Flirting with Teethaster

Jad-Nicolas Khoury *
CSS440 - Advanced Computer Graphics

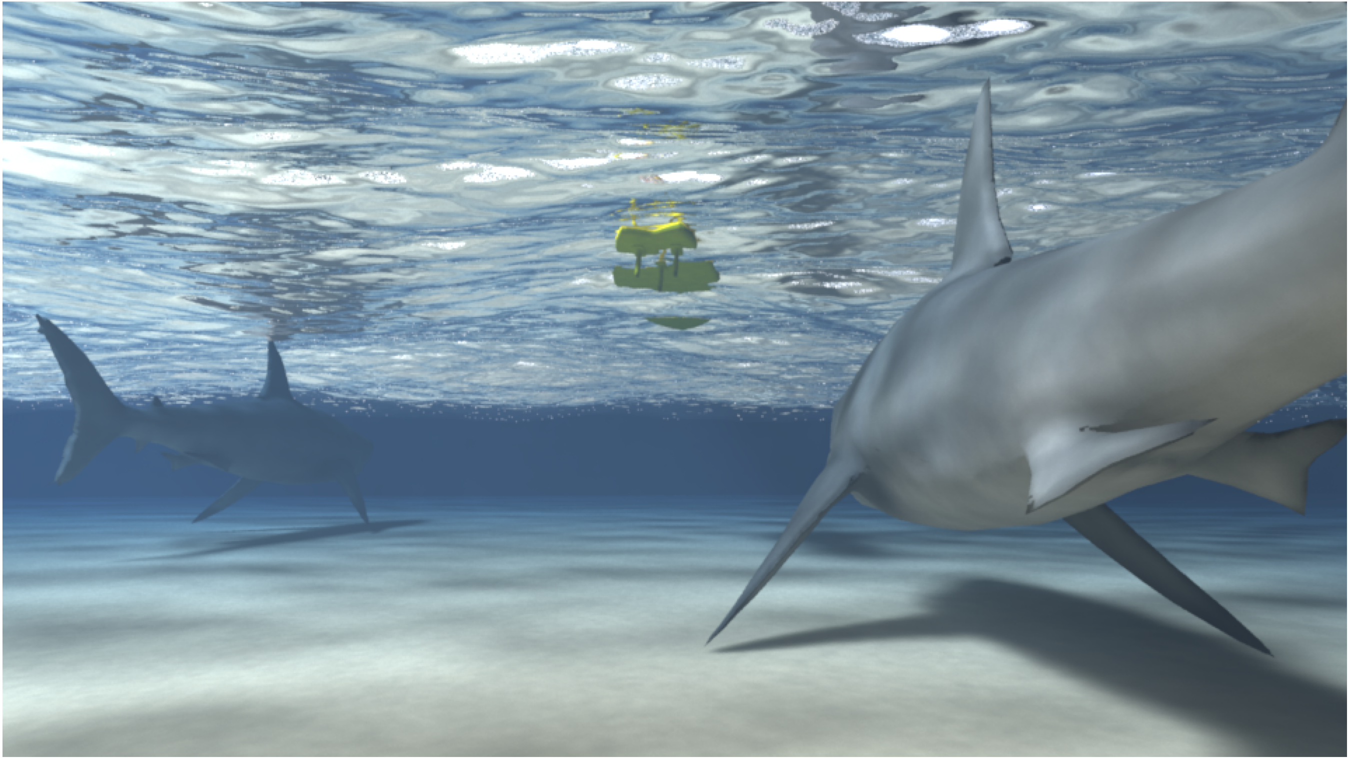

Figure 1: *Final Render, 50M photons per iteration, 12 iterations, g=0.874*

Abstract

This report will describe, explain and prove the implemented algorithms for the final project of the Advanced Computer Graphics course given at EPFL by W. Jakob

Keywords: Photon mapping, global illumination, homogeneous mediums, Beam Radiance Estimate

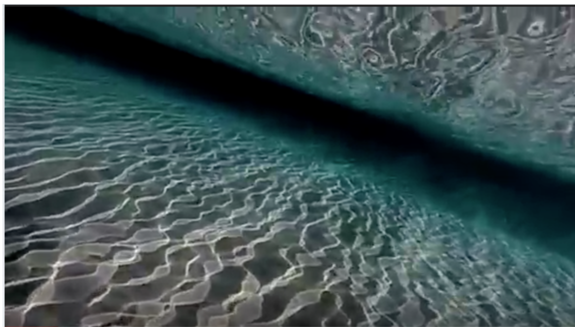

Figure 2: *Inspiration image, taken from an online video*

*e-mail:jad-nicolas.khoury@epfl.ch

1 Objectives

This project aimed at rendering an underwater scene with realistic anisotropic homogeneous medium simulation. The inspirational image shows beautiful caustics projected on the sea-floor as well as sea water as participating medium (cF fig.2). The features implemented for this scene are:

- Probabilistic Progressive Photon Mapping
- Environment Map (a.k.a. Image Based Lightning)
- Texturing
- Normal Mapping
- Mesh Design / Physical Simulation

As Photon Mapping is quite problematic with microfacet surfaces, even more when we have 100% indirect illumination, we will not discuss this model in this report, and therefore consider that any surface is either specular (dielectric or mirror) or diffuse.

2 Progressive Photon Mapping

2.1 Photon Mapping Implementation

As 100% of the illumination in the scene is indirect, most non-metropolized path tracing algorithms fail. As Photon Mapping is

ALGORITHM 1: StoreDiffusePhotons

Input: A reference to the incoming Ray (ray) and its current RGB power ($power$), a 2D sample (s_{BSDF}), a 1D sample ($s_{roulette}$), and an intersection (its)
Output: A boolean ($roulette_shot$) set to true if the Russian Roulette shot, and the updated ray and power
$sampledColor$ = sample BSDF at intersection point;
$p_roulette$ = min(0.99, max($sampledColor$));
if $s_{roulette} > p_roulette$ **then**
  $roulette_shot$ = TRUE;
else
  $roulette_shot$ = FALSE;
  $power* = sampledColor/p_roulette$;
  $photonMap$.push(new Photon($its.position$, $-ray.d$, $power$));
end
ray = sampled direction from BSDF sampling;
return $roulette_shot$

ALGORITHM 2: SpecularReflection

Input: A reference to the incoming Ray (ray) and its current RGB power ($power$), a 2D sample (s_{BSDF}), a 1D sample ($s_{roulette}$), and an intersection (its)
Output: A boolean ($roulette_shot$) set to true if the Russian Roulette shot, and the updated ray and power
$sampledColor$ = sample BSDF at intersection point;
$p_roulette$ = min(0.99, max($sampledColor$));
if $s_{roulette} > p_roulette$ **then**
  $roulette_shot$ = TRUE;
else
  $roulette_shot$ = FALSE;
  $power* = sampledColor/p_roulette$;
end
ray = sampled direction from BSDF sampling;
return $roulette_shot$

the go-to algorithm when aiming at rendering caustics, I first implemented a standard version of this global illumination algorithm, using both nn-search and maximum radii with the help of [Jensen et al. 2001]. I modified a little bit the preprocessing step in charge of filling the photon map in order to isolate some key subroutines and get an almost integrator-looking function. We therefore distinguish two subroutines:

- First, one that serves as photon storing subroutine, described in alg.1
- Second, an almost similar algorithm, depicted in alg.2 that modifies the ray and power when hitting a specular surface. The only difference compared to the previous subroutine is that this one doesn't store any photons.

The preprocess algorithm in charge of preparing the photon map becomes very simple, as shown in algorithm 3.
We compare the results of this algorithm from the result of standard path tracing (without M.I.S.) in fig 4, with parameters tweaked such that the rendering time (preprocessing included) is approximately the same. The first observation is that for some reasons, the image rendered using Photon Mapping is brighter, but also less noisy.

2.2 Probabilistic Progressive Photon Mapping Implementation

Given the size of the rendered scene, and the natural bias of "pure" photon mapping, the implemenatation of a progressive version has been required. I implemented the Probabilistic PPM as proposed in [Knaus and Zwicker 2011]. My implementation did not particularly innovate with regard to the proposed algorithm in the aforementioned paper, appart from the part that allowed direct visualization of the progress in Nori, which will not be detailed here. Fig. 5 compares the result of the Progressive Photon Mapping algorithm with standard path tracing for the same render time. The parameters have been tweaked such that the needed time to render 25 iterations is approximately the same time needed by the Photon Mapping reference.

3 Textures and Normal Mapping

3.1 Textures

Implementing texture is fairly easy, and just needed the implementation of a struct in the BSDF header file. To load external image files, I imported the stb_image library by simply copying the header library in the include folder. Reading the texture and converting from the unsigned char array to color are all standard algorithms detailed with stb_image, and the only addition to these standard function has been the handling of tiled uv coordinates. Indeed, when exporting meshes with uv coordinates, some uv can be outside the $[0, 1]$ range, which is problematic when converting from uv coordinates to pixel coordinates. The solution has been to simply add or substract 1.0 from each coordinates until it's in the right range. Then, when sampling or evaluating a diffuse BSDF, we then simply pass the uv coordinates of the intersection to the sample/eval function, which will then use the texture to read the albedo at the current position, if a texture file has been assigned to the BSDF.

3.2 Normal Mapping

Normal Mapping relies on the same Texture struct. Nevertheless, this texture is not used in the BSDF itself but rather in the accel class, when searching and defining an intersection. We map the uv coordinates of the final intersection point as before, but this time we use the color sampled from the texture to modify the normal of the intersection point. First, as the color has channels between 0 and 1 but the normal has coordinates between -1 and 1 (after normalization), we map the sampled color to $[-1, 1]$ by simply multiplying by 2 and substracting 1 and normalizing the result. We then use the original normal of the intersection to put the new normal in world coordinates, and then use this last result to define the shading frame of the intersection point.

3.3 Results

Figure 6 shows the result of applying the texture, the normal map, and then both to our shark model. These images have been rendered using a simple Path tracer. Unfortunately, given the size of the final scene and the way Photon Mapping works, the texture are not very visible on the final rendering.

4 Volumetric Photon Mapping and Beam Radiance Estimate

4.1 Volumetric Photon Tracing

To complete our preprocess algorithm, we have to create a subroutine to store volumetric photons like we did before for diffuse ones, detailed in algo 4. This algorithm stores volumetric photon while the sample distance allows it, and at the end of its execution,

ALGORITHM 3: Photon Tracing

Input: A pointer to the current scene ($scene$)
Output: A filled photon map
$photonMap$ = new empty photon map;
$roulette_shot$ = FALSE;
ray = empty Ray;
$power$ = empty Color;
its = empty Intersection;
while *photonMap is not full* **do**
  $scene$.emitPhoton(&ray, &$power$);
  $power = power/nb_photon$;
  $roulette_shot$ = FALSE;
  while $roulette_shot$ == *FALSE* **do**
    $scene.rayIntersect(ray, its)$;
    if *no intersection is found* **then**
      break;
    end
    if *BSDF at intersection is diffuse* **then**
      $roulette_shot$ = StoreDiffusePhoton(ray, $power$, new 2D sample, new 1D sample, its)
    else
      $roulette_shot$ = SpecularReflection(ray, $power$, new 2D sample, new 1D sample, its)
    end
  end
end
build $photonMap$'s KDTree

ALGORITHM 4: StoreVolPhotons

Input: A pointer to the current scene ($scene$), to the current sampler ($sampler$), to the current Ray (ray) and power ($power$)
Output: nothing
t, pdf_t = medium.sampleDistance(new 2D sample);
its = new empty Intersection;
w_i, w_o = new 3d vectors;
$hitSurface = scene.rayIntersect(ray, its)$;
max_t = distance from ray origin to intersection;
while $t < max_t$ **do**
  if *$scene$ doesn't contain $ray(t)$* **then**
    break;
  end
  $ray.origin = ray(t)$;
  $power$ *= medium.sigmaS $*$ medium.Transmittance(t) / pdf_t;
  $volPhotonMap$.push(new Photon($ray.origin$, $-ray.d$, $power$));
  $w_i = ray.d$;
  w_o = medium.samplePhaseFunction(w_i, next 2D sample);
  $ray.d = w_o$;
  $hitSurface = scene.rayIntersect(ray, its)$;
  t, pdf_t = medium.sampleDistance(new 2D sample);
  max_t = distance from ray origin to intersection;
end

the ray points toward the next surface interaction (if appropriate). The system is then ready for the StoreDiffusePhoton that can be used right after, as the power has been scaled and the ray has been appropriately scattered. The only modification then needed in the preprocess algorithm is to call this subroutine when the ray is currently in the medium.
Also note that I don't force the size of the volumetric map, i.e. I just record photons as they scatter and intersect surfaces, until I reach the global phonton number.

4.2 The Medium Class

This class implement everything needed for coloured anisotropic homogeneous mediums.
First, the Henyey Greenstein phase function, than can be sampled or evaluated, and which exactness has been proven with a warptest, see fig. 3.
Second, a function that allows to sample a distance in function of the parameters of the medium, and to compute its pdf. As we implement coloured mediums, we have to adapt the standard functions: given two random 1D samples ζ_1 and ζ_2, we compute the sampled distance as:

$$i = floor(3 * \zeta_1) \tag{1}$$

$$t = -\ln(1 - \zeta_2)/\sigma_t[i] \tag{2}$$

and the corresponding pdf as the average of the standard pdf equation across the 3 channels

$$pdf_t = \frac{\sum_{n=0}^{2} \sigma_t[i] * \exp(-t * \sigma_t[i])}{3} \tag{3}$$

4.3 Beam Radiance Estimate

The implementation of the Beam Radiance Estimate has not required any innovation with regard to what is described in its paper [Jarosz et al. 2008]. It has been nevertheless extremely hard to debug because of the number of parts it connects:

- The Volumetric Photon Map
- The BBH Data Structure
- The Phase Function sampling and evaluation
- The distance computation and pdf
- The Transmittance equation

As I don't have time to implement another kind of medium participation just for the sake of comparison, this report will instead show the different results we obtain when we play around with the available parameters.

- fig. 7 shows the effect of scaling σ_s and σ_t, when they are used as float
- fig. 8 shows the effect of the coloured implementation
- fig. 9 shows the effect of changing the phase fonction's g

5 Environment Map

The implementation of Image Based Lightning has been inspired by the implementation proposed by PBRT ([Pharr et al. 2016]). We used importance sampling but unfortunately not mipmapping. As a result, fig. ¡envmap¿ showing the result of my implementation shows extremely pixelated regions in the background.
As the envmap I used in the final rendering was not HDR and that I needed a LOT of incoming radiance, I boosted the high luminance regions with a huge factor both when constructing the DPDF used for importance sampling and when retrieving the colour when emitting photon.

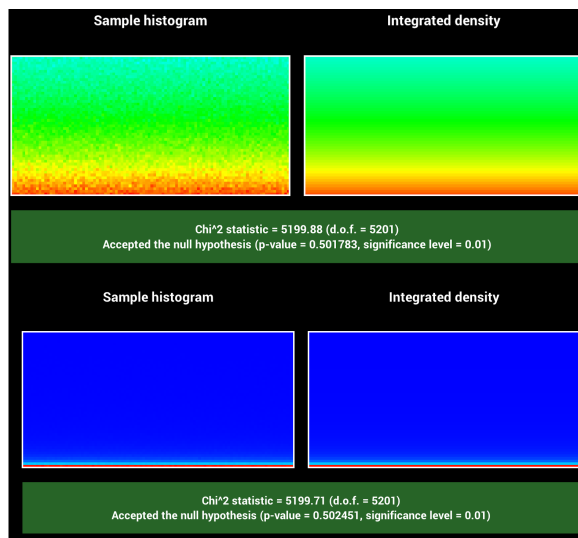

Figure 3: *Warptest of the Henyey-Greenstein phase function. Top:* $g = 0.1$*, Bottom:* $g = 0.9$

6 Mesh simulation, design

The ocean surface has been created from a big triangle mesh on which I applied the Ocean Modifier of Blender, so I don't know if that qualifies as mesh design or simulation.
The shark mesh and its textures are available free on CGTrader: `https://www.cgtrader.com/free-3d-models/animals/fish/great-white-shark-e945a4e090cd71acbf4cfcc6ff54ad9c/`
The Human and its texture has been provided by a friend (Clian Millet) who works in modelisation and who uses it as low poly test human. Since the human sits above the water, I didn't need high details since the few patches of transparence of the water surface only show a highly deformed image of him.
The surf has been designed on Maya by the same friend.

7 Conclusion

After spending an immense amount of time implementing PPM and BRE, everything worked in small text scene (often Cornell Boxes), and I expected the final rendering to naturally show caustics on the floor and "God Rays" under water, but unfortunately it was not the case. I tried overwriting the phase sampling function to always return the incoming ray and therefore not to deviate the light ray at scattering event, but it didn't change anything. I conclude from this experiment that the absence of sharp caustics comes from the ocean mesh rather than my algorithms. I would have loved to spend more time tweaking and testing different scene setup, but since my whole project would be nothing without the B.R.E., making it work was my only objective during two long weeks, and by the time I debugged it, I barerly had the time to put my scene together on blender and launch the render.

References

JAROSZ, W., ZWICKER, M., AND JENSEN, H. W. 2008. The beam radiance estimate for volumetric photon mapping. In *ACM SIGGRAPH 2008 classes*, ACM, 3.

JENSEN, H. W., CHRISTENSEN, P. H., AND SUYKENS, F. 2001. A practical guide to global illumination using photon mapping. *ACM SIGGRAPH 2001 Course Notes CD-ROM*.

KNAUS, C., AND ZWICKER, M. 2011. Progressive photon mapping: A probabilistic approach. *ACM Transactions on Graphics (TOG) 30*, 3, 25.

PHARR, M., JAKOB, W., AND HUMPHREYS, G. 2016. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

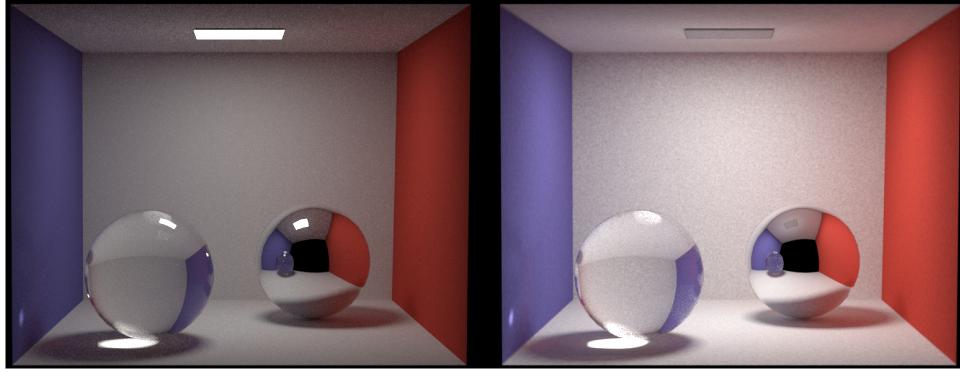

Figure 4: *Left: path tracing, right: PM with 50M photons, k = 300, both roughly 5min. Please ignore the dimmed emmiter mesh on the right.*

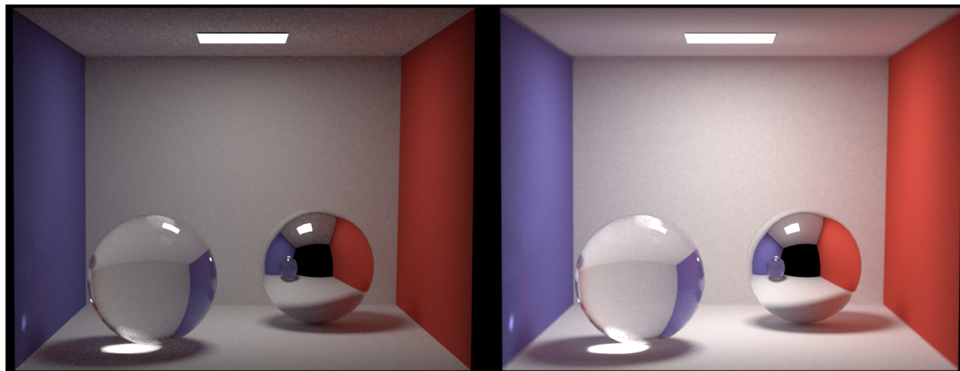

Figure 5: *Left: path tracing, right: PPM with 25 iterations, 5M photons / iteration, k = 100, both roughly 5min.*

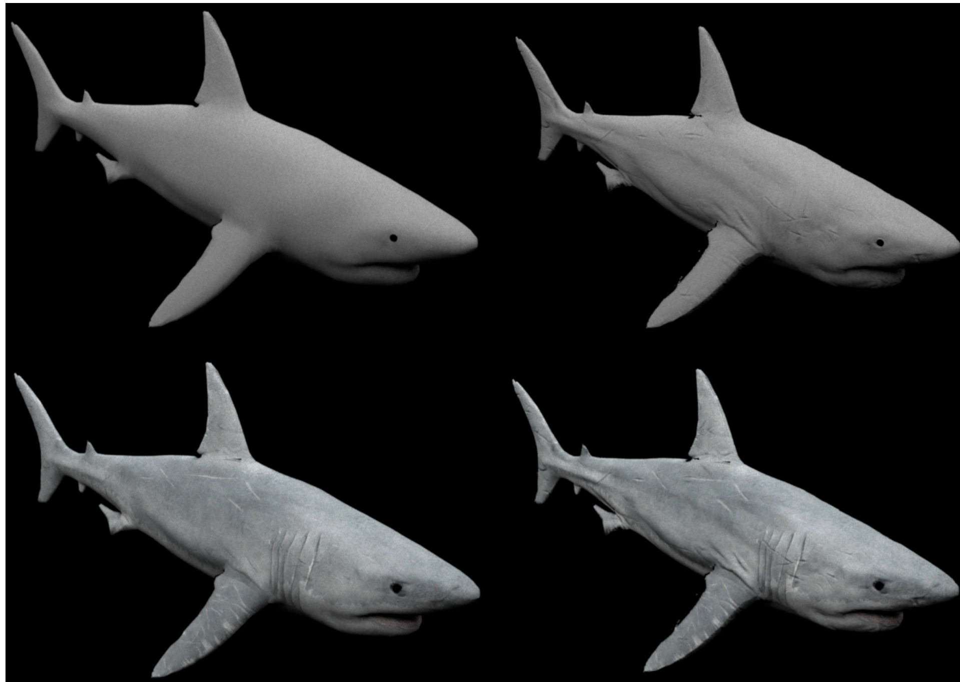

Figure 6: *From top-left to bottom-right: no texture, normal map, texture, both*

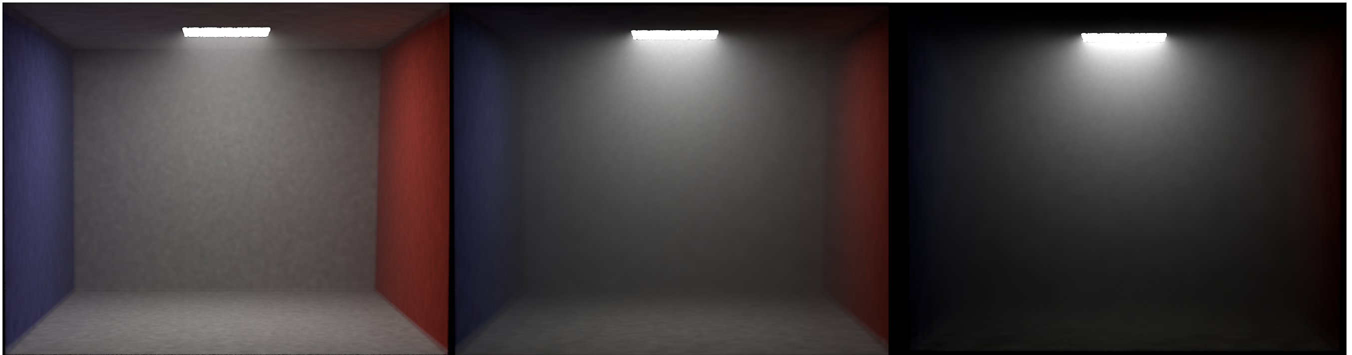

Figure 7: *B.R.E. - Comparison of the result with* $\sigma_s = \sigma_t = 0.5, 1.0, 2.0$ *(from left to right), 10M photons, k = 300, g=0*

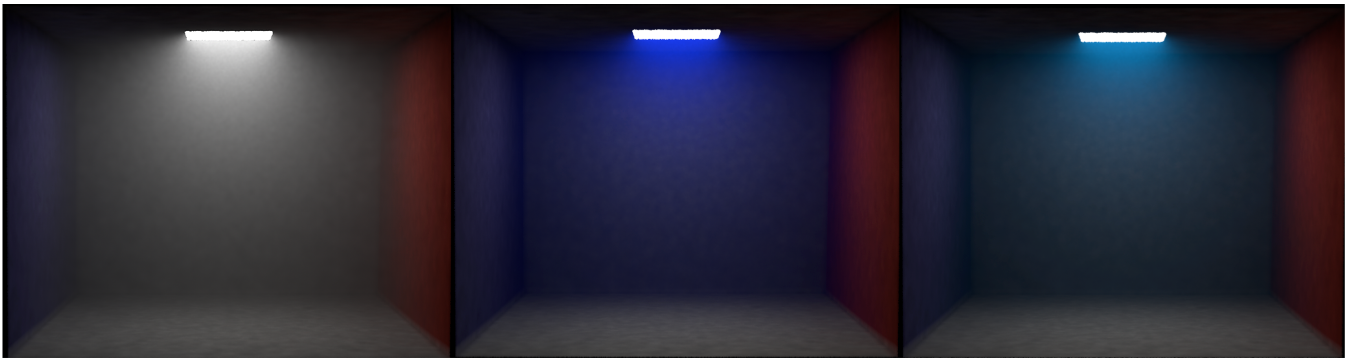

Figure 8: *B.R.E. - Comparison of the result with* $\sigma_s = (1,1,1), (0,0,1), (0,0.5,0.8)$ *(from left to right).*
10M photons, k = 300, g=0, $\sigma_t = (1,1,1)$

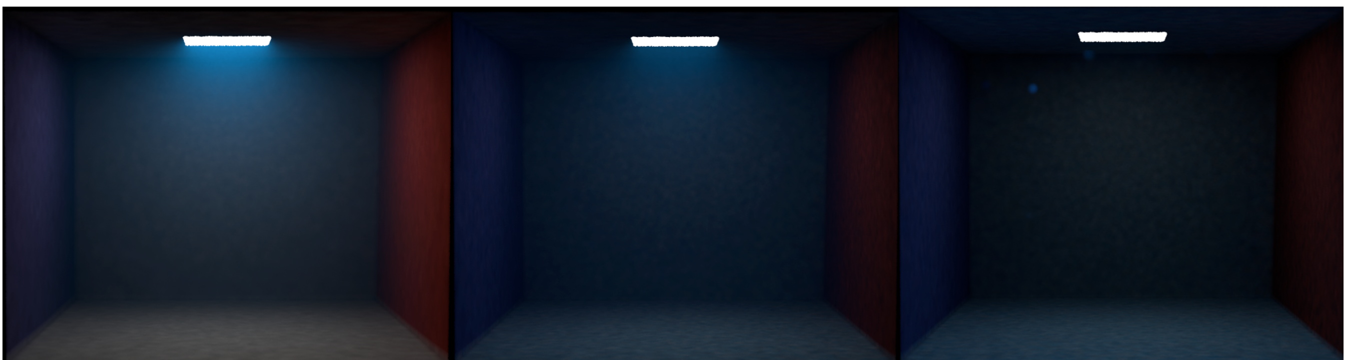

Figure 9: *B.R.E. - Comparison of the result with* $g = 0.0, 0.5, 0.99$ *(from left to right).*
10M photons, k = 300, $\sigma_t = (1,1,1)$, $\sigma_s = (0,0.5,0.8)$