DH2323 Project Report : Implementing a basic pathtracer

Jonathan Guichard (jgui@kth.se - 941028-3577)
Project blog : https://dh2323jgui.wordpress.com/

Specification

Pathtracing is a method of rendering images of three-dimensional scenes such that the illumination is faithful to the reality. It simulates many effects "out-of-the-box", such as shoft shadows, ambient occlusion and indirect lighting, that otherwise have to be coded into other rendering methods, for example such as raytracing. The fundamental principle consists of casting light rays starting from the camera, then recursively casting new random rays in order to determine the color of the hit area. This method makes use of Monte Carlo integration and the rendering equation.

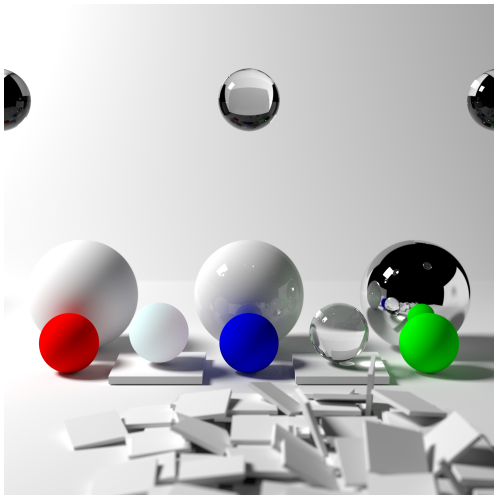

Figure 1: Example of an image rendered using a pathtracer, showcasing the capabilities of the method.

The first goal for this project is to implement a very simple pathtracer, capable of rendering a scene of the same complexity as the one used in labs 2 and 3 of the rendering track.
In other words, the pathtracer we implement must be capable of rendering a simple Cornell Box with a fixed surface light source on the roof, of variable size and emitted color. The camera has a fixed position and can not rotate, and we consider all the objects present in the scene to be made of the same material and to be ideally diffuse. We do not render

any complex materials and phenomenons that would require some specific handling, such as water, glass, specular surfaces or caustics, to only name a few.

The second objective is to compare pathtracing to the two other different rendering methods implemented during the labs of the rendering track.

Related work

The original pathtracing algorithm was first proposed by Kajiya in *The Rendering Equation* [3], in 1986. This is the algorithm we will base our implementation on for this project.

An improved version of this pathtracing algorithm was introduced by Lafortune and Willems in 1993 [2]. This algorithm is based on [3], and it significantly improves the performances for indoor scenes where indirect lighting is important, and it is also able to better deal with complex phenomenons such as light caustics. The basic principle consists of tracing rays originating from both the camera and the light sources, and to then "connect" both rays in order to form a full path. In practice this strategy converges much faster than regular pathtracing, despite the extra calculations needed.

The two papers cited above constitute the basis for many other academic works aiming to further improve the pathtracing algorithm. Pathtracers are however of course not only a theoretical topic of interest, and are nowadays widely used in the animation movie industry. The algorithms used to produce those movies are of course greatly improved compared to the brute-force version we are implementing in this project, especially when it comes to speed and denoising, as detailed in [1]. The very heart of the technique remains however the same as the one we will implement.

Theoretical aspects

Pathtracing is quite similar to raytracing, as for each pixel, rays are traced from the camera to determine a collision point with the scene we are rendering. However, once the collision point has been determined, we do not use mathematical models to determine the illumination of this point, but rather try to simulate at a lower level what whould physically happen in reality.

The rendering equation [3] is the cornerstone of pathtracing, and tells us how we should compute outgoing light from an intersection point x. It is as follows (we consider that all the wave lengths have the same behavior) :

$$L_o(w_o, x) = L_e(w_o, x) + \int_\Omega f(w_i, w_o, x) L_i(w_i, x)(w_i \cdot n) dw_i \tag{1}$$

Where w_o is the direction of the outgoing light, w_i the direction of the incoming light, f is the bidirectional reflectance distribution function, n the normal of the surface hit by the ray, and L the radiance. In other words, the light we see when looking at an object is made up of the light emitted by this object and by the reflected light (in our viewing direction) that is hitting the point x on the object, and this from all possible directions.

However, in order to determine the radiance of the outgoing light, we need to compute an integral over an hemisphere centered around the normal of the surface as equation (1) shows us. In order to compute that integral, we make use of what is called Monte Carlo Integration. The idea is to randomly choose some elements in the integration domain Ω, and use those elements only in order to approximate the true value of the integral. In mathematical terms, we would approximate an integral I with $\widetilde{I}$ as follows [6] :

$$\widetilde{I} = \frac{1}{N} \sum_{i=0}^{N-1} \frac{f(X_i)}{pdf(X_i)} \tag{2}$$

Where N is the number of samples, X_i a random sample belonging to the integration domain, and *pdf* the probability distribution function for selecting a random sample. When performing a uniform sampling, which is our case, this simply becomes an average of uniformly randomly selected rays, as the probability density function is the same for every sample. The bigger the number of samples, the closer we should be to the real value.

Putting those two theoretical elements together, the following pseudocode would be the "heart" of our pathtracer. It computes the outgoing light from a point x situated on an object surface :

$reflected\_light = 0$;
repeat
  Pick a random direction $\vec{d}$ at uniform probability such that $\vec{d} \cdot \vec{n} \geq 0$, with $\vec{n}$ the normal of the surface hit;
  $cos \leftarrow \vec{d} \cdot \vec{n}$;
  $BRDF \leftarrow 2 * reflectance\_surface * cos$;
  $reflected\_light \leftarrow reflected\_light + incoming\_light\_from(x, \vec{d}) * BRDF$;
until $number\_of\_samples = N$;
$outgoing\_light = emitted\_light\_by\_surface + reflected\_light/N$;

NB We only handle ideally diffuse surfaces, which have the same reflectance regardless of the viewing angle, therefore we do not take this viewing angle into account in the above pseudocode.

The only remaining element we need to figure out is how to know the value of the incoming light to a point and from a given direction, since we need that information when computing the reflected light.
The way a pathtracer handles this is through recursion : in order to know the incoming light, find the originating surface point of that light, and apply the pseudo-code presented above. We then only need to set a maximum depth limit to this recursion, in order for the program to actually be able to finish its execution. When we reach that maximum depth, we simply consider that there is no incoming light (we can see this as the light "being" perfectly black).

The following figure gives a good visualization of how a pathtracer is working at its core :

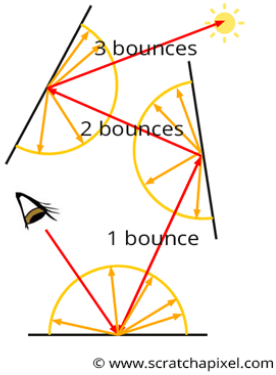

Figure 2: The base principle of pathtracing. For clarity, not all rays have been drawn. Image courtesy of www.scratchapixel.com [6].

Implementation Remarks

Monte Carlo Integration In order to select a random direction within a hemisphere with a uniform probability, we chose to implement a method based on random gaussian variables, as detailed in the very last section of [7] (equation 16). This solution is originally for picking a random point with uniform probability on a sphere, but can easily be reduced to a hemisphere. If we end up with a direction belonging to the "wrong" hemisphere, we simply invert the randomly picked vector (multiply all its elements with -1).

Möller-Trombore intersection algorithm In order to speed-up the intersection computations, we decided to replace the code from Labs 2 and 3 with the Möller-Trombore intersection algorithm, which yields significantly better performances, as this method does not require matrix inversions or any other costly operations. For more details, please refer to [5].
The thought process to solve performance issues is further detailed in the project blog.

Multithreading Finally, in order to further speed up our pathtracer and take advantage of modern computing power, we decided to implement some basic multithreading. In our implementation, each pixel color is being computed in parallel, as those computations are independent and have no side effects.
The thought process to solve performance issues is further detailed in the project blog.

Gamma correction In order to get a more faithfull rendering, we decided to implement gamma correction with $\gamma = 2.2$.

Results, Experiments and Discussions

NB The following images where generated with a depth of 3 and gamma correction of $\gamma = 2.2$, unless otherwise specified. The colors of the scene were also slightly changed on some renders to get more "popping" colors when generating them. Those color differences are therefore not caused by the rendering technique in itself, simply by a different scene definition.

When running the pathtracer detailed previously, the first observation we make is that the algorithm converges quite slowly towards a noiseless image. The observed noise is directly impacted by the number of samples we choose to use. Indeed, with a low number of samples, the accuracy of the Monte Carlo estimate is lower, leading to a higher variation of the color value for neighboring pixels, which is visually expressed as noise, as shown by figure 3.

Based on this observation, we decided to slightly alter the pathtracing algorithm presented page 4. Instead of performing a Monte Carlo integration at *every* recursion step,

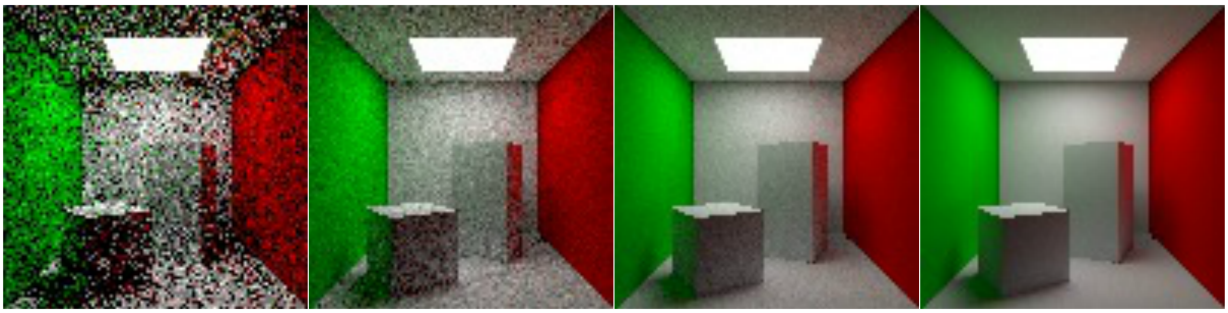

Figure 3: Effect of the number of samples on noise in the final image. The number of samples have been multiplied by 10 for each image, going from left to right.

we decide to only perform this integration at the "pixel level". We are now casting M rays originating from the camera instead of just one, rays which we then average. When one of those rays hits a surface, we cast only one child ray, at uniform probability within the hemisphere centered around the normal to the surface that was hit, and so on and so forth until we reach the maximum recursion level. We of course make use of the rendering equation each time a ray hits a surface, the only difference being that we use only one sample to "perform" the Monte Carlo integration.
The justification behind this modification is that we want to spend more time where it really matters, that is at the pixel level. Indeed, computing a very precise light contribution at deep recursion levels will only very slightly contribute to a more accurate pixel color, if at all. We also expect any precision problems for the Monte Carlo integrations to be corrected when averaging at the pixel level. The expected result is thus less noise for a similar running time, while the image retains its original characteristics. Figure 4 showcases a comparison of both methods :

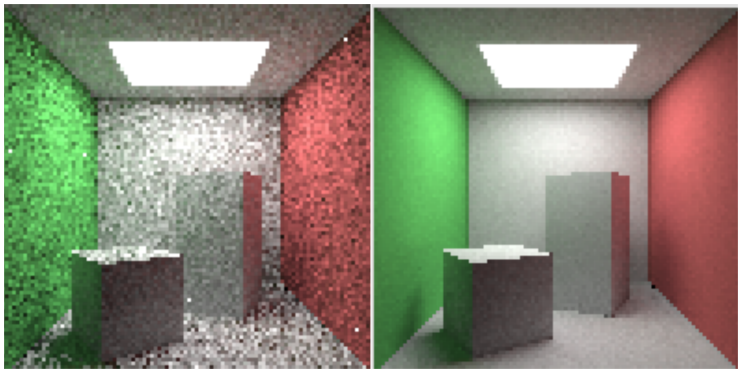

Figure 4: Comparison of the original pathtracing method (left) with the faster version (right). Those two images where generated for a similar running time.

As we can see, there seems to be no detail loss with this new method, as we still have smooth shadows, indirect lighting, and ambient occlusion. We have therefore made the

choice to keep this new method in our implementation.

Figure 5 show us the effect of depth on the final rendered result.
When the depth is just one, we only "see" the light source and nothing else, as expected, since we are not able to compute any light component other than the emitted light.
For a depth of two, we are now able to see something else than just black and white, but the image is still not very eye-pleasing, as many areas of the picture are pitch black. Again, this can be explained by the fact that the rays don't have enough "bounces" to reach the light source, leading to some very black spots (especially for the roof, it's indeed impossible to hit the light source with just one ray starting from any point on the roof). We are still able to observe interesting features tough, such as smooth shadows and some sort of ambient occlusion.
Finally, the rendered results start to be much nicer looking for a depth of 3 or more, since we are now actually having enough bounces to eliminate the pitch black spots, and this also allows us to see some indirect lighting (we indeed need a least 3 bounces to see those features appear). Interestingly though, there is little to no visible difference when using a depth of 4 compared to a depth of 3. The contribution to the final result seems to be very minor, as one could have expected, for deeper recursion levels. A maximum recursion depth of 3 therefore seems to be the best time/details compromise.

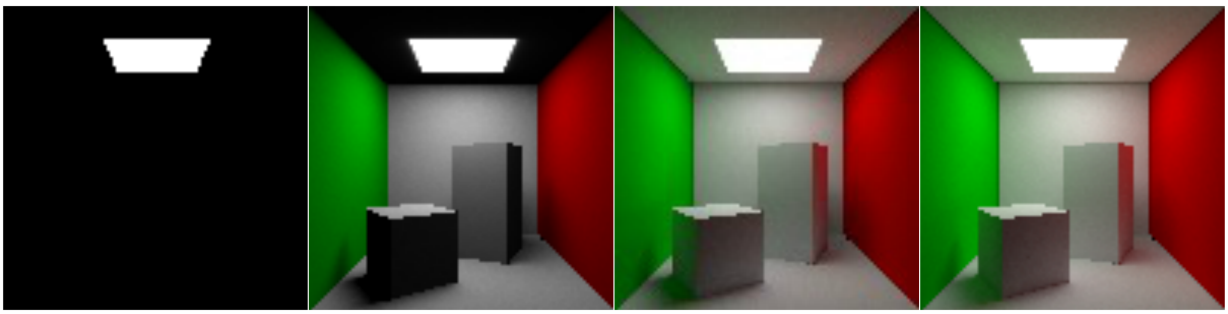

Figure 5: Effect of the maximum recursion depth on the final result, varying from 1 to 4 (left to right). These images were generated with 50000 samples.

The final result obtained for this basic pathtracer is shown in figure 6. The rendering time was approximately of 20 hours for 512x512 pixels on a 4-core 2.2Ghz CPU. The same scene was generated with the Raytracer from Lab 2 and the Rasterizer from Lab 3, shown in figure 7, for comparison.

The main conclusion we can draw from this project is that we are able to achieve very interesting and eye-pleasing results in just a few lines of code with a very general and elegant method, which can be a surprising and counter-intuitive statement at first. However, some very nice features, such as indirect lighting, smooth shadows, and ambient occlusion work "out-of-the-box" for this method, while it would require some specific non-trivial handling

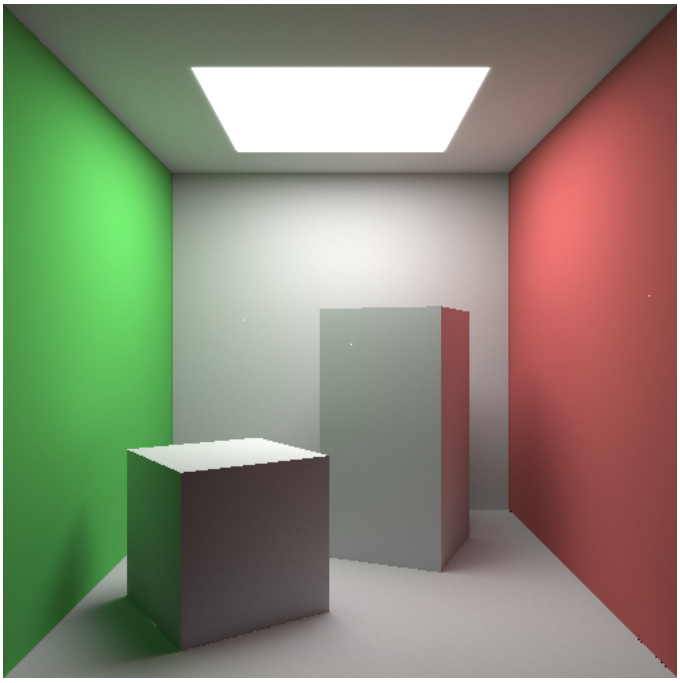

Figure 6: Our pathtracer rendering a Cornell box using the faster method (100000 samples per pixel).

in methods such as Raytracing or Rasterization.
As a general conclusion, we could arguee that the closer the method is to the physical reality, the less specific handling it requires and the better looking the results are, but this comes of course at the cost of performance.

Perceptual user studies

Perceptual user studies can give us some very interesting and valuable insight on how to improve rendering methods. We can for example perform those studies in order to improve and judge the quality of a rendering method, but we could also make use of those studies in order to speed up the algorithm.

We can for example cite [4], which aims at improving performances by simplifying the mesh to render based on the viewing distance and angle. Details that are too far to be observed, or that can't be observed, do not need to be precisely computed (or even computed at all), and can be safely approximated without worsening the perceived quality of the image. The findings of this study could of course be applied to our implementation and greatly improve the handling of complex scenes.

We could also try to improve the speed of the pathtracer by introducing some sort of convergence stopping criteria for the Monte Carlo integration. As we can see on figure 3,

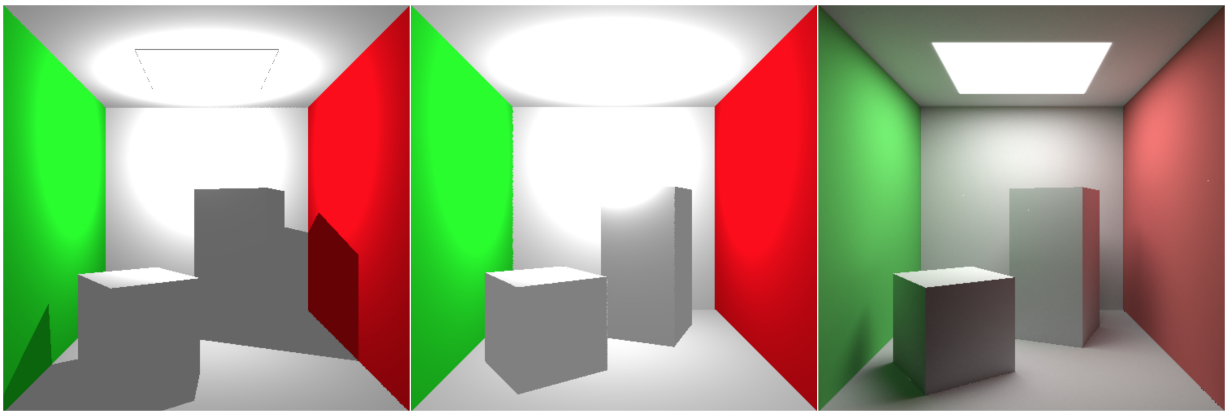

Figure 7: A similar scene being rendered by our raytracer (left, $\approx 10s$), rasterizer (center, $\approx 1s$), and pathtracer (right, $\approx 24h$). The raytracer and pathtracer required roughly the same amount of lines of code, while the rasterizer required approximately the double.

some pixels - usually the well lit ones - tend to converge much faster. Therefore having a hard coded number of samples for every pixel might not be the best approach : we could end up spending a lot of time on pixels for which we already have a good estimate, while spending not nearly enough time on more complicated ones. It can therefore be interesting to come up with a better stopping criteria.
We could then perform a perceptual user study in order to assess the quality of the stopping criterias we come up with. Test subjects would view a succession of images, generated with different stopping criterias, and ask them to rate those images according the perceived faithfulness to reality. This should ideally be done for images representing different scenes, in order to try to provide the best overview and assessment possible for the different stopping criterias. The presented images should be shuffled in random order to try to reduce any sort of bias the subject could have, with some images being repeated in the hope of reducing the "noise" we could get from the study. This is loosely inspired by the perceptual user study conducted by J. Shabo for his master's thesis work involving virtual crowds.

Bibliography

[1] P. Christensen and W. Jarosz. The path to path-traced movies. In *Foundations and Trends in Computer Graphics and Vision*, volume 10, pages 103–175, 2014.

[2] Yves D. Willems Eric P. Lafortune. Bi-directional path tracing. `http://graphics.cs.kuleuven.be/publications/BDPT/BDPT_paper.pdf`, 1993. Accessed: 2017-05-04.

[3] James T. Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150, 1986.

[4] D. Luebke and B. Hallen. Perceptually driven interactive rendering. `https://www.cs.virginia.edu/~luebke/publications/pdf/perceptual.ir.pdf`, 2001. Accessed: 2017-05-04.

[5] T. Möller and B. Trumbore. Fast, minimum storage ray/triangle intersection. In *Journal of Graphics Tools*, 1997.

[6] Scratchapixel. Global illumination and path tracing. `https://www.scratchapixel.com/lessons/3d-basic-rendering/global-illumination-path-tracing`. Accessed: 2017-05-12.

[7] Eric W. Weisstein. Sphere point picking. From WolframMathWorld - `http://mathworld.wolfram.com/SpherePointPicking.html`. Accessed: 2017-05-10.