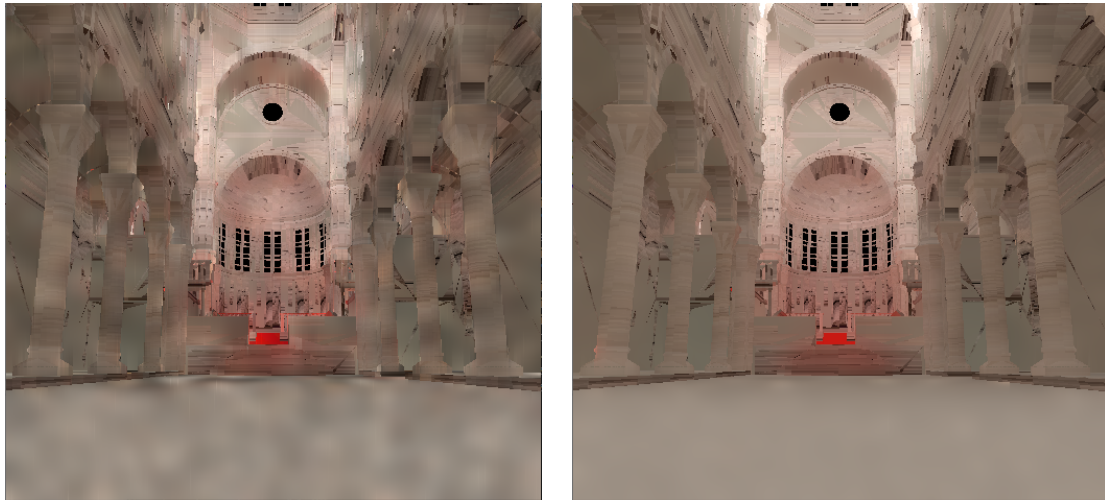


Figure 2: Image of Sibenik cathedral with the bilateral filter, rendered using $\text{SIGMA_ILLUMINATION} = 50.0$, $\text{SIGMA_FIREFLY} = 25.0$ and 1 spp (left) and 100 spp (right).

MIS When turning on Multiple Importance Sampling, quite some fireflies are introduced. For this we added firefly suppression. This works quite well, but there is a trade-off. The trade-off being that the pixels near the light, which have a lot of luminance, are also filtered out.

More SPP fireflies Another trade-off is that firefly suppression creates prettier images in the first few frames, but as the spp increases, the averaged illumination from the accumulator decreases and thus are missed by the firefly suppression. This shows in bright highlights showing up in the later frames.