DH2323 Computer Graphics and Interaction
Parallel Path Tracing in CUDA
Project Report

Henrik Dahlberg*

KTH Royal Institute of Technology

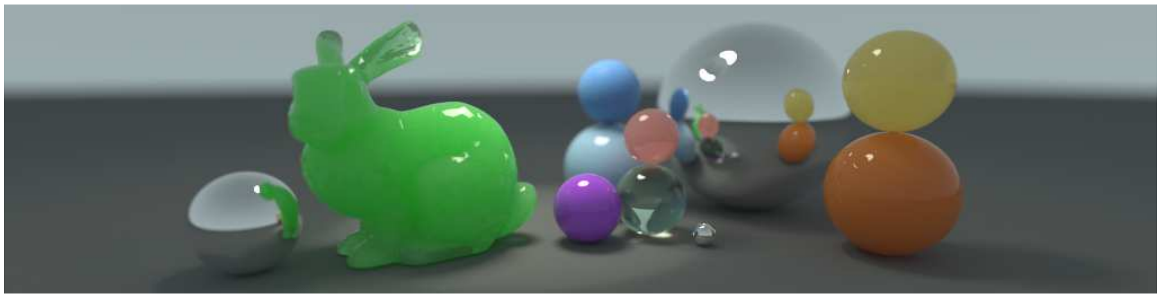


Figure 1: *Rendering showing the final state of the path tracer. It features triangle meshes, spheres, finite aperture sampling depth of field, anti-aliasing, Fresnel reflection and refraction and isotropic Monte Carlo subsurface scattering. The image was rendered with 42,000 samples per pixel with a maximum of 40 bounces per ray.*

Abstract

We present the implementation of a physically-based parallel path tracer in CUDA/C++. We utilize the parallel computational power of modern GPUs to render photo-realistic images of simple, well-lit scenes orders of magnitude faster than a serial implementation. The system features a moveable camera allowing interactive navigation through the scene and is capable of rendering triangles meshes and spheres, glossy, diffuse, specular and transmissive materials as can model isotropic participating media and subsurface scattering.

1 Introduction

In the field of computer graphics, global illumination (GI) techniques are a collection of algorithms with the aim of adding realistic lighting to a scene. The algorithms are tasked with computing measurements of the incident radiance from the light sources as well as from surfaces and participating media from which light has been scattered. The methods are used widely to solve light transport problems in film making and architectural visualization and has begun seeing use in computer games and other real-time applications.

The rendering equation (1) is an integral equation that describes the exitant radiance L_o from a point $\mathbf{x}$ on a scene object as the emitted radiance L_e plus the scattered incident radiance L_i at that point from the rest of the scene.

$$L_o(\omega_o, \mathbf{x}) = L_e(\omega_o, \mathbf{x}) + \int_{\mathcal{M}} L_i(\omega_i, \mathbf{x}) d\mu(\mathcal{M}) \tag{1}$$

The equation was introduced by James Kajiya [1986] who simultaneously introduced the path tracing algorithm.

Path tracing is aimed at solving the rendering equation by sampling light paths of arbitrary length by casting rays from the camera and tracing them through the scene, successively computing their scattering by sampling the appropriate scattering distribution. The light transport problem is transformed to an integration problem by formulating individual measurements as a path integral over the set of paths of all lengths Ω, where a path $\bar{\mathbf{x}} \in \Omega$ [Veach 1997].

$$I_j = \int_{\Omega} f_j(\bar{\mathbf{x}}) d\mu(\bar{\mathbf{x}}) \tag{2}$$

The integrand f_j is the *measurement contribution function* and is sampled by generating a path $\bar{\mathbf{x}} \in \Omega$ from some probability distribution $p_j(\bar{\mathbf{x}})$ and the samples are then combined to generate an unbiased estimate of the measured radiance I_j incident to the camera using the standard Monte Carlo estimator. The estimate $\hat{I}_j$ for N path samples can be written as the standard Monte Carlo integral estimate.

$$\hat{I}_j = \frac{1}{N} \sum_{i=1}^{N} \frac{f_j(\bar{\mathbf{x}}_i)}{p_j(\bar{\mathbf{x}}_i)} \tag{3}$$

CUDA is a parallel computing framework created by Nvidia, allowing the use of CUDA-compatible graphics processing units for general purpose computing. GPUs are highly parallel multi-core systems with the capability of launching thousands of threads tasked with performing operations in parallel. In the path tracing framework, the individually sampled paths $\bar{\mathbf{x}}$ are independent and the process of sampling paths to generate an image representation of a 3D scene - an array of pixel measurements I_j - is therefore inherently parallel in that we can assign one thread per path.

2 Related Work

In this section we present a selection of a few sources that have been used directly as sources of inspiration for this project from the vast amount of references available in this field.

Matt Pharr and Greg Humphrey's famous book on physically-based rendering [2004] is concerned with presenting the theory of rendering photo-realistic scenes while simultaneously implementing the

*e-mail:hdahlb@kth.se

theory in a software rendering system. While their presentation does not include a GPU implementation, it is pedagogical and has served as a reference for the system design aspects of this project.

The Brigade renderer is a commercial rendering software by OTOY [Bikker and van Schijndel 2013]. It uses GPU cloud computing to render path traced images at interactive speeds and their video demonstrations are an inspiration for where projects like the one described in this report may end up in the future.

Peter Kutz and Karl Li, current employees of Walt Disney Animation Studios did a similar project to this [Kutz and Li 2012]. Their work has served as a guide and an instructive source of inspiration for this project.

Samuel Lapere runs a blog on rendering that hosts a series of tutorials for path tracing on the GPU with CUDA [Lapere 2016]. While the examples shown on the blog are not used directly in this work, they have showcased features and technicalities in the implementations that are helpful.

3 Implementation and Results

We present the steps taken in the implementation of the GPU path tracer. An account of the development progress made throughout the project is readily available at the project blog [Dahlberg 2016].

3.1 Random Number Generation

In order to perform Monte Carlo simulations we need to sample distributions such as the bidirectional scattering distribution functions (BSDFs) present in the scene. In our setting it is therefore essential that we can generate uniformly distributed random numbers on the GPU. We also want the samples taken to be uncorrelated between GPU threads and over simulation iterations. If this is not the case there will be bias and artefacts introduced in the generated image.

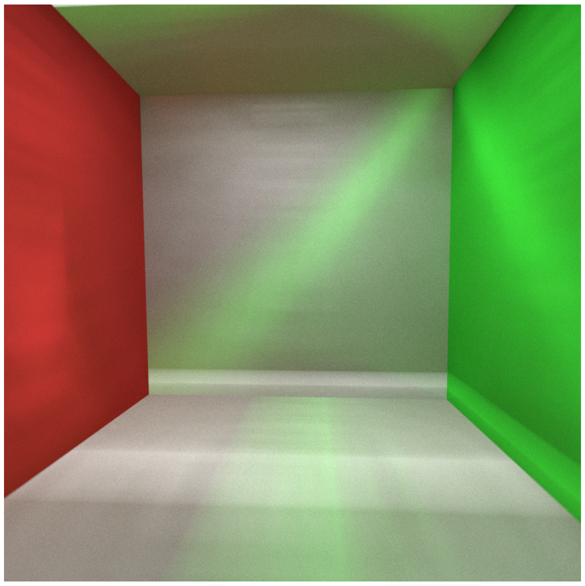


Figure 2: *Bias and rendering artefacts introduced from passing correlated seeds to initialize the random number generator.*

We use the Thrust library to generate uniformly distributed random variables [Bell and Hoberock 2011]. When a new direction is sampled for a ray during a scattering event, the ray pixel index, iteration number, time and ray bounce depth are hashed and combined to generate a seed for the initialization of a thread-specific random number generator.

3.2 Image Accumulation

In each iteration $t = \{1, 2, \ldots, N\}$, where N is the final number of samples per pixel, a path $\bar{\mathbf{x}}$ is sampled for each pixel j and its contribution to the Monte Carlo estimate of the pixel measurement I_j is updated sequentially.

$$\hat{I}_{j,t} = \frac{1}{t}\left[(t-1)\hat{I}_{j,t-1} + \frac{f_j(\bar{\mathbf{x}})}{p_j(\bar{\mathbf{x}})}\right] \tag{4}$$

Expanding this recursion for $t = N$ gives expression (3). This allows us to observe the convergence of the simulation while we are collecting and incorporating more samples into the image estimate. A CUDA array is mapped to an OpenGL buffer that is displayed to the image, and the output of the computations performed on the GPU is stored in the array.

3.3 Generating Path Samples

When computing the Monte Carlo estimate $\hat{I}_j$ we need the path samples $\bar{\mathbf{x}}$. It is the generation of these samples that are at the core of the path tracer and that which is responsible for the majority of the computational intensity. We initialize an array of rays and an array of pixel indices that each ray is assigned to. We then compute the ray origins by sampling the aperture and sample a ray direction through a point on the image plane in the ray initialization step step. The aperture sampling produces the depth of field effect visible in Fig. 3 and Fig. 4. The rays are then traced in parallel through the scene in an iterative fashion, the nearest intersection with the scene geometry is computed. The material properties are taken into account after which each ray is scattered from the intersected geometry point by sampling a new direction. In this work we have implemented perfectly diffuse samples as well as reflection and refraction using the Fresnel equations. The procedure is summarized in pseudocode in algorithm (1).

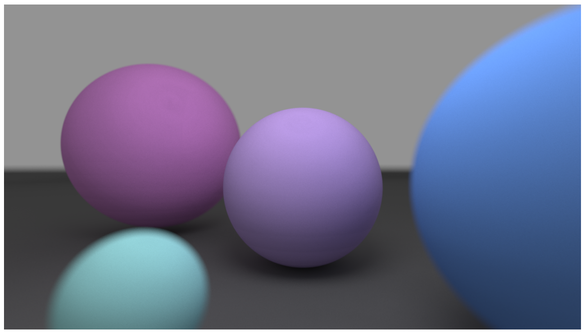


Figure 3: *Rendering generated in an early state of the path tracer. Shows the depth of field effects generated by sampling the ray origins in the camera ray casting from an aperture of finite non-zero radius.*

3.3.1 Ray Parallelization

The path tracer utilizes ray parallelization as opposed to pixel parallelization. We assign a CUDA thread i to each ray and keep track of which pixel measurement I_j the ray belongs to through an array of ray pixel indices $\bar{\mathbf{p}} = \{\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_M\}$, where M is the number of pixels in the image being computed. The ray pixel indices $\bar{\mathbf{p}}$ is a

Algorithm 1 Path tracing

```
Input: Scene $S$ and camera specification $C$
Output: Path traced image $I$
 1: for each iteration do
 2:     InitData(Color $\bar{\mathbf{c}}$, RayPixelIncides $\bar{\mathbf{p}}$) in parallel
 3:     InitRaysFromCamera($\bar{\mathbf{r}}$, $S$, $C$) in parallel
 4:     while rays $\bar{\mathbf{r}}$ not all terminated do
 5:         RayTrace($\bar{\mathbf{r}}$, $S$) in parallel
 6:         SampleBSDF($\bar{\mathbf{r}}$, $S$) in parallel
 7:                 ▷ Mark terminated rays for compaction $\mathbf{p}_i \leftarrow -1$
 8:         CompactTerminatedRays($\bar{\mathbf{r}}$, $\bar{\mathbf{p}}$)
 9:         ▷ Remove elements where the ray pixel index $\mathbf{p}_i = -1$
10:     end while
11:     AccumulateColorToImage($I$, $\bar{\mathbf{c}}$) in parallel
12: end for
```

mapping from the ray/thread index i to the pixel index through the relation $\bar{\mathbf{p}}_i = j \in \{1, 2, \ldots, M\}$, so that the ray $\mathbf{r}_{\mathbf{p}_i}$ contributes to the measurement estimate $\hat{I}_j$. This allows us to compact away terminated rays to avoid having warps with idle threads waiting for the rest of the threads to finish the computation.

3.3.2 Stream Compaction

To increase the occupancy of the GPU in scenes where many of the rays are terminated early, we want to rearrange the work load so that only rays that are not terminated are considered and grouped together, maximising the efficiency of each warp. Example scenes where rays are terminated early are outdoor scenes where many rays exit the scene to the surrounding atmosphere.

When the potential radiance contribution of a ray is under a certain threshold or when the ray exits the scene into the background, the ray is marked for termination by setting the corresponding ray pixel index $\mathbf{p}_i = -1$. We then perform stream compaction by removing the elements of the ray pixel index array whose values are negative through the `thrust::remove_if` function of the Thrust library [Bell and Hoberock 2011]. In table 1 we show the number of live rays at each propagation step for an iteration in the renderer, and the number of CUDA thread blocks launched in each step.

Ray depth	Live rays	Thread blocks
1	786677	3600
2	431576	3073
3	199922	1686
4	106416	781
5	64018	416
6	40786	251
7	26987	160
8	18265	106
9	11389	72
10	5776	45
11	2310	23

Table 1: *As the rays are propagated along the path being sampled, they will eventually terminate. If we compact away the terminated rays, we can utilize as much of the computational power of the GPU as possible by dedicating threads only to rays that are still being propagated.*

If we would not compact away the rays that have been terminated, we would have a large number of blocks that are doing very little work as most of the threads assigned to the terminated rays may be waiting idle for the few threads assigned to the live rays to terminate. By performing the stream compaction, we can guarantee that the GPU is launching thread blocks with threads assigned only to live rays that are doing useful work.

We can see however that once we reach a certain ray depth along the path, many rays have terminated and there may be too few rays alive to fully occupy the GPU. Aila and Laine [2009] uses a large pool of rays to further increase the occupancy of the GPU, ensuring that each thread is always tasked with a vast amount of work by performing computations on the pool of rays. We leave the implementation of their paper as future work.

3.4 Reflection and Refraction

We implement reflective and transmissive materials by computing the reflection and transmission directions using Snell's law when a ray intersects a surface that exhibits these properties. To allow for glossy materials as well as purely reflective and transmissive materials, we utilize Russian roulette sampling to decide if we will take a reflection sample, a refraction sample or a diffuse sample. This procedure has the same expected outcome as splitting the ray path and following each of the sub-paths. To do this, we compute the probability of reflection R using the Fresnel equations for unpolarised light. We generate a uniform random number $u \sim U(0, 1)$ and propagate the ray along the reflected direction if $u < R$. If $u > R$, we propagate along the transmitted direction if the material is transmissive and take a diffuse sample if the material is not, resulting in the glossy appearance of some spheres in e.g. Fig 4.

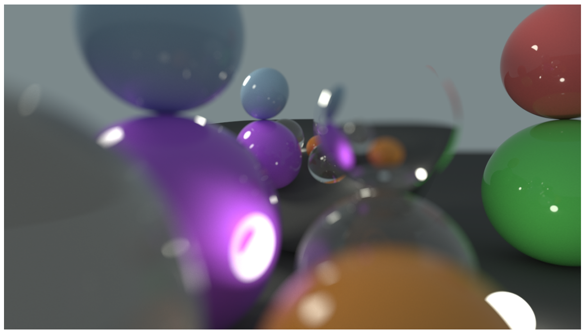
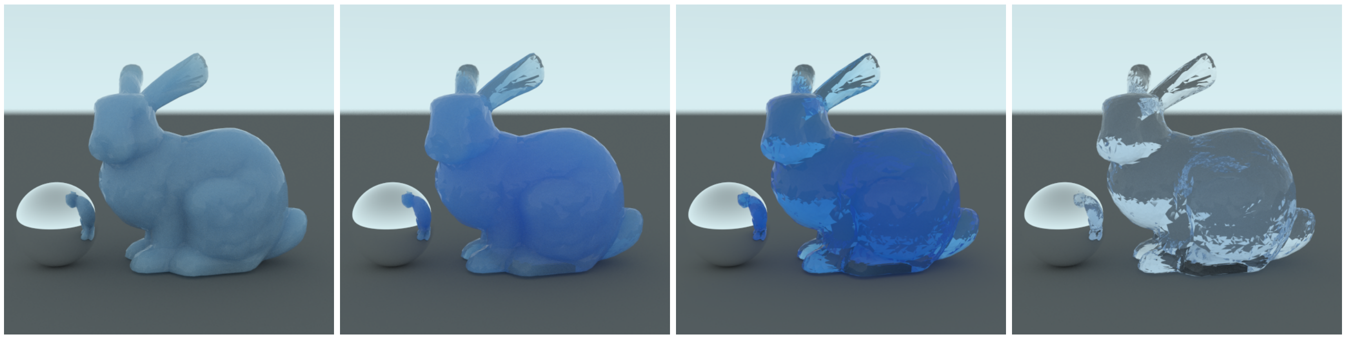


Figure 4: *Sphere scene rendering showing the depth of field effect resulting from finite radius aperture sampling as well as reflection, refraction and glossy materials.*

3.5 Isotropic Subsurface Scattering

To render things like milk, marble and skin, the scattering and propagation of light inside the material needs to be considered. In this renderer we implement isotropic Monte Carlo subsurface scattering which is a computationally intensive brute-force method to simulate these effects. We implement it by letting each ray be aware of which medium it is propagating through. If the ray has been transmitted through a material, we assign the medium parameters to the ray, which holds information about the absorption and scattering properties of the material that ultimately determines the appearance of the material.

The implementation consists of a scattering step and an absorption step. We use the reduced scattering coefficient σ_s, which is the inverted reduced mean free path to determine how far the ray is scattered before an absorption event. The reduced mean free path is the equivalent of a series of mean free path steps that precede an absorption event. Since we only consider homogeneous materials



(a) $\sigma_s = 64$ **(b)** $\sigma_s = 16$ **(c)** $\sigma_s = 4$ **(d)** $\sigma_s = 1$

Figure 5: *Four renderings of the Stanford bunny [Levoy et al. 2005] for reduced scattering coefficients σ_s of (a) 64, (b) 16, (5c) 4, and (d) 1. In (d), the absorption coefficient has been reduced, resulting in a brighter blue color. A higher scattering coefficient is equivalent to a shorter mean free path of the particles being scattered and results in less transparency in the material, and the color of the material is determined by the absorption.*

in this project, we can compute the distance d travelled by the ray before an absorption event by sampling a uniform random variable $u \sim U(0,1)$ and set $d = -\log(u)/\sigma_s$. This can be derived from the cumulative probability density function of a photon interaction with a participating medium at a position along a direction [Jensen and Christensen 1998].

In addition to propagating the ray after being scattered, we compute the absorption. This is done using the Beer-Lambert law that relates the absorption with the properties of the medium that a light ray is travelling through [Swinehart 1962].

$$T = \exp\left(-\int_0^d \mu(z)dz\right) \tag{5}$$

T is the fraction of light that is transmitted in an absorption event for a ray that has travelled a distance d and $\mu(z)$ is the absorption coefficient. In this project we only consider materials with uniform absorption coefficient $\mu(z) = \mu$, and the Beer-Lambert law is reduced to $T = e^{-\mu d}$. For a more complete derivation and account on the theory of radiation scattering, see the work by Chandrasekhar [1960].

Fig. 5 displays the visual properties of subsurface scattering for different parameter values. In (a) the reduced scattering coefficient σ_s is high which leads to a low expected value for the sampled travelled distance of the ray d. As a result, a ray will most likely not travel far into the material before it is scattered back out through the surface again, resulting in the almost solid appearance. As the reduced scattering coefficient σ_s reduced, the the sampled scattered distance will be longer on average, and more rays will exit the object through a point on the surface that is farther away from where it entered. Note also how the color of the object appears darker the more transparent it is, since a larger d gives a larger negative exponent in the Beer-Lambert law (5). In (d) the absorption coefficient μ has been reduced as well, resulting in a bright transparent blue.

4 Future Work

When the triangle meshes exhibit finer detail, the triangle count grows very large and the procedure of checking each ray for intersection with each triangle quickly becomes infeasible. A common strategy to circumvent this is to introduce an acceleration structure such as k-d trees, octrees or bounding volume hierarchies. The overarching strategy is to reduce the number of ray-triangle intersection tests needed by instead testing for intersection against a volume that conservatively encapsulates all the other objects. If a ray misses the conservative volume, then trivially there is no reason to check if the ray would intersect any of the objects that the volume contains. The volume is then successively partitioned further so that the majority of the ray-triangle intersections can be avoided. It is essential to implement such an acceleration structure to improve the real-time performance of the renderer if the scenes are allowed a higher complexity than those used in this project.

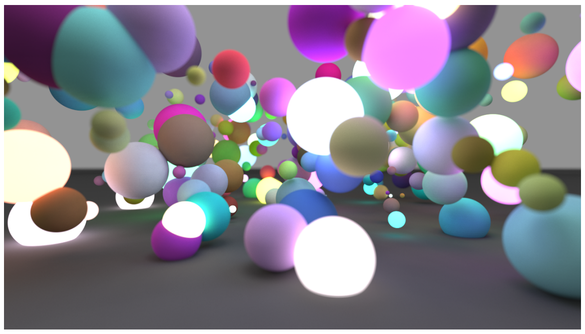


Figure 6: *Scene consisting of 300 spheres with randomly generated diffuse color, emission, position and radius.*

In this project we have only sampled paths starting from the camera, scattering and propagating throughout the scene until they are terminated. This works well for well-lit scenes such as in Fig. 8, where most paths contribute to the final image due to the emission of the background. If a path does not hit a source that emits light, the path will not contribute to the final image. In cases where the light sources of a scene are small or highly occluded, the variance in the computed image will be high because of this and the image will require far more samples to converge. To reduce variance and increase the efficiency of the renderer, it is beneficial to sample light sources directly and trace paths starting from the light sources. One can then use multiple importance sampling to combine paths sampled from the light sources with paths sampled from the camera, connecting vertices of camera paths with vertices from light paths and computing their contribution to the final image [Veach and Guibas 1995]. This method is known as bidirectional path tracing and has been shown to significantly reduce variance, especially in scenes with complex light paths involving reflections, refractions and other indirections between the camera and light sources [Veach

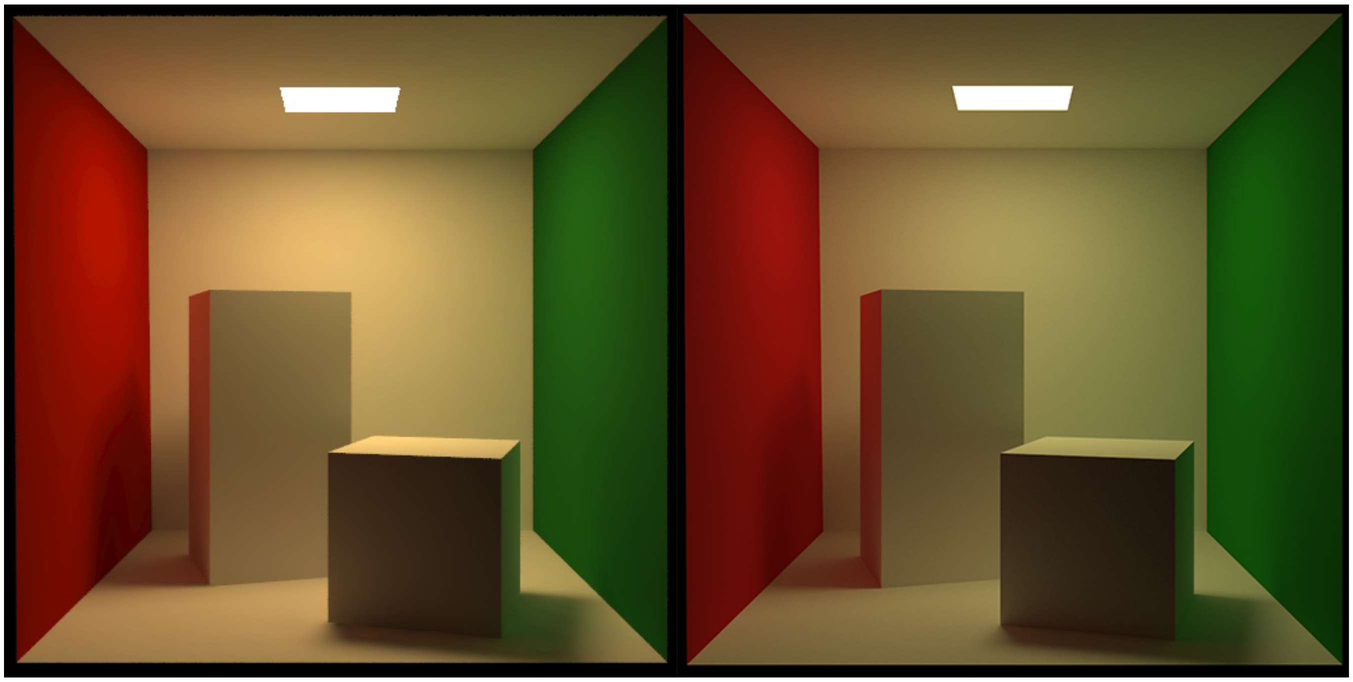


Figure 7: *A comparison between a reference image of the Cornell box scene generated from measured data (left) and an image generated in the path tracer developed in this project (right). The scene geometry and camera is modelled after the measured data. The exact colors of the wall, the emission of the light source and the camera field of view is unknown and the right image was thus generated after selecting colors and camera parameters that generated an image resembling the reference by inspection. The image and scene data was retrieved from http://www.graphics.cornell.edu/online/box/data.html (accessed: 2016-09-24).*

and Guibas 1995; Veach 1997].

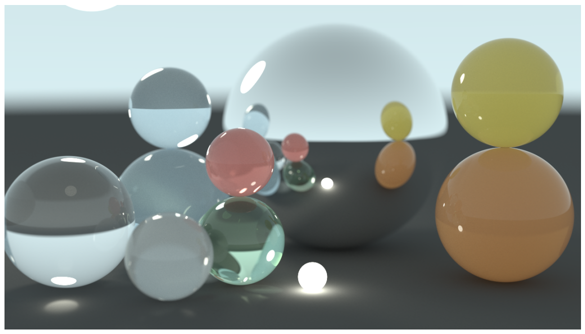


Figure 8: *Rendering of a scene with materials exhibiting subsurface scattering, reflections and refraction in a well-lit environment. The simulation converges faster when the scene background is brighter and most light paths contribute to the final image.*

Ultimately, the list of improvements to a system like the one implemented in this project can be made arbitrarily long. In addition to the above as well as performing code restructuring, optimizations and implementing efficient hand-tuned ray tracing frameworks [Aila and Laine 2009; Laine et al. 2013], some potential extensions are

- Microfacet models
- Anisotropic subsurface scattering using the Henyey-Greenstein phase function [Henyey and Greenstein 1941]
- Texture, normal, displacement and environment mapping
- Metropolis light transport [Veach and Guibas 1997]

Acknowledgements

Peter Hedman at University College London for discussions on modern acceleration structures. Samuel Lapere at the University of Auckland for clarifications regarding color conversion. Samuel Lapere, Peter Kutz and Karl Li for implementation details and examples.

References

AILA, T., AND LAINE, S. 2009. Understanding the efficiency of ray traversal on GPUs. In *Proceedings of the conference on high performance graphics 2009*, ACM, 145–149.

BELL, N., AND HOBEROCK, J. 2011. Thrust: A productivity-oriented library for CUDA. *GPU computing gems Jade edition 2*, 359–371.

BIKKER, J., AND VAN SCHIJNDEL, J. 2013. The Brigade renderer: A path tracer for real-time games. *International Journal of Computer Games Technology 2013*.

CHANDRASEKHAR, S. 1960. *Radiative transfer*. Courier Corporation.

DAHLBERG, H., 2016. Portfolio. http://henrikdahlberg.portfoliobox.net/dh2323projectblog. Accessed: 2016-09-23.

HENYEY, L. G., AND GREENSTEIN, J. L. 1941. Diffuse radiation in the galaxy. *The Astrophysical Journal 93*, 70–83.

JENSEN, H. W., AND CHRISTENSEN, P. H. 1998. Efficient simulation of light transport in scenes with participating media using photon maps. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, 311–320.

KAJIYA, J. T. 1986. The rendering equation. In *ACM Siggraph Computer Graphics*, vol. 20, ACM, 143–150.

KUTZ, P., AND LI, K. 2012. Peter and Karl's GPU path tracer. University of Pennsylvania.

LAINE, S., KARRAS, T., AND AILA, T. 2013. Megakernels considered harmful: wavefront path tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference*, ACM, 137–143.

LAPERE, S., 2016. Ray Tracey's blog. http://raytracey.blogspot.com. Accessed: 2016-09-23.

LEVOY, M., GERTH, J., CURLESS, B., AND PULL, K. 2005. The stanford 3d scanning repository. *URL http://www-graphics. stanford. edu/data/3dscanrep*.

PHARR, M., AND HUMPHREYS, G. 2004. *Physically based rendering: From theory to implementation*. Morgan Kaufmann.

SWINEHART, D. 1962. The Beer-Lambert law. *J. Chem. Educ 39*, 7, 333.

VEACH, E., AND GUIBAS, L. J. 1995. Optimally combining sampling techniques for Monte Carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, ACM, 419–428.

VEACH, E., AND GUIBAS, L. J. 1997. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, ACM Press/Addison-Wesley Publishing Co., 65–76.

VEACH, E. 1997. *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University.