Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction
Specialization: Computer Graphics

Focal Path Guiding

MASTER THESIS

Author: Bc. Petr Šádek
Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Year: 2025

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Šádek Petr** Personal ID number: **474479**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Graphics and Interaction**

Study program: **Open Informatics**

Specialisation: **Computer Graphics**

II. Master’s thesis details

Master’s thesis title in English:

Focal Path Guiding

Master’s thesis title in Czech:

Navádění světelných cest podle ohnisek

Guidelines:

Review algorithms using path guiding for realistic image synthesis. Based on the recent publication of Rath et al. [1], design and implement an algorithm identifying focal points (places with a high concentration of light paths in space), which are subsequently used for efficient light sampling of light paths in the path tracing method. The implementation will make an efficient use of the GPU through the Falcor framework. Test the implementation on at least five different scenes and determine for which scenes the method provides improvements in path tracing convergence speed. Identify the bottlenecks and possible improvements of the original method.

Bibliography / sources:

[1] Rath, Alexander, Ömercan Yazici, and Philipp Slusallek. "Focal Path Guiding for Light Transport Simulation." ACM SIGGRAPH 2023 Conference Proceedings. 2023.
[2] Müller, Thomas, Markus Gross, and Jan Novák. "Practical path guiding for efficient light transport simulation." Computer Graphics Forum. Vol. 36. No. 4. 2017.
[3] Bitterli, B., Wyman, C., Pharr, M., Shirley, P., Lefohn, A., & Jarosz, W. (2020). Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. ACM Transactions on Graphics (TOG), 39(4), 148-1.
[4] Ouyang, Y., Liu, S., Kettunen, M., Pharr, M., & Pantaleoni, J. (2021). ReSTIR GI: Path resampling for real time path tracing. In Computer Graphics Forum (Vol. 40, No. 8, pp. 17-29).
[5] Vorba, J., Hanika, J., Herholz, S., Müller, T., Křivánek, J., & Keller, A. (2019). Path guiding in production. In ACM SIGGRAPH 2019 Courses (pp. 1-77).
[6] NVIDIA Falcor framework. https://developer.nvidia.com/falcor

Name and workplace of master’s thesis supervisor:

doc. Ing. Jiří Bittner, Ph.D. Department of Computer Graphics and Interaction

Name and workplace of second master’s thesis supervisor or consultant:

Date of master’s thesis assignment: **15.02.2024** Deadline for master's thesis submission: **07.01.2025**

Assignment valid until: **21.09.2025**

doc. Ing. Jiří Bittner, Ph.D.	Head of department’s signature	prof. Mgr. Petr Páta, Ph.D.
Supervisor’s signature		Dean’s signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.	
Date of assignment receipt	Student's signature

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses. In my thesis, I used the Writefull Overleaf plugin for grammar correction. I did not use any other artificial intelligence tool.

Prague, date .....

.....
Bc. Petr Šádek

Acknowledgments

I would like to thank doc. Ing. Jiří Bittner, Ph.D. for supervising my thesis, his feedback and suggestions.

Bc. Petr Šádek

Title:
Focal Path Guiding

Author: Bc. Petr Šádek

Supervisor: doc. Ing. Jiří Bittner, Ph.D.

Abstract: Main objective of this thesis is the reimplementaion of focal path guiding algorithm in the real-time rendering framework Falcor from NVIDIA. We start with an introduction to the problematic of ray tracing. Then we go through several ray tracing algorithms. Ending with the mentioned focal path guiding algorithm. We describe the implementation written in GPU shaders. In the end, we evaluate this implementation on different scenes and identify its strengths and weaknesses.

Key words: ray tracing, path guiding, photorealistic computer graphics

Název práce:
Navádění světelných cest podle ohnisek

Autor: Bc. Petr Šádek

Vedoucí práce: doc. Ing. Jiří Bittner, Ph.D.

Abstrakt: Hlavním cílem této práce je reimplementace algoritmu navádění světelných cest podle ohnisek v real-time renderovacím frameworku Falcor od NVIDIA. Začneme úvodem do problematiky vrhání paprsků. Poté se podíváme na různé metody vrhání paprsků. Jako poslední z nich popíšeme zmíněný algoritmus navádění světelných cest podle ohnisek. Popíšeme implementaci napsanou v GPU shaderech. Na konec zhodnotíme tuto implementaci na různých scénách a identifikujeme její kvality a nedostatky.

Klíčová slova: vrhání paprsků, navádění cest, fotorealistická počítačová grafika

Contents

Acronyms

AABB Axis-Aligned Bounding Box

BRDF Bidirectional Reflectance Distribution Function

BSDF Bidirectional Scattering Distribution Function

CPU Central Processing Unit

FPS Frames Per Second

GI Global Illumination

GPU Graphics Processing Unit

GUI Graphical User Interface

IS Importance Sampling

MIS Multiple Importance Sampling

MSE Mean Squared Error

PDF Probability Density Function

RIS Resampled Importance Sampling

ReSTIR Reservoir-based Spatio-Temporal Importance Resampling

List of Figures

Chapter 1

Introduction

Focal path guiding is a specialized method of ray tracing. Ray tracing is a general term for computer graphics methods that use simulation of rays of light to compute (photorealistic) images of an artificial scene. There are many different approaches to ray tracing. We will go over some of them in the following chapter. We will start with the necessary theory to describe the distribution of light in the scene. Then we will go over the methods, starting with the simple ones, then continuing to more complicated ones, and finally ending with the main focus of this thesis - focal path guiding.

The distinguishing aspect of this method is its ability to recognize focal points. Focal points are, in general, some small areas of space with high density of light passing through them. Some examples of focal points are caustics or focal points of lenses. This method is the first one that generalizes for handling of all possible types of focal points. Up until now, there were only methods that specialize just for a subset of them. This method is even capable of effectively rendering the famous camera obscura scene (figure 1.1).

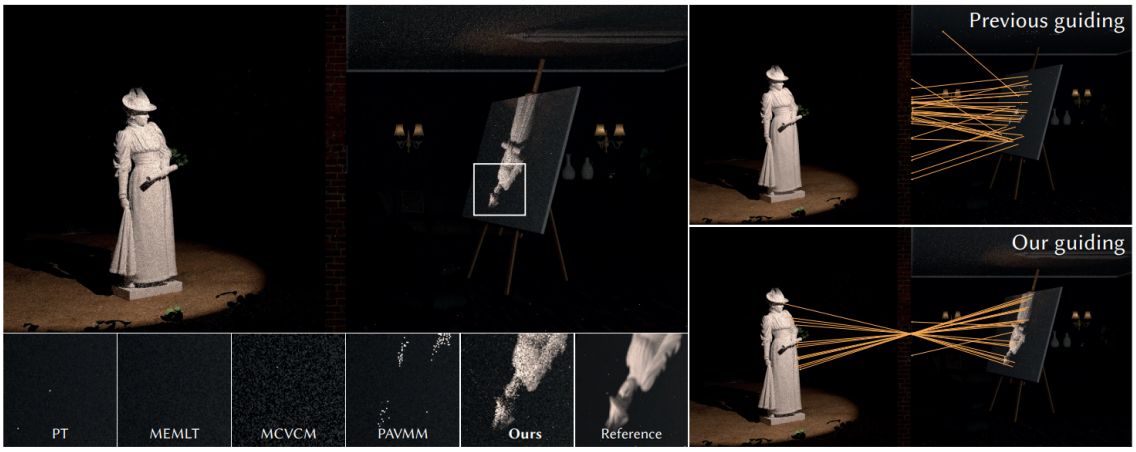


Figure 1.1: Camera obscura scene comparison of multiple methods (Rath et al. 2023 [1]).

The main goal of this thesis is to re-implement the focal path guiding algorithm from Rath et al. 2023 [1] in real-time rendering framework Falcor [2] from NVIDIA. It will be described in the chapter 3. The hardest challenge is the transition from their CPU implementation to the GPU implementation in Falcor shaders. Some simplifications of the original algorithm must be made for this transition to be possible.

Chapter 2

Ray Tracing

Ray tracing, in general, is a simulation of interactions of rays of light with some artificial scene. Specifically, this thesis concentrates on applications of ray tracing in realistic image synthesis, but the concept of ray tracing is used in many different areas other than computer graphics such as, for example, estimation of signal strength for wireless networks, ballistic analysis, or optical design of lenses.

Ray tracing in computer graphics is a somewhat inverse approach to rasterization. Rasterization projects every visible object in the scene onto the screen and colors the associated pixels. On the other hand, ray tracing goes through every pixel and computes its color based on intersections of shot rays with scene objects.

In most applications, we do specifically backward ray tracing, which means that we are shooting rays of light from the viewer to the light sources. This may be a little counterintuitive, because in reality it happens in the opposite direction. But because we are only interested in light which hits the viewers eyes, it is more effective to shoot rays from the eye, let it bounce in the scene until it hits some light source, and then reconstruct the light contribution from this path as would happen in the opposite direction.

In this chapter, we will first describe required theoretical light physics background - BRDF [3] and the rendering equation [4]. Then we will go through several ray tracing methods. There are many ways to approach ray tracing. Some might be targeting more for computational speed, others for quality, they can be biased/unbiased, or specialized for some specific scene conditions or some specific light effects. We will start by describing the basic path tracing method [4], which is a general approach to solving the rendering equation and works as a foundation for other methods that we will describe. After that, we will continue with ReSTIR [5], [6], [7], which introduces a smart way of sampling to make path tracing more effective. In the end, we will look at path guiding [8], [9] and its variant focal path guiding [1], which is the main topic of this thesis.

2.1 Physics background

To express the amount of light at some point in space and in some direction we will use physical quantity radiance, mostly represented by the symbol L with unit $[Wm^{-2}sr^{-1}]$. It basically means how many photons are emitted in some direction in unit solid angle on unit area in one unit of time.

2.1.1 BRDF

Now we need to describe how the light interacts with some surface with some kind of material. Basic light interactions with some material at some point in space can be described with the relationship of incoming and outgoing radiance. The function describing this relationship is called the bidirectional reflectance distribution function (BRDF [3]) and looks like this:

$$f\left(\omega_i, x, \omega_o\right) = \frac{dL_o\left(x, \omega_o\right)}{dE\left(x, \omega_i\right)} = \frac{dL_o\left(x, \omega_o\right)}{L_i\left(x, \omega_i\right)\cos\theta_i d\omega_i}$$

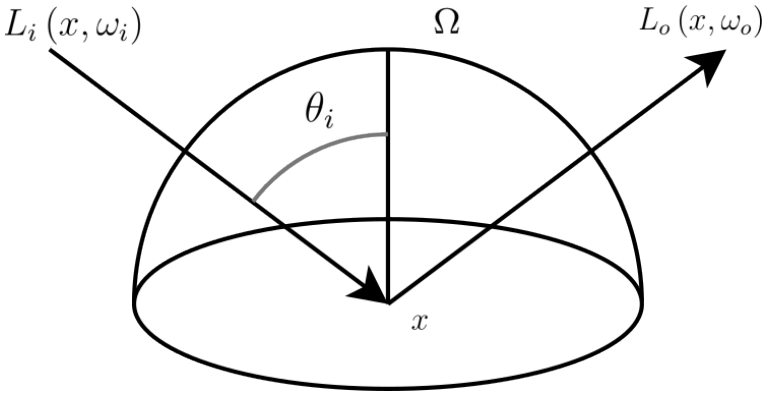


Figure 2.1: BRDF terms visualized.

Where f is the BRDF, x is a point on the surface, ω_i and ω_o are the incoming and outgoing directions and θ_i is an angle between the incoming direction and the surface normal. You can see these terms visualized on figure 2.1. We can see that it is simply the ratio of change in outgoing radiance L_o and change in incoming irradiance E. Irradiance is similar to radiance, but it's not dependent on the solid angle (the unit is $[Wm^{-2}]$). Therefore, there is a relationship between radiance and irradiance that we can use to obtain the final identity with only radiances. This function could also be defined for each light wavelength separately, but in computer graphics we typically reduce the problem to just RGB.

With the BRDF we can express many different properties of the material, such as the overall color by changing how much of the light is reflected and how much is absorbed for each color channel, or for example glossiness by increasing the reflected amount around the perfect reflection direction. There are many illumination models for describing the BRDF. One of the most basic ones is Phong's illumination model, which divides the light into 2 components, diffuse and specular. But there are more complicated models such as Cook-Torrance or Ward.

2.1.2 Reflection Equation

Now that we have the description of interaction with the surface materials, it's time to compute how much radiance we actually see when looking at some point from some direction. From the BRDF formula, we can derive:

$$\begin{aligned} f(\omega_i, x, \omega_o) &= \frac{dL_o(x, \omega_o)}{L_i(x, \omega_i)\cos\theta_i d\omega_i} \\ dL_o(x, \omega_o) &= L_i(x, \omega_i) f(\omega_i, x, \omega_o)\cos\theta_i d\omega_i \\ L_o(x, \omega_o) &= \int_\Omega L_i(x, \omega_i) f(\omega_i, x, \omega_o)\cos\theta_i d\omega_i \end{aligned}$$

In this way, we can obtain the total reflected radiance by integrating over a hemisphere Ω of all input angles. Also, the material itself could emit some light, so we will add the term L_e and we will get the local reflection equation:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f(\omega_i, x, \omega_o)\cos\theta_i d\omega_i$$

2.1.3 Rendering Equation

However, the previous equation describes only the local behavior of the light. It ignores the fact that the light may bounce multiple times before reaching its destination. So, we will introduce a function $r(x, \omega_i)$ that returns the surface position y from where the light comes. Now we can describe the input radiance as the output radiance of the previous reflection.

$$L_i(x, \omega_i) = L_o(y, -\omega_i) = L_o(r(x, \omega_i), -\omega_i)$$

By applying this to the local reflection equation, we will finally get the rendering equation [4], which describes global light propagation in the scene. For simplicity, we will replace L_o with just L.

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L(r(x, \omega_i), -\omega_i) f(\omega_i, x, \omega_o)\cos\theta_i d\omega_i$$

We could use another form of this equation, which is more useful in some scenarios. Instead of integrating over all directions on the hemisphere Ω, we could integrate over all surfaces in the scene S. We achieve this by substituting $d\omega = dA\frac{\cos\Theta}{r^2}$.

$$L(x, \omega_o) = L_e(x, \omega_o) + \int_S L(y, y \to x) f(x \to y, x, \omega_o) G(x, y) V(x, y) dA$$

Where y is the point from which the light comes, $y \to x$ is a unit directional vector from y to x, $G(x, y) = \frac{\cos\Theta_x \cos\Theta_y}{r^2}$ is the geometric term, which decreases the incoming radiance by distance r and by cosines of angles to the surface normals, $V(x, y)$ is the visibility term, which is 1 when y is visible from x and 0 otherwise.

2.2 Monte Carlo Integration

Now that we have the model for describing global illumination, it's time to compute our radiance at some desired position and direction by solving the rendering equation. But this is a very hard problem, the equation contains something like an infinite recursive integral. Even for very simple scenes, it is nearly impossible to solve this problem analytically. So, we will have to solve it numerically, in this case specifically by Monte Carlo integration [10].

First, we will show how to integrate locally with one bounce with different sampling methods. We will then extend it to paths with multiple bounces, and thus using the path tracing algorithm.

2.2.1 Uniform Sampling

We will first simplify the rendering equation in the form with integral over surfaces. We will drop the surface point x and the viewing direction ω_o (look at them as inner parameters). We will denote the function inside the integral as l and rename the point y to s.

$$l(s) = L(s, s \to x)\, f(x \to s, x, \omega_o)\, G(x, s)\, V(x, s)$$

$$L = \int_S l(s)\, dA$$

Now we can approximate this integral by uniformly generating N random samples $s_1 \dots s_N$. And averaging their l function results:

$$\langle L \rangle^N_{uniform} = \frac{1}{N} \sum_{i=1}^{N} l(s_i)$$

This is the simplest Monte Carlo estimator. Increasing the number of samples should increase the quality of the approximation. It works, but it converges to the correct solution very slowly. Because we are sampling all the points with the same uniform probability, we will generate many samples whose contribution to the result is minimal (for example, points in the darker places in the scene).

2.2.2 Importance Sampling

To solve the problems of uniform sampling, we could sample the points with different probabilities $p(s_i)$ based on how important they are - how much they contribute to the result (thus importance sampling). However, for the result to remain unbiased (the average should converge to the integral), we need to weight the results by $1/p(s_i)$.

$$\langle L \rangle^N_{is} = \frac{1}{N} \sum_{i=1}^{N} \frac{l(s_i)}{p(s_i)}$$

In this way, the results of more probable samples will be decreased, and the results of less probable samples will be increased, which balances the fact that more probable samples will be generated more often.

If the probabilities are chosen well, then we converge faster to better solutions. The problem is that we do not know the actual underlying distribution. However, we can use some pretty good estimates that somehow correlate with the results of the l function. We could use BRDF sampling, which samples directions with higher BRDF with higher probability (for example, sampling according to diffuse and specular components in the Phong reflectance model [11]), or we could use light source sampling, which samples directions to light sources depending on their emitted power (for example, sampling directions in environmental map based on intensities at each pixel [12]).

2.2.3 Multiple Importance Sampling

We can improve the previous approach by combining multiple probability density functions (MIS [13]). For example, combining the BRDF sampling with light source sampling. If we have M such sampling methods, we generate $N_1 \dots N_M$ numbers of samples for each method, then we combine the results with the following estimator:

$$\langle L \rangle_{mis}^{M,N} = \sum_{j=1}^{M} \frac{1}{N_j} \sum_{i=1}^{N_j} w_j\left(s_i\right) \frac{l\left(s_{j,i}\right)}{p_j\left(s_{j,i}\right)}$$

We have added another weight $w_j\left(s_i\right)$, which should compensate for the different sampling of different methods. It is recommended to use, for example, the balance heuristic:

$$w_j\left(s\right) = \frac{N_j p_j\left(s\right)}{\sum_{k=1}^{M} N_k p_k\left(s\right)}$$

If we use the balance heuristic and the numbers of samples are all the same, then the estimator simplifies to the following:

$$\langle L \rangle_{mis}^{M,N} = \frac{M}{N} \sum_{j=1}^{M} \sum_{i=1}^{N/M} \frac{l\left(s_{j,i}\right)}{\sum_{k=1}^{M} p_k\left(s_{j,i}\right)}$$

So, in the end instead of dividing by one probability, we divide by sum of probabilities of the current sample with respect to all the sampling methods. This ensures that the better sampling method for the specific scenario has more impact on the result.

2.2.4 Path Tracing

In path tracing [4], we recursively create a path of multiple bounces. At each bounce, we compute the light source sample and add it to the result. At each bounce, we randomly decide based on the albedo of the material whether to continue the path or not (if we don't continue, then it means the light was absorbed by the material). If we decide to continue, then we create a new bounce by computing the BRDF sample. Similarly to integrating multiple outgoing samples in methods with just one bounce, we will integrate multiple path samples by uniformly averaging their results.

This algorithm is very simple to implement. It should be unbiased and should slowly converge to correct solutions. So, it is mostly used as a reference algorithm for comparison with some other more optimized algorithms. You can see simple depiction of path tracing on figure 2.2.

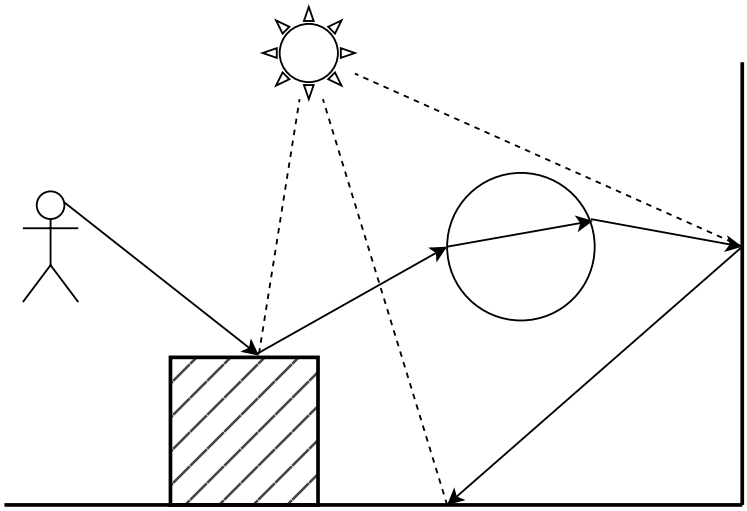


Figure 2.2: Path tracing visualized.

2.3 Scene Intersections

We should also mention one complicated part of ray tracing, which is finding ray intersections with the scene objects. It is not the main focus of this thesis, but it has great impact on the performance of ray tracing algorithms and it is the reason that rasterization is more beneficial in terms of performance for computation of direct illumination. The problem is usually solved by using some kind of hierarchical data structure, which accelerates spatial search of scene objects (typically triangles), giving us candidates of objects that could intersect the ray. For each of these candidates, we perform an intersection test to check if the intersection actually happens. The popular data structures for this task with best practical results are kd-trees [14] and BVH (bounding volume hierarchy [15]) trees. We will use similar types of data structure in this thesis for path guiding, namely an octree [16].

2.4 ReSTIR

In this section, we will describe the Reservoir-based Spatio-Temporal Importance Resampling (ReSTIR [6]) algorithm, which drastically improves the render time performance in comparison with the classical path tracer. First, we will describe a variant of this algorithm limited to direct illumination, starting with a new sampling algorithm (RIS [5]), then simplifying this sampling process by using reservoirs and then reusing samples by combining multiple reservoirs. In the end, we will look at a generalized variant of this algorithm for computing global illumination [7].

2.4.1 Resampled Importance Sampling

The main principle of Resampled Importance Sampling (RIS [5]) is that we generate M samples $\{s_1, \ldots, s_M\}$ according to some simple PDF p and then randomly select one of them with probabilities proportional to some more complicated PDF $\hat{p}$. The simple PDF p should be easy to sample from, it could be, for example, light source sampling. The complicated PDF $\hat{p}$ should be something that better represents the actual underlying distribution of radiance in the scene, but we may not be able to sample from it. We could choose $\hat{p}$ as the unshadowed light contribution of the sample $L(s, s \to x) f(x \to s, x, \omega_o) G(x, s)$, which is basically the inside of the integrand in the rendering equation without the visibility term $V(x, s)$. In this way, by selecting the samples proportional to $\hat{p}$ we approximate the underlying distribution of radiance in the scene and thus select samples with higher contributions more often.

We will now concentrate on how to select the sample with probability proportional to $\hat{p}$ from M generated candidates. In our case, if we choose $\hat{p}$ to be the mentioned unshadowed light contribution, then $\hat{p}$ is not even a valid PDF, because the result of the integration over it may be greater than 1. But that won't be a problem, because we will construct a new valid PDF from $\hat{p}$ for selection of the samples in the following way. First, we will define weights w as ratio of complicated PDF $\hat{p}$ and simple PDF p:

$$w(s) = \frac{\hat{p}(s)}{p(s)}$$

Then the probability of selecting index $i \in \{1, \ldots, M\}$ from candidate samples $\mathbf{s} = \{s_1, \ldots, s_M\}$ is:

$$p(i \mid \mathbf{s}) = \frac{w(s_i)}{\sum_{j=1}^{M} w(s_j)}$$

We can clearly see that this PDF sums up to 1 and the probabilities are proportional to $\hat{p}$. Now we can select a sample $y = s_i$ by randomly generating a number from 0 to 1 and then finding the index i by searching in which of the intervals of the $p(i \mid \mathbf{s})$ the generated number belongs to. From this sample we can compute 1-sample RIS estimator:

$$\langle L \rangle_{ris}^{1,M} = \frac{l(y)}{\hat{p}(y)} \cdot \frac{1}{M} \sum_{j=1}^{M} w(s_j)$$

If $\hat{p}$ is the PDF from which we sample, then we could simply use the $l(y)/\hat{p}(y)$ term. But because it is not the case, we compensate it by the average of the weights of the M samples. Finally, we can get N sample estimator by repeating this process N times and averaging the results, each time we select from new M samples.

$$\langle L \rangle_{ris}^{N,M} = \frac{1}{N} \sum_{i=1}^{N} \left(\frac{l(y_i)}{\hat{p}(y_i)} \cdot \frac{1}{M} \sum_{j=1}^{M} w(s_{i,j}) \right)$$

2.4.2 Weighted Reservoir Sampling

We can simplify the process of selecting the y sample from the M generated candidates. Instead of first generating all of the M samples, remembering their weights w and after that selecting one sample, we could select the resulting sample in the process of generating the M samples and don't remember the individual samples and their weights.

So, we generate the samples one by one. At each step, we remember sum of all the processed weights w_{sum} up to this point. We also remember the currently selected sample y, in the beginning we could set it to the first generated sample. At each step we generate a new sample s_i, compute it's weight w_i, generate random number $r \in \langle 0, 1 \rangle$. If $r < w_i / w_{sum}$, then we set the resulting sample to s_i.

This procedure is equivalent to the selection of samples in RIS. If we add a new sample s_{m+1} to the reservoir of m already processed samples, then the probabilities of selecting the samples are:

$$p(m+1) = \frac{w(s_{m+1})}{\sum_{j=1}^{m+1} w(s_j)}$$

$$p(i) = \frac{w(s_i)}{\sum_{j=1}^{m} w(s_j)} \left(1 - \frac{w(s_{m+1})}{\sum_{j=1}^{m+1} w(s_j)} \right) = \frac{w(s_i)}{\sum_{j=1}^{m+1} w(s_j)} \quad \forall i \in \{1, \ldots, m\}$$

2.4.3 Spatiotemporal Reuse

The reservoirs are not only useful for simplifying the local sample selection. If we have 2 reservoirs, we can easily combine them in $O(1)$ time. Let there be 2 reservoirs r_1, r_2 with the selected samples y_1, y_2 and weight sums $w_{sum,1}$, $w_{sum,2}$. Then we can combine them simply by treating y_2 as a new sample, which we put in the r_1 reservoir with weight $w_{sum,2}$ (the same procedure as described in the previous section). The combined result now lies inside the r_1 reservoir.

In this way, we could, for example, combine reservoirs of neighboring pixels. Because their distributions of radiance are probably very similar, it makes sense to apply this reuse. But the distributions are still a little different, so we compensate for this by multiplying the resulting weight sum by $\hat{p}_1(y)/\hat{p}_2(y)$, where $\hat{p}_1$ is the complex PDF of the pixel of the reservoir to which we are adding the result of the second reservoir, whose complex PDF is $\hat{p}_2$.

Spatial reuse does exactly this combining procedure described in previous paragraphs with k neighboring pixels (for example, k could be 8 immediately neighboring

pixels). We repeat this n times and we effectively get a combination of k^n neighboring reservoirs.

We could also combine reservoirs with temporal reuse. This means that we combine the reservoir in the current pixel with the reservoir from the previous frame at the same pixel. The reasoning behind this is similar to that of spatial reuse. If we don't move with the camera drastically, then the distributions should be similar. But we should still compensate with the $\hat{p}_1(y)/\hat{p}_2(y)$ term. After that, if the results that we propagate to the next frame are these combined results, then we are effectively reusing reservoirs from all the previous frames, not just the reservoirs from the previous frame.

2.4.4 ReSTIR Global Illumination

To generalize ReSTIR for global illumination (ReSTIR GI [7]), we sample even the points that are not light sources, instead of sampling only from light sources. The radiance at these points is now computed with path tracing. We can compute RIS with weighted reservoir sampling in the same way as in basic ReSTIR. Spatiotemporal reuse is still being applied in a similar way, but in Ouyang et al. 2021 [7], they made a few changes. Instead of representing the samples as directions, they represent them as positions of surface points. This preserves the samples more accurately when spatial reuse is used. When merging 2 reservoirs, they also compute similarity of the samples and decide whether to merge them or not.

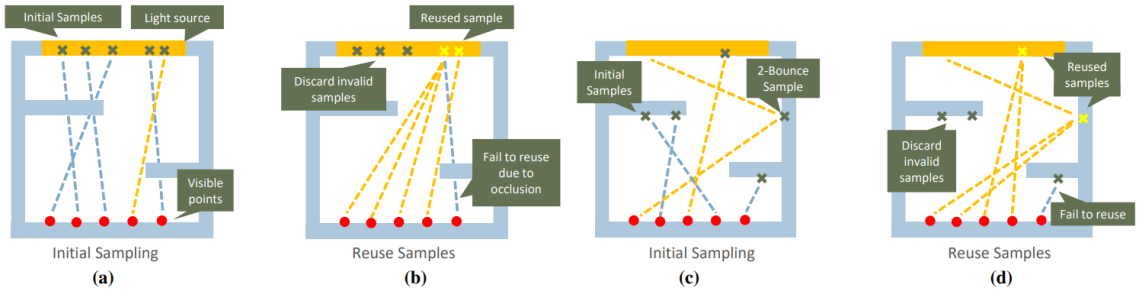


Figure 2.3: Comparison of ReSTIR **(a)**, **(b)** and ReSTIR GI **(c)**, **(d)** from Ouyang et al. 2021 [7].

2.5 Path Guiding

Path guiding is a general term for methods which somehow approximate the global distribution of radiance in the scene and leverage this knowledge for better sampling in the path tracing algorithm. One of those methods is Practical Path Guiding for Efficient Light-Transport Simulation [8].

This method uses hierarchical data structures to divide the space and directions of the scene and stores the approximate amount of radiance inside of the nodes of these data structures. Specifically, it uses a kd-tree for subdivision of space and at each leaf of the spatial kd-tree it subdivides all directions with a quadtree. The spatial kd-tree (or spatial binary tree) divides its nodes always in half, and alternates between the x, y, z axes in order. The directional quadtree uses cylindrical coordinates to represent directions. Similarly to the spatial tree, it subdivides the nodes in half of the intervals. You can see an example of the trees in the figure 2.4.

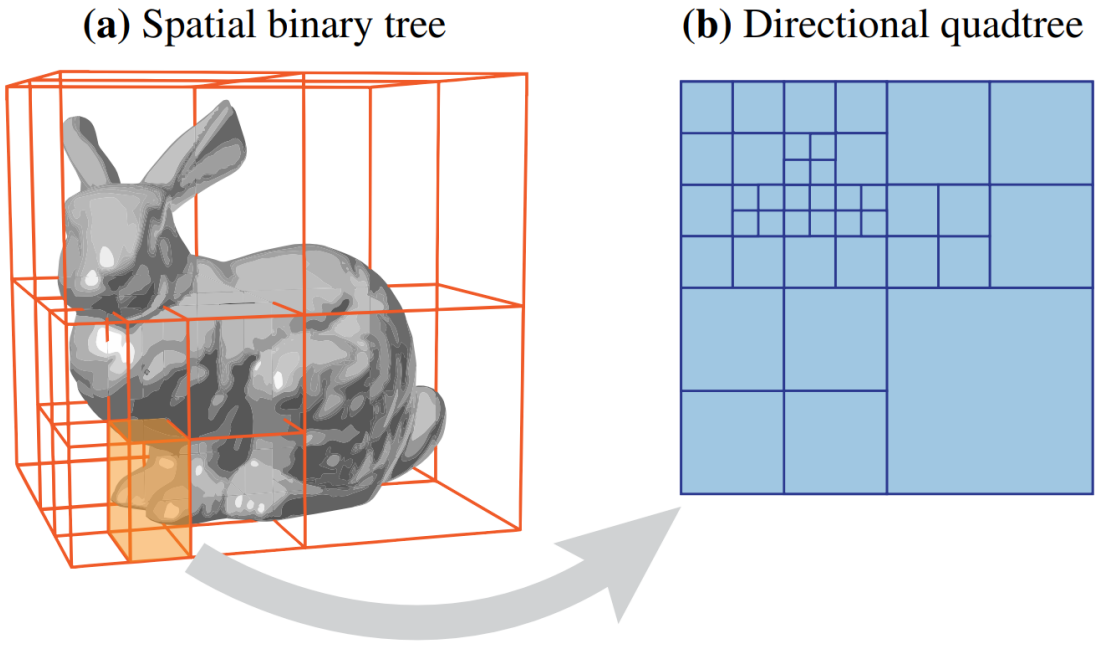


Figure 2.4: Spatio-directional subdivision trees from Müller et al. 2017 [8].

In the algorithm, there are 2 phases that are iteratively repeated. In the first phase, we are collecting estimated radiance inside our tree structures. Based on this, in the second phase, we adapt the trees to our current distribution of radiance by subdivision when some conditions are met.

2.5.1 Collecting Estimated Radiance

In this step, we perform the classical path-tracing. When we form a path, we iterate over all its vertices v, with their respective directions ω. First, we perform a search in the spatial tree and accumulate radiance at each node that contains the position v. We also remember the number of vertices that visited each node. After that, when we reach a leaf node containing v, we will traverse its directional tree in a similar way: visit nodes containing ω and accumulate the estimated radiance.

An important observation is that because we accumulate only inside the spatial nodes that contain the current path vertex v, it means that we approximate only

the distribution of the radiance at the surface. So we have no information about the distribution in the empty space between the surfaces, which may lead to some problems, which are solved by the method described in the section about focal path guiding.

2.5.2 Tree Adaptation

We split the nodes of the spatial kd-tree based on the number of vertices that visited each node. If the number is higher than some tweaked constant, we will split the node. In Müller et al. 2017 [8] they use specifically $c\sqrt{2^k}$, which is proportional to the number of traced paths in the k-th iteration, because they are increasing the number of samples in each iteration exponentially.

The nodes inside the directional quadtree are split based on the ratio of accumulated radiance of the current node and the total accumulated radiance in the current quadtree (value stored at the root node). In Müller et al. 2017 [8] they use the threshold ratio $\rho = 0.01$. The total number of nodes in the quadtree is then proportional to $1/\rho$.

2.5.3 Sampling

Now that we have an approximation of the radiance distribution stored in our spatio-directional trees, it's time to use it for better sampling of outgoing directions (technique from McCool et al. 1997 [17]). The process is simple. We first traverse the spatial kd-tree, to find a leaf node that contains our current vertex v. We then take its directional quadtree. Then we randomly traverse the nodes in depth, with the probability p_n of choosing a specific node equal to $p_n = L_n/L_p$, where L_n is the accumulated radiance of the current node and L_p is accumulated radiance of its parent. When we reach a leaf, we simply generate the sample by randomly choosing one direction from the range that the leaf represents with uniform probability. Then after we compute the incoming radiance of the sample $l(s)$, we will weight the result similarly to IS by $1/p(s)$, where $p(s) = L_n/L_t$, where L_n is the accumulated radiance of the current node and L_t is accumulated radiance of the entire directional quadtree (value stored in root node).

This method of sampling could be used, for example, instead of standard light source sampling in the classical path tracer algorithm, which should have better results for non-analytical light sources. Or, it could be used for sampling new bounces, which should have better results at surface caustics.

2.6 Focal Path Guiding

Focal path guiding [1] is a variant of path guiding, which specializes in capturing the so-called focal effects and is the main topic of this thesis. We define term focal point as a small region of space where multiple light paths from different directions converge. Focal effects are then arsing due to presence of focal points in the scene.

There are many different types of focal points. Some are illustrated in figure 2.5. In Rath et al. 2023 [1] they established 3 main categories of focal points: direct focal points, indirect focal points and virtual images. Direct focal points are either small light sources themselves (image a), or the camera position. Indirect focal points are either surface caustics (brightly iluminated spots, image b), or light passing through narrow gaps (camera obscura effect, image c). Virtual images are caused by reflections or refractions (image d), or multiple combinations of them.

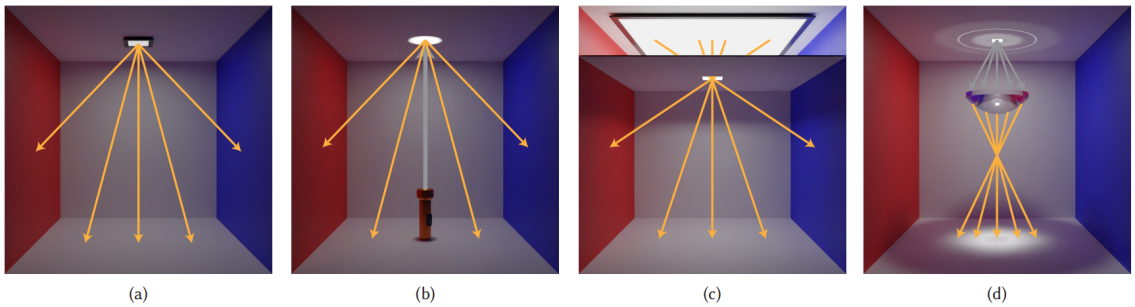


Figure 2.5: Different types of focal points from [Rath et al. 2023 1].

This method should handle all of these types of focal effects well. In contrast, the path guiding method Müller et al. 2017 [8] from previous section is capable of handling direct focal points and some of the indirect focal points, such as surface caustics. But it performs poorly for the other types of focal points that are not surface-bound. The main reason for this behavior is that the data structure from Müller et al. 2017 [8] approximates only the surface distributions, not the distributions in the space between.

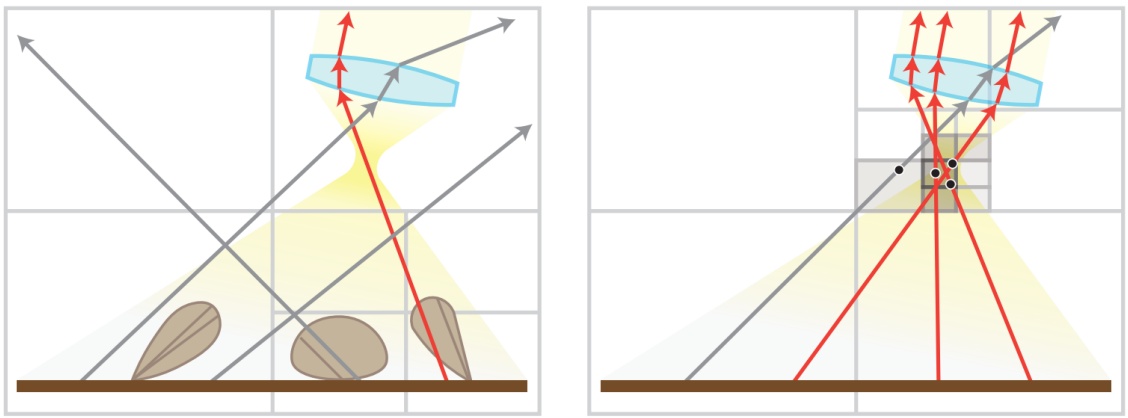


Figure 2.6: Comparison of path guiding from Müller et al. 2017 [8] on the left and focal path guiding on the right. Taken from Rath et al. 2023 [1].

The main difference from the previous method is in the data structure itself and in the method of accumulating estimated radiances inside of it. You can see

comparison of the methods in figure 2.6. Here we use an octree [16] instead of a kd-tree [14] for spatial subdivision (which itself shouldn't change the results, it's more a performance matter, but it would be worth a try to keep the kd-tree and compare), and we drop the dependence on direction, so we don't use the directional quadtrees anymore. The radiance estimates are now accumulated at each node that is intersected with a specific ray from our path, instead of accumulating only at the nodes containing the starting point of the ray. In the following sections, similarly to classical path guiding, we will describe the process of collecting estimated radiance, which we use to estimate focal densities, then adaptation of the tree structure and finally sampling random directions from the tree.

2.6.1 Estimating Focal Densities

In the nodes of the octree data structure, we store accumulated radiance a_v, from which the so-called focal densities p_v can be computed (v represents the voxel of the octree node). First we compute density times volume $\alpha_v = p_v|V_v|$, also called the selection probability:

$$\alpha_v = \frac{a_v}{a_r}$$

Where a_v is accumulator of our current node and a_r is accumulator of the root node. Now we can simply divide by volume $|V_v|$ of the node to get the focal density p_v:

$$p_v = \frac{\alpha_v}{|V_v|}$$

The estimation of focal densities has two iterative phases. The first phase is the initial guess, which roughly places higher densities near our focal points. The second phase is the narrowing phase, which narrows the distribution closer to the true focal points. You can see these phases visualized in the figure 2.7. In Rath et al. 2023 [1] they use 10 iterations for the initial guess and 5 for the narrowing phase.

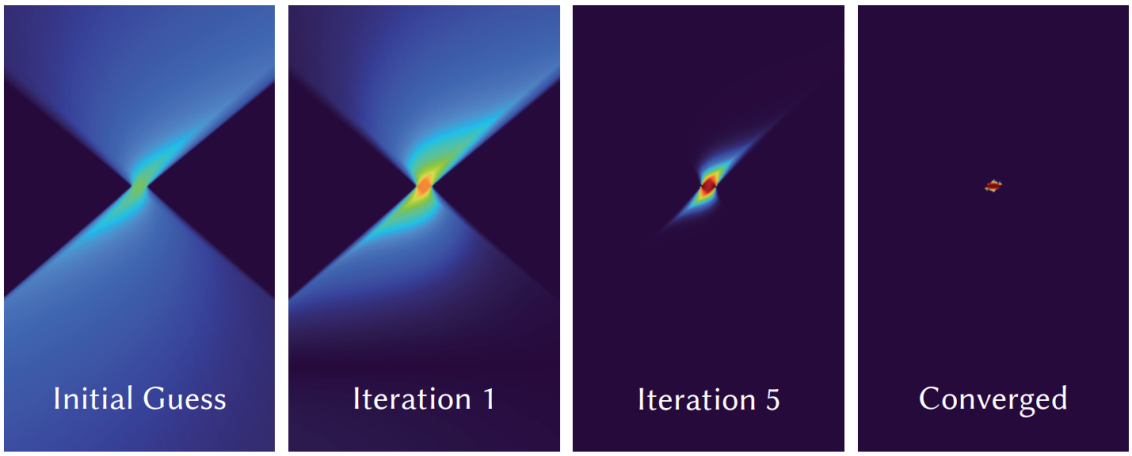


Figure 2.7: Visualization of the focal density estimation, from Rath et al. 2023 [1].

The initial guess phase is simple. We perform a classical path tracing for each pixel with multiple samples. For each bounce in the path, determined by generated sample point s, we go through every octree node voxel v that is intersected by the ray and we add the following value to its accumulator a_v:

$$(t_1 - t_0)\frac{l(s)}{p(s)}$$

Where $(t_1 - t_0)$ is the distance traveled through the voxel (t_0 and t_1 are the entry and exit point distances from the ray start), $l(s)$ is estimated radiance at sample s (same notation as used so far) and $p(s)$ is the probability of selecting the sample s (which will be described in the section about sampling). The reasoning for dividing by $p(s)$ is similar to that in IS. The $(t_1 - t_0)$ results from the integration over the voxel.

The narrowing phase is a little more complicated. Our goal is to weight the accumulated contributions by how likely is the sample generated from voxel intersected along the ray (in the initial guess phase, the contributions were roughly the same along the ray). In this way, we shift the greater values closer to the focal points. We will mark the mentioned weight of the voxel v as w_v. If we have our sample point s and a starting point of the ray x, we will define the normalized ray direction as ω. Our contribution to the accumulator α_v now looks like this:

$$w_v(x, \omega)\frac{l(s)}{p(s)}$$

Where $l(s)$ and $p(s)$ are the same as previously in the initial guess and the $w_v(\omega)$ is:

$$w_v(x, \omega) = \frac{\alpha_v p_v(x, \omega)}{\sum\limits_{v' \in I(V)} \alpha_{v'} p_{v'}(x, \omega)}$$

Where $I(V)$ are all the voxels intersected along the ray, α_v and all the $\alpha_{v'}$ are selection probabilities from the previous iteration, $p_v(x, \omega)$ is a probability of randomly selecting a voxel along the ray, which is equal to:

$$p_v(x, \omega) = \frac{t_1^3 - t_0^3}{3|V_v|}$$

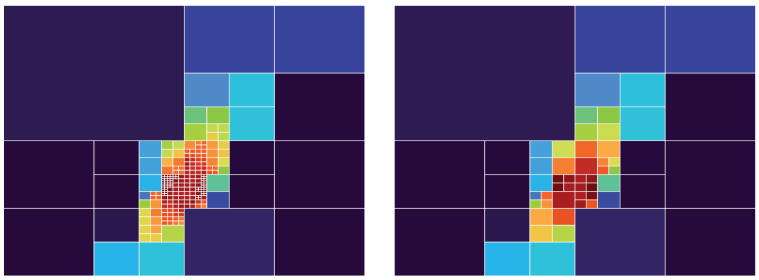
Where t_0 and t_1 are again the entry and exit point distances from the ray start x and $|V_v|$ is volume of the voxel v.

2.6.2 Tree Adaptation

There are two operations that are performed with the tree: splitting and pruning. The splitting operation is performed after each density estimation iteration and the pruning operation is performed only after the last iteration of the algorithm.

In the splitting phase, a node is split if the selection probability α_v is greater than some threshold γ. In this way, we give more precision to the areas potentially containing focal points. In Rath et al. 2023 [1] they state that the value $\gamma = 10^{-1}$ works well for their scenes.

We also perform the pruning phase because the variance between child nodes of some parent node may not be large enough to justify the cost of the traversal. The decision whether to collapse a node is simple. We collapse the node if the maximum leaf node density p_{max} in the subtree at the node is less than two times the average density p_{avg} in the subtree.



(a) Before pruning **(b) After pruning**

Figure 2.8: Visualization of the octree pruning, taken from Rath et al. 2023 [1].

2.6.3 Sampling

Sampling of the ray direction is done in a similar way as sampling from directional quadtree in path guiding from Müller et al. 2017 [8]. In our case we randomly traverse the octree, with probability of selecting a child node ch from some parent node p equal to a_{ch}/a_p. In this way, a leaf node l is selected with probability $\alpha_l = a_l/a_r$ (where r is the root node). If we reach a leaf node l, then we randomly select a point $y \in V_l$ from inside of it's volume V_l with uniform probability. Finally, we create our sample direction ω by normalizing the vector from our current surface point x to the generated point s.

To compute the probability of selecting the sample $p(s)$, we need to accumulate the selection probabilities of all nodes that could generate the same direction:

$$p(s) = \sum_{v \in I(V)} \alpha_v p_v(x, \omega)$$

Where, similarly as in section 2.6.1, $I(V)$ are all the voxels intersected along the ray, α_v are the selection probabilities, $p_v(x, \omega)$ is a probability of randomly selecting a voxel along the ray.

Chapter 3

Implementation

The implementation was done with the Falcor rendering framework [2]. The Focal Path Guiding method [1] is implemented as a collection of several custom render passes integrated within the framework. These render passes are then put together in a render graph with a script. When we run this script with some input scene and configuration, we will get our rendered image.

In this chapter we will first start with a small introduction into the Falcor framework. Then we will describe the main parts of the implementation from top-down approach. First, we will look at the render graph and introduce the used render passes and describe how they are connected. Then we will describe how the main data structure of this algorithm - focal density octree is implemented in memory and how the traversal algorithm works. Then we will describe each of the custom render passes. Starting with focal density estimation passes which maintain the focal density octree - accumulation, splitting and pruning. Then we will finally look at the focal path guiding pass, which uses the focal density octree to sample guided rays and outputs estimated radiance for the final image. In the end we will look at passes used for visualization.

3.1 Falcor Framework

Falcor is a real-time rendering framework from NVIDIA, written in C++. It's use is targeted mainly for research and prototype projects. Supports DirectX 12 and Vulkan graphics APIs, Python scripting, DXR raytracing API from NVIDIA and many other features. It provides basic rendering functionality which abstracts the lower level graphics APIs, scene loading, window creation and simple immediate mode GUI.

The intended way to use the framework is by writing custom render passes. Render pass is a rendering unit that has some inputs and outputs and can be connected to other render passes in a render graph. The render graph is then the final product that renders our desired scene into some image. The inputs and outputs of the render passes are images that represent some intermediate results. The scene and global settings can be accessed by all render passes.

To give an example, we could create a simple render graph from 3 passes: path tracer pass, accumulate pass and tone mapper pass. The path tracer pass traces the scene with 1 sample for each pixel and outputs the resulting radiancies. Then the accumulate pass accumulates the path tracer outputs to a local storage and outputs

their average. Finally, the tone mapper pass maps the high dynamic range values from the accumulate pass to values that could be displayed on the monitor. You can see this example visualized in the Falcor render graph editor in figure 3.1.

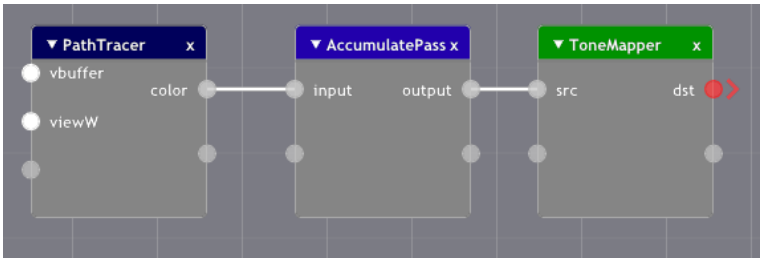


Figure 3.1: Example render graph in Falcor framework.

The render passes are written in C++. Falcor provides many standard render passes that we can use in our render graph together with our custom render passes, such as the mentioned standard path tracer, accumulator and tone mapper passes. The render graph can be created in C++ too by making a separate custom application that calls the Falcor framework. But a simpler way to create a render graph is with a Python script. We can open these scripts in Mogwai application (figure 3.2) which is part of the Falcor framework. It displays the render graph output in real-time and provides simple GUI for tweaking of the render passes and scene settings. You can also create simple scenes with Python scripts and load them with Mogwai.

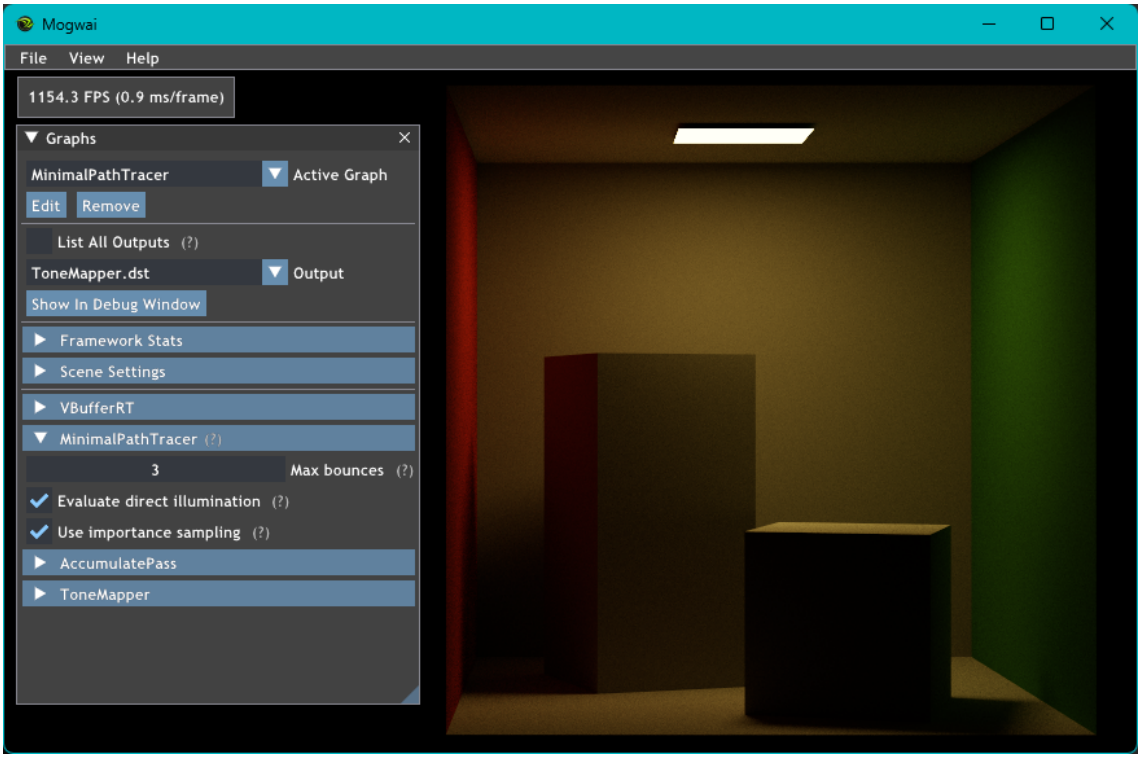


Figure 3.2: Mogwai application from Falcor framework.

An important part of the implementation of render passes are shaders, which are the part of the code that is executed on GPU. Shaders in Falcor are written in Slang programming language. Slang is a fairly new programming language. One of the advantages is multi-platform support - cross-compiles to many other shading

languages, such as HLSL, GLSL, SPIRV and even C++ CPU code. It also brings in many modern language features. The main one is the support for better code modularity with imports. You can simply import some shared functionality from another file which is not supported by default in languages such as GLSL and HLSL. This allows for creation of shader libraries that can be easily shared and used. The Falcor framework has one such library in its codebase and it contains many useful functionalities for rendering, such as several BSDF sampling and evaluation methods, material and scene helper code, random number generators and much more.

Slang supports standard shader types, such as vertex and pixel shaders from rasterization pipeline and compute shaders. But it also supports new raytracing shaders, which allow us to take advantage of the hardware acceleration for this specific task on latest graphics cards. The raytracing shaders are extensively used in implementation of this project because of their performance advantages. We will briefly introduce how they work in the next section.

3.1.1 NVIDIA Raytracing Pipeline

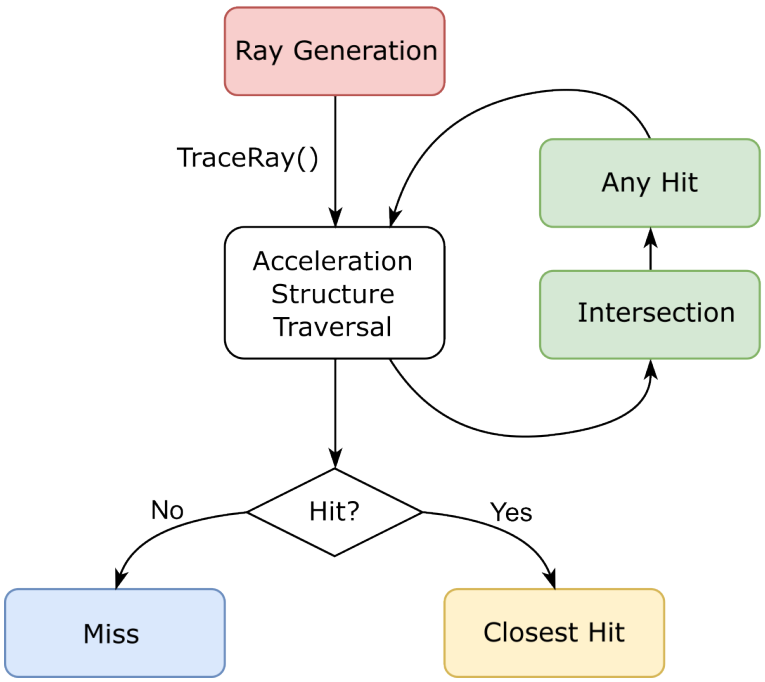


Figure 3.3: NVIDIA Raytracing Pipeline [18].

The raytracing pipeline consists of 5 shaders, 3 mandatory and 2 optional. You can see them in figure 3.3. The 3 mandatory are: ray generation shader, miss shader and closest hit shader. The ray generation shader is the starting point of raytracing. It is called for each pixel and creates initial ray pointing from camera through that specific pixel. It then calls TraceRay function, which shoots the ray into the scene. The ray is then evaluated with acceleration structure traversal, which finds if there are any intersections. If there are no intersections, then the miss shader is called. This shader is typically used to sample from an environment map or simply use some background color. If there are some intersections, then the closest hit shader

is called for intersection closest to the ray origin. This shader is typically used for lighting calculations of opaque geometry.

The 2 remaining optional shaders are any hit shader and intersection shader. The any hit shader is called for each intersection and is typically used for alpha testing for evaluation of non-opaque objects. The intersection shader can be used to implement custom intersections of user-defined geometry.

3.2 Render Graph

The render graph of the focal path guiding implementation consists of 4 custom render passes and 3 standard passes from the Falcor framework. The 4 custom render passes in code are FocalDensities, NodeSpliting, NodePruning and FocalGuiding passes. The 3 standard passes are VBufferRT, AccumulatePass and ToneMapper pass. You can see how they are connected in Falcor render graph editor on figure 3.4. This shows the exact connections how they are in the framework, but it doesn't show how exactly are the passes executed, because some of them are executed only for specific number of iterations. To get a better idea of the execution flow, we will describe it with the diagram in figure 3.5.

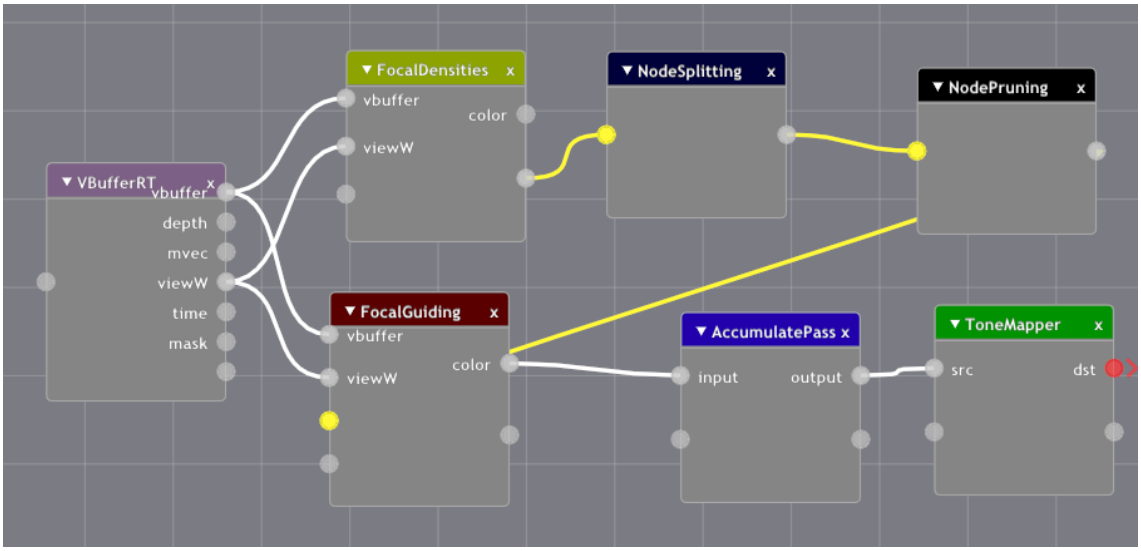


Figure 3.4: Focal path guiding render graph in Falcor framework.

In figure 3.5 the graph is divided into 2 parts: estimation of focal densities and rendering of the final image with focal path guiding. The focal density estimation part builds the focal density octree and is run only at start of the program for fixed amount of iterations equal to $n_{init} + n_{narrow} + 1$. The rendering part then uses this precomputed octree for the guiding and can be run as long as we want, depending on the desired number of samples.

The focal density estimation starts with the initial density accumulation which computes the initial guess as described in section 2.6.1. It runs for n_{init} iterations. Then it switches to the density accumulation with narrowing, which works mostly the same as initial density accumulation, but it additionally applies narrowing weights, similar to those described in section 2.6.1. Both accumulation phases are implemented in the same pass FocalDensities (figure 3.4). The narrowing accumulation runs together with the splitting pass, which deepens the octree. Both the

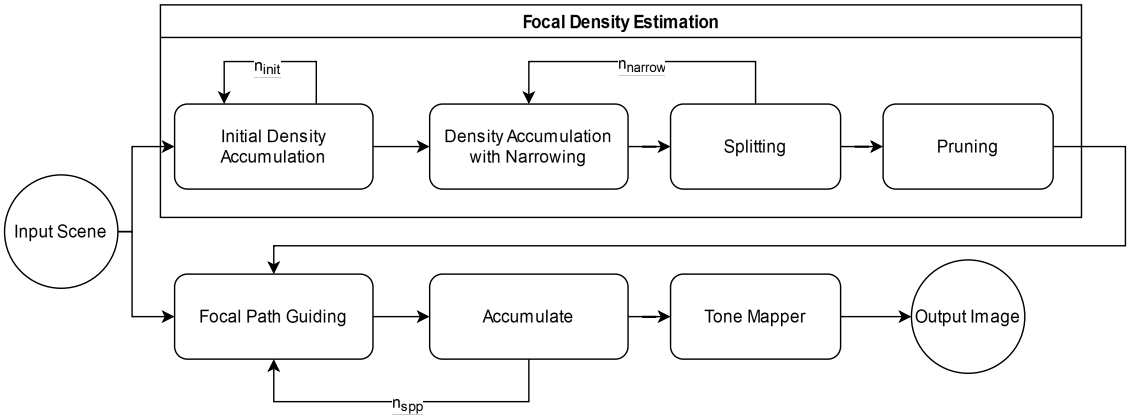


Figure 3.5: Focal path guiding render graph flow diagram.

narrowing and splitting are run for n_{narrow} iterations. At the end of the focal density estimation, the pruning pass is executed to reduce the size of the final tree. Both the splitting and pruning pass use the same criteria for splitting/pruning as described in section 2.6.2.

After the focal density estimation is finished, it begins rendering the image with focal path guiding pass, which uses sampling described in section 2.6.3. Each run of this pass evaluates radiance of one sample per pixel. Then the accumulate pass accumulates the focal path guiding pass outputs to a local storage and outputs their average. Finally, the tone mapper pass maps the high dynamic range values from the accumulate pass to values that could be displayed on the monitor.

3.3 Focal Density Octree

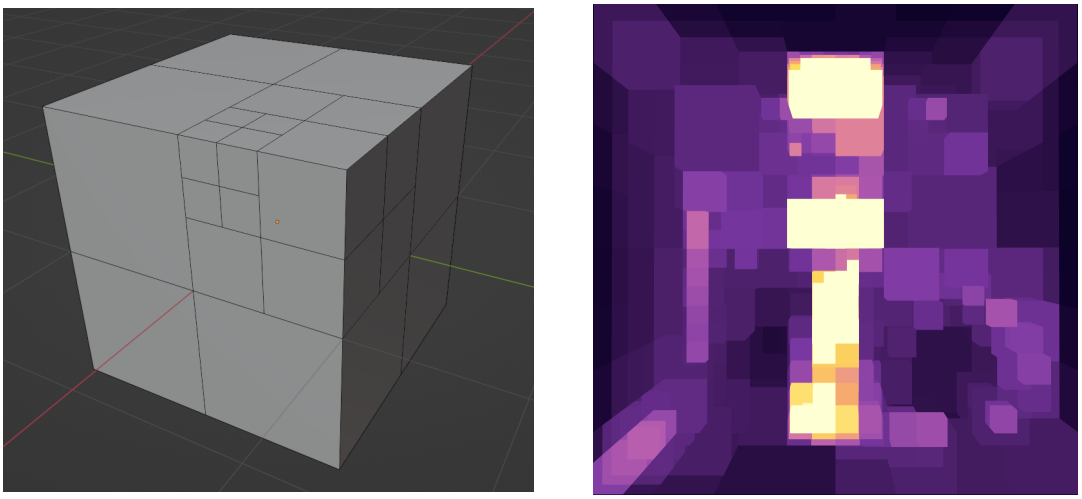


Figure 3.6: Octree data structure. Simple tree node visualization on left. Visualization of stored densities on right.

Similarly to Rath et al. 2023 [1], in this implementation an octree data structure [16] has been chosen to represent the distribution of focal densities in space. You can see a simple visualization of an octree in figure 3.6 on the left. Each node can be visualized as a cube and its child nodes are the 8 cubes that are created from

subdivision of this cube. On the right image of the figure 3.6 you can see an example visualization of focal densities stores inside an octree.

The reason for choosing an octree for this task is that it is simple to represent volumetric data with it, which is our case with the focal densities. We could use kd-tree instead, for example. But that could have some disadvantages. The tree would be deeper and it would be more complicated to maintain the structure. But it could be interesting to implement this method with kd-tree in the future and compare it.

3.3.1 Data Structure

The octree is stored in a buffer on GPU as an array of DensityNode struct shown in figure 3.7. This struct actually does not contain focal density of that specific node, but it contains densities of all 8 of its node children stored in DensityChild struct. The index stored in DensityChild is an index to DensityNode containing children of the DensityChild. If this index is 0 it means that the node is a leaf. The accumulator attribute contains accumulated radiance in this node. When divided by global accumulator (all accumulated radiance) and volume of the node it gives us the focal density for the node. The parentIndex and parentOffsetAndDepth are used to access parent nodes from children nodes, which is needed for the pruning pass.

```
struct DensityChild
{
    uint index;
    float accumulator;
}

struct DensityNode
{
    DensityChild childs[8];
    uint parentIndex;
    uint parentOffsetAndDepth;
}
```

Figure 3.7: Focal density octree node structure.

The size of each DensityNode is $8 \times (4+4)+4+4 = 72$ bytes, which is 9 bytes per octree node, because we store 8 nodes together in one DensityNode. It could be further reduced if we remove the parentIndex and parentOffsetAndDepth attributes, which could be done after the density estimation phase is finished and copy the values to the simplified structure. The size of each node would then effectively be 8 bytes. To minimize the size of the structure is important, because it has a great impact on performance through the cache miss ratio. That is the main reason why the structure is specified in this way. Little disadvantage is that the root node cannot store any data inside, so its accumulator is stored in a separate variable.

3.3.2 Ray Traversal

The main algorithm that uses the focal density octree is traversal of the nodes intersected by some ray. In this implementation it is used for accumulation of focal densities, PDF estimation of sampled ray directions and visualization of the stored densities. The algorithm used in Rath et al. 2023 [1] for octree traversal is from Revelles et al. 2000 [19]. It is one of the most effective algorithms for this task. But for our implementation, it has some disadvantages. The main goal of the algorithm from Revelles et al. 2000 [19] is to find the first intersection of the ray with some geometry. This implies that the algorithm must traverse the nodes in order from the to the ray origin. But in our case it does not matter in which order are the nodes traversed. We can remove a lot of complexity of the algorithm by not caring about the order. Also, the original algorithm is recursive, which is problematic on GPU. This implementation uses while cycle in combination with a stack. But this does not mean that the implemented algorithm is more effective than some GPU-friendly variant of the one from Revelles et al. 2000 [19]. This is something worth investigating in future work.

Algorithm 1 Density Octree Ray Traversal

```
parentStack = {0}
childStack = {0}
stackSize = 1
box = sceneBoundingBox
while nodesStackSize > 0 do
  topIndex = stackSize - 1
  if childStack[topIndex] ≥ 8 then
    stackSize−−
    if stackSize > 0 then
      box = extendBox(box, childStack[topIndex−1]−1)
    end if
  else
    node = getNode(parentStack[topIndex], childStack[topIndex])
    currBox = shrinkBox(box, childStack[topIndex])
    if rayIntersectsAABB(rayOrigin, rayDirection, currBox) then
      if node.isLeaf() then
        evaluateLeaf(node)
      else
        evaluateInnerNode(node)
        parentStack[stackSize] = node.index
        childStack[stackSize] = 0
        box = currBox
        stackSize++
      end if
    end if
    childStack[topIndex]++
  end if
end while
```

Now we will look at the pseudocode of the algorithm 1 used in this implementation. It uses 2 stacks of the same size: parentStack and childStack (in the actual implementation, they are in one stack, but for simplicity here they are split into two). The parentStack contains indices of parents of currently traversed nodes, initially it contains zero index - their root node. The childStack contains child indices (0 to 7) referring to the child nodes of parent node stored in parentStack, also initially zero - first child. Each iteration of the while cycle evaluates one child node and increments the index in childStack. When all 8 children are evaluated, it pops top of the stack. The evaluation of each node fist checks for intersection with its AABB. If intersected and the node is not a leaf, then push this node on the parentStack and in next iteration we will evaluate it's children. This means that the traversal is depth-first. The bounding box used in the traversal can be shrank or extended based simply on the index of the current child.

3.4 Focal Density Accumulation Pass

The focal density accumulation pass accumulates the estimated radiance into the octree. The pass itself does not change the structure of the octree (it does not add or delete the nodes), it only updates the accumulators stored inside the nodes.

Radiance is estimated with path tracing, using the same algorithm as in Focal Path Guiding pass, which will be described later in section 3.7. For each evaluated path for each ray bounce the algorithm from section 3.3.2 is used to traverse the intersected octree nodes and atomically adds to their accumulators the estimated radiance for this specific path.

Because path guiding is used to generate the ray paths, the accumulated results are influenced by estimated densities from previous iterations. Because of this, the octree is double-buffered. In the first buffer we store the octree from previous iteration that is read-only and is used in path guiding. The second buffer is used for accumulation of the newly estimated radiance, initially contains the copy of the first buffer. The buffers are then swapped after each iteration.

Additionally, between each two iterations, all node accumulators are multiplied by configurable decay (default value is 0.5), which is there to prevent the values of growing to high, which would cause numeric problems.

3.4.1 Initial Phase

Before the density accumulation begins, an initial octree with fixed depth is created (default starting depth is 3). The accumulators are initialized with values greater than zero to prevent division by zero when sampling directions from the tree. So the initial tree represents uniform focal density in space. The root node of the octree has the same position and dimensions as bounding box of the rendered scene.

The accumulation of estimated radiance $l(s)$ uses the same weights as described in section 2.6.1:

$$(t_1 - t_0)\frac{l(s)}{p(s)}$$

In figure 3.8 we can see visualization of the resulting densities stored in the octree after the initial accumulation phase is finished. On left is the testing scene used for this example and will be used in multiple next sections. On the right, we can see the visualization. Brighter colors correspond to higher densities, and darker colors correspond to lower densities. You can see that the algorithm found the light source, the bright cube in the top part of the image. Here the light is not analytic, it's just a polygon with emissive material, so it's important to locate it.

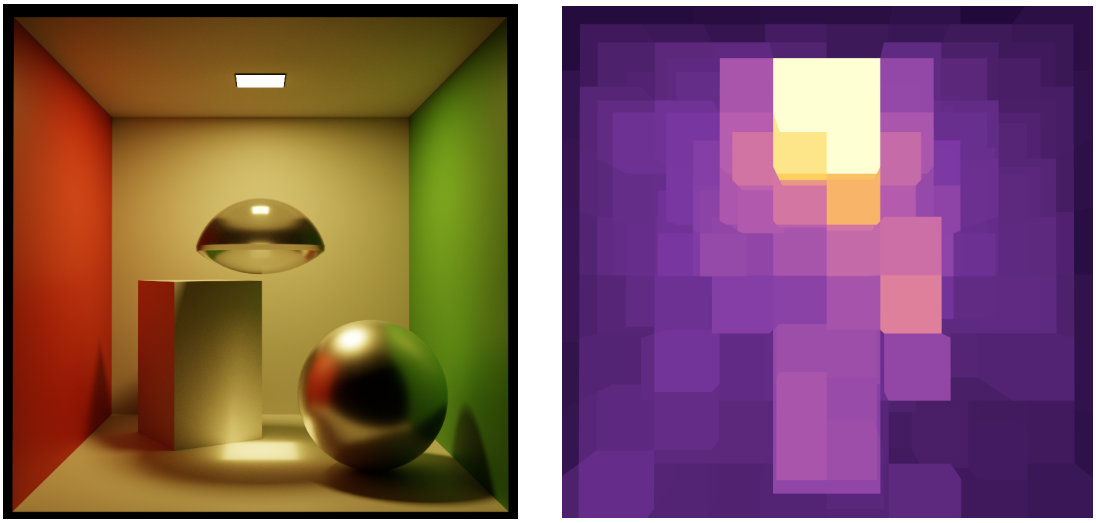


Figure 3.8: Initial phase of density accumulation. On the left is used testing scene. On the right are visualized computed densities (brighter colors correspond to higher densities).

This phase basically approximates the distribution of light in the scene. But it does not necessarily locate all the focal points in the scene, because the radiance passing through can be much lower than the one emitted from the light sources. For example, the light focused from the lens placed in the middle is not that prominent when we look at the visualization of the densities. To give more importance to these regions, the narrowing phase comes in.

3.4.2 Narrowing Phase

The narrowing phase is almost identical to the initial phase. The only difference is in the weights used when accumulating the densities. Again we use the same weights as in Rath et al. 2023 [1] described in section 2.6.1, but with a little tweak, adding narrow factor n:

$$w_v(x,\omega) = \frac{(\alpha_v p_v(x,\omega))^n}{\sum_{v' \in I(V)} (\alpha_{v'} p_{v'}(x,\omega))^n}$$

If we use narrow factor $n \geq 1$, we can tweak the strength of the narrowing. Default value $n = 1$ makes it identical to the original implementation.

In figure 3.9 we can see the impact of the narrowing weights on the stored densities. Some parts, for example inside of the lens and the caustic on the floor, are more prominent here. Contrary, areas close to the wall or inside of the box and sphere are darker. This is still with the same number of nodes and depth. To get more precise focal areas we add in the splitting pass.

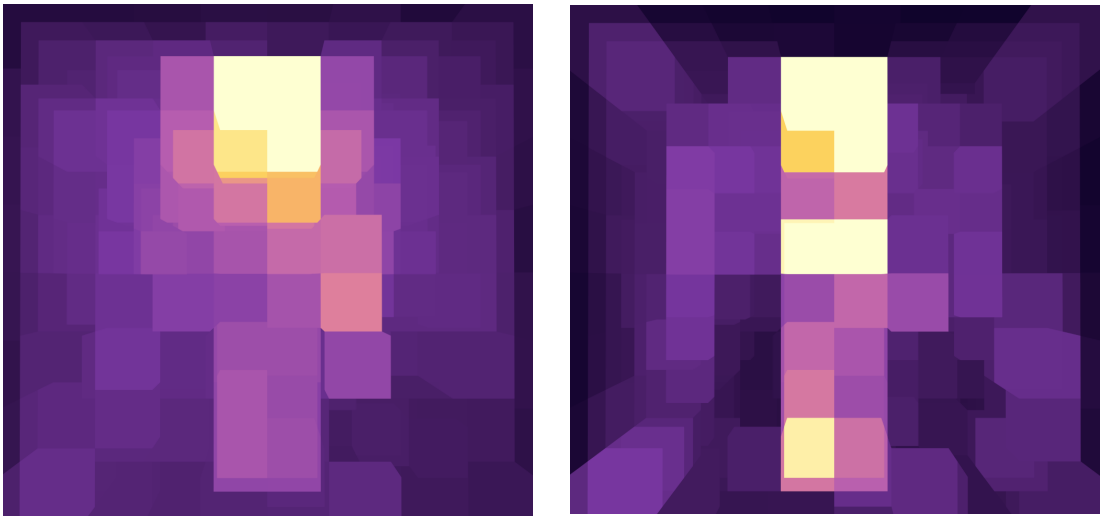


Figure 3.9: Narrowing phase of density accumulation. On the left is result of 2 iterations of the initial phase. On the right is result of 3 more iterations of the narrowing phase (narrowing factor $n = 1$).

3.5 Splitting Pass

The splitting pass deepens the octree based on the stored densities. Each iteration can deepen the structure by 1 level. It is done with a compute shader that is executed on all nodes. If node is a leaf and fulfills the splitting criterion, then 8 child nodes are created. Accumulator value stored in their parent is distributed between them equally. The number of nodes is an atomic variable, by incrementing it we reserve index in the nodes buffer for the newly created children. The buffer has some preallocated free space, so we simply fill it up this way.

The splitting criterion is the same as in Rath et al. 2023 [1]. We split the node if its density times volume α is lower than the splitting threshold (default value is 0.001).

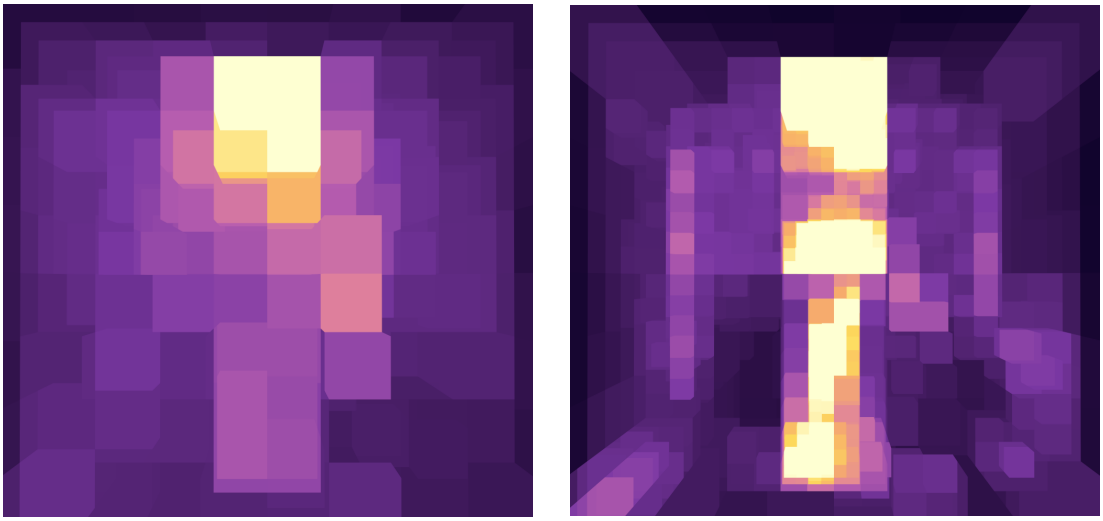


Figure 3.10: Splitting together with narrowing. On the left is result of 2 iterations of the initial phase. On the right is result of 3 more iterations of the narrowing phase together with splitting (narrowing factor $n = 1$).

In figure 3.10 we can see the splitting applied together with narrowing. The focal areas are now even more prominent and their shape is more precise. But it also created a lot of nodes in areas with roughly the same density, which has negative impact on the performance of the traversal - more nodes need to be traversed and it is not always justified by some benefit. The pruning pass helps with this problem.

3.6 Pruning Pass

The pruning pass reduces the size of the octree by removing children of nodes that have a small density variance in their subtree. The heuristic for deciding whether we should prune a node is the following: prune the node if $p_{max} \leq 2p_{avg}$, where p_{max} maximum leaf node density in the subtree and p_{avg} average leaf node density in the subtree.

The implementation is done with a compute shader that is called iteratively for each depth layer, starting from the lowest one. In each iteration, the shader prunes nodes on the current depth level that fulfill the heuristic and computes the p_{max} and p_{avg} densities that are reused in the next iterations. A node is pruned simply by marking it as a leaf, this is done by setting the index in DensityChild structure to 0.

In figure 3.11 we can see the pruning applied after the narrowing and splitting. Many of the unnecessary nodes, for example the ones close to the walls, were pruned. It also pruned many of the nodes with higher density, but the most important parts remained and the representation is more compact now.

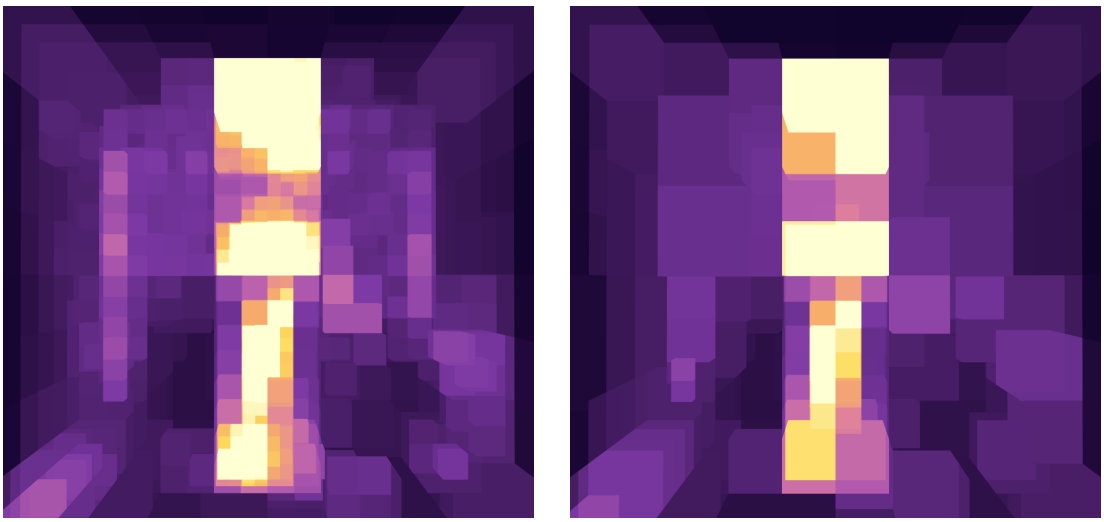


Figure 3.11: Pruning of the focal density octree. On the left is a result of 2 initial accumulation iterations and then 3 iterations of the narrowing phase together with splitting (narrowing factor $n = 1$). On the right is the same execution with a pruning pass added at the end.

3.7 Focal Path Guiding Pass

This is the final pass that renders the image of our scene with path tracing. In figure 3.12 we can see an example scene render with this pass. It is implemented on GPU with shaders from the raytracing pipeline described in section 3.1.1. The ray path is generated with a simple cycle that is limited by configured maximum number of bounces (default is 5). At each iteration, it samples new ray direction and calls the standard TraceRay() function, which searches for intersections of the ray with the geometry by traversing the acceleration data structure handled by the GPU drivers. The path can be terminated early in the cycle either when the ray shot is evaluated as a miss, or when the sampling decides to absorb the incoming ray.

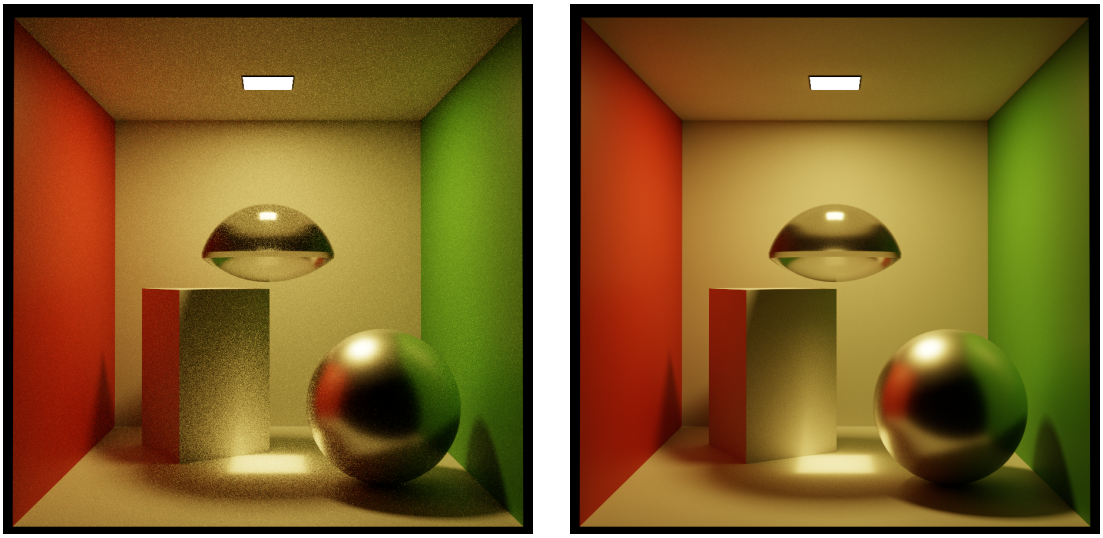


Figure 3.12: Focal path guiding algorithm used on simple Cornell box scene. On the left is render after 40 seconds. On right is render after 20 minutes.

For sampling of ray directions at each bounce we use a combination of BSDF sampling and guided sampling using the focal density octree. If analytical light sources are present in the scene, light source sampling is used additionally. The BSDF sampling and light source sampling is performed with a standard implementation provided by the Falcor framework. The guided sampling uses the procedure from Rath et al. 2023 [1] described in section 2.6.3.

Each bounce is generated either by BSDF sampling or by guided sampling. To decide which sampling method is used at each bounce, we use static selection probability p_g, which is set to 0.5 by default. We generate random number $r \in [0, 1]$ and if $r \leq p_g$, then guided sampling is used, otherwise BSDF sampling is used.

In cases when roughness of the material is close to zero (perfect reflection) or when the material has high specular transmission (perfect transmission), we use only the BSDF sampling. The perfect reflection/transmission BSDF is zero for most directions and only the value of the direction representing the perfect reflection/-transmission is greater than zero (and probably very high). Because of that the guided sampling has near to zero probability of generating the perfect reflection or transmission.

To make the Monte Carlo integration of the estimated radiance unbiased, in cases when we select between both BSDF and guided sampling, we use MIS weights:

$$w_{mis}(s) = \frac{1}{p_g p_{guide}(s) + (1 - p_g) p_{brdf}(s)}$$

Where p_g is probability of selecting guided sampling, p_{guide} is the PDF of the guided samples and p_{brdf} is the PDF of BRDF samples. We multiply our estimated radiance $l(s)$ by this weight.

3.8 Visualization

There are 2 types of visualizations implemented: visualization of densities stored inside the focal density octree and visualization of rays generated by focal guiding. These visualizations are useful for debugging and for analyzing the effectiveness of certain parts of the implementation. They are implemented as additional render passes, that can be optionally connected to the render graph described in section 3.2.

3.8.1 Focal Density Visualization

The focal density visualization is done by shooting rays from the camera into the density octree for each pixel. The traversal algorithm described in section 3.3.2 is used to traverse all intersected nodes. The algorithm then outputs either maximum or average of all the intersected nodes based on the configuration. These values are then mapped to colors. Here we use the Inferno color map, which is perceptually uniform and should therefore be ideal for visualization of the scalar density values. Brighter values correspond to high focal densities and darker values correspond to low focal densities. The used color map can be configured and switched to, for example, Viridis or Plasma. In figure 3.13 we can see visualization of maximum densities. The output is blended with the scene geometry, which can be turned off.

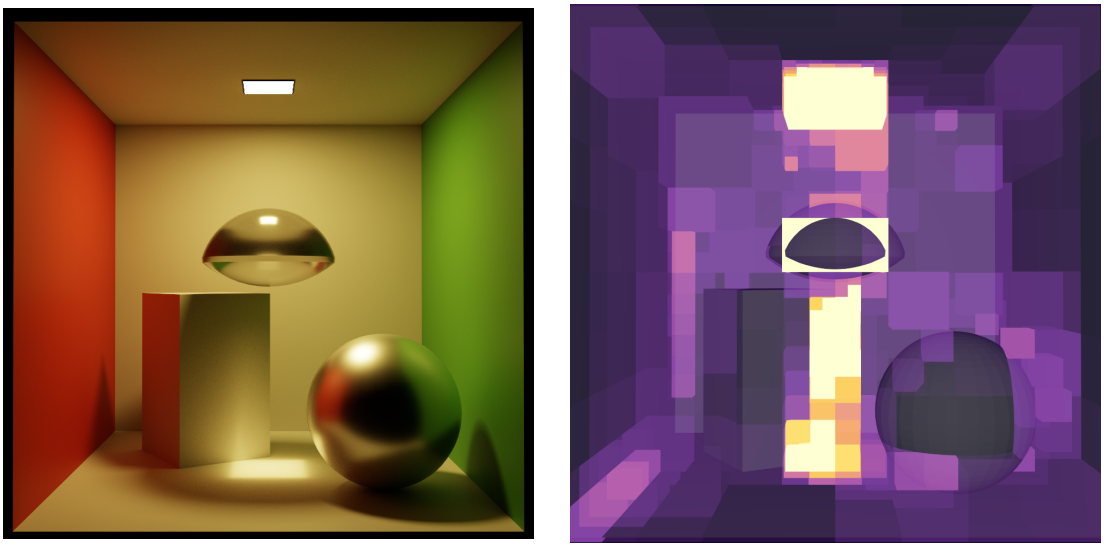


Figure 3.13: Visualization of focal densities. On the left is the render of used scene. On the right is the visualization of maximal densities blended with the scene geometry.

Because the focal densities can be of arbitrary non-negative size, we need to use minimum and maximum thresholds for clamping the density values before we

map them to color. We can see example of different thresholds in figure 3.14. The thresholds are configurable and it's useful to be able to tweak them to get better idea of where the highest values are. In the figure on the right image with higher threshold you can see that the light source has largest focal density, which is not deducible from the left image with lower threshold.

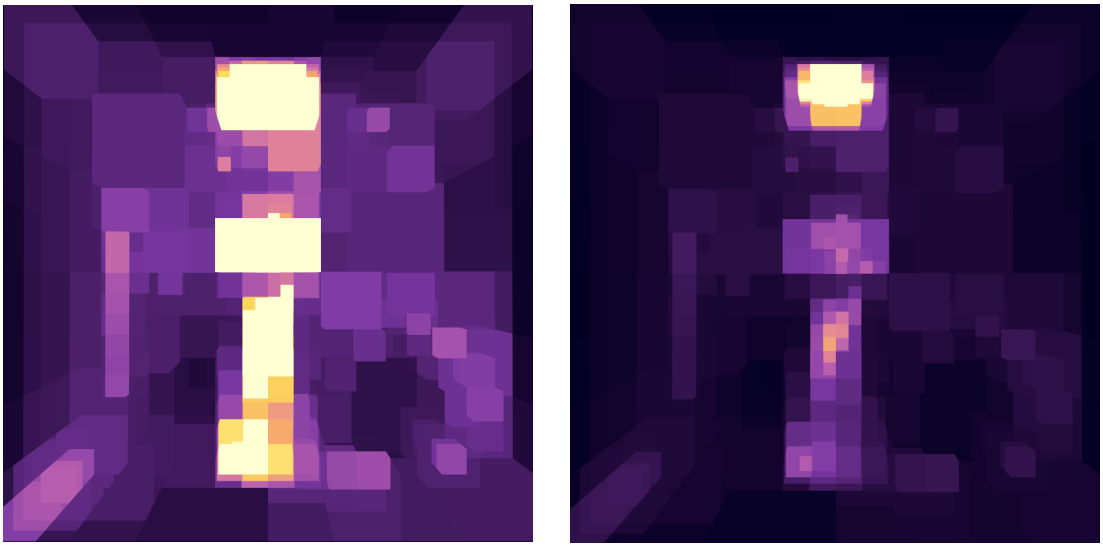


Figure 3.14: Maximum intensity projection visualization of focal densities. Maximum thresholds 1 on the left, 8 on the right.

In the figure 3.15 you can see the same visualization, but the displayed values are average densities along the ray. The average intensity projection can sometimes give us better idea of the distribution of the densities in space. On the other hand, the maximum intensity projection can be better for locating important parts of the scene and also, because it displays sharp edges of the nodes, it can be useful for analyzing structure of the octree.

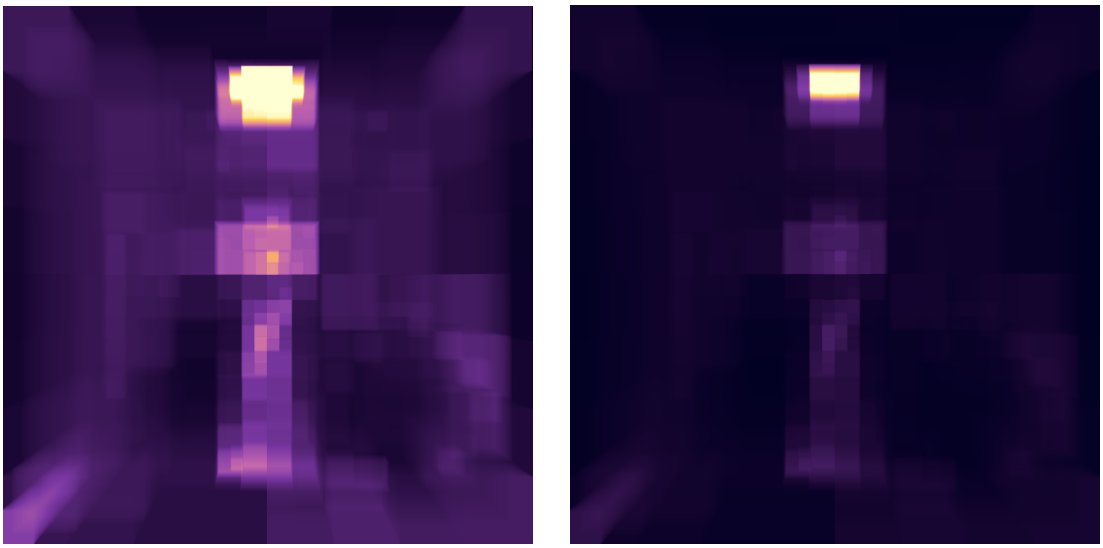


Figure 3.15: Average intensity projection visualization of focal densities. Maximum thresholds 1 on the left, 8 on the right.

3.8.2 Ray Sampling Visualization

We can visualize the rays generated by the focal path guiding pass described in section 3.7. The implementation is similar to the one in focal path guiding pass, only instead of storing the estimated radiance to output image, we store the lines of the evaluated paths to a buffer so that we can render them in rasterization pass. Also, the rays start from the same position, which can be changed by pressing Shift and right mouse button at some position in the scene when using the Mogwai application. In the figure 3.16 we can see visualization of ray paths of length 3.

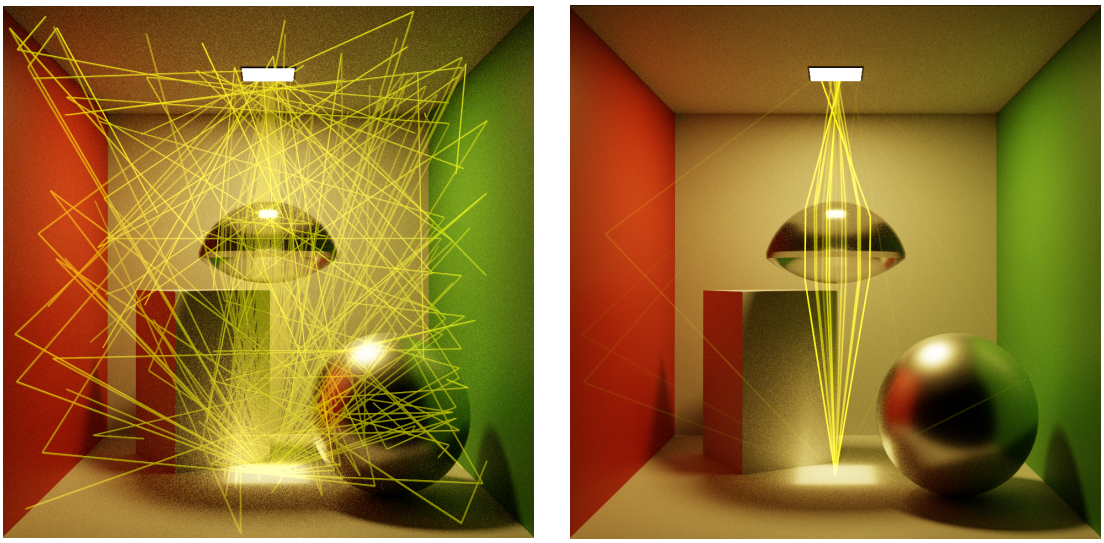


Figure 3.16: Visualization of ray paths (not depth tested). On the left are generated paths from a single position at the middle of the floor. On the right are the same paths, but the intensity of the lines is set proportional to radiance estimated from the path - only paths that hit the light are visible.

We can also visualize PDF at some point. You can see the example in figure 3.17. This can be useful for checking correctness of the guided sampling.

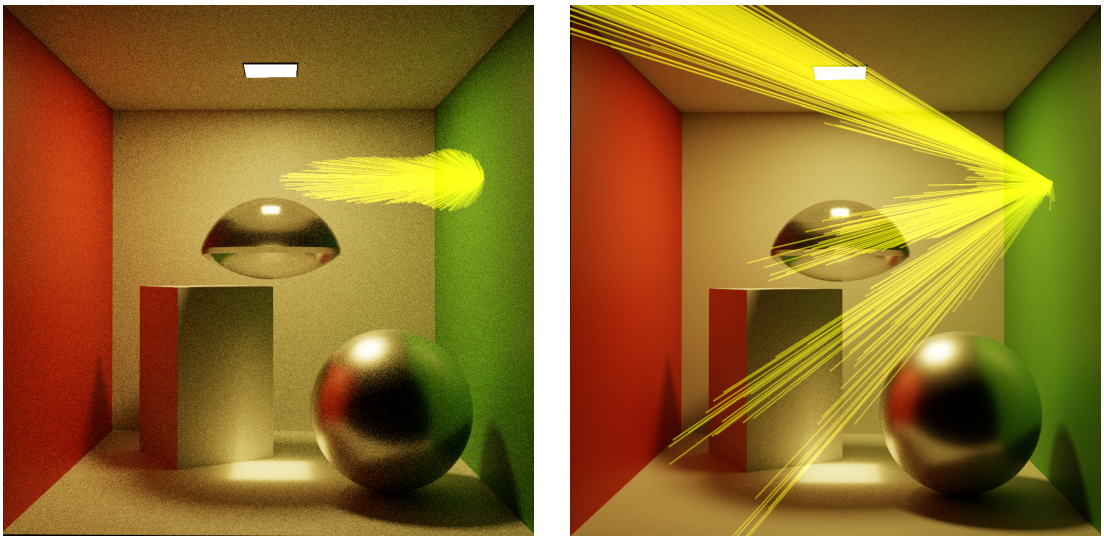


Figure 3.17: Visualization of PDF with current view direction (not depth tested). On the left is BRDF PDF. On the right is guided PDF.

Chapter 4

Results and Evaluation

In this chapter, we will analyze the outputs of the implementation, described in chapter 3, on multiple different scenes. We will examine the quality of the generated octree and compare the rendered results with a standard path tracer implementation. We will start with simple scenes demonstrating different types of focal points. Then we will look at how the implementation handles more complicated scenes (in terms of number of objects and size). In the end, we will summarize the results in the last section.

4.1 Environment

All presented images were rendered on a PC with AMD Ryzen 5 3600 processor and NVIDIA GeForce RTX 3060 graphics card. If not specified otherwise, the focal path guiding render graph has the following parameters: 5 maximum ray bounces, 12 iterations of density estimation, first 6 are the initial estimation passes, the other 6 are narrowing passes in combination with splitting. For comparison with our results, we use classical path tracer implementation from the Falcor framework, MinimalPathTracer. It also uses a maximum of 5 ray bounces by default.

In figures comparing the two methods, we use the following description scheme: method, time, spp, MSE. Where method is the method used to render the image and is either pt or guided: pt is the classical path tracer, guided is the focal path guiding algorithm implemented in this thesis. Time shows the time it took to render the image in seconds or minutes. When comparing 2 renders, we use equal time. Next, spp means samples per pixel - in our case it is the number of generated ray paths for each pixel, which is equal to the number of executions of the path tracing/guiding pass. MSE is the mean squared error of the presented image and a reference image, which was rendered separately for a very high number of samples per pixel, typically taking up to an hour to render.

4.2 Focal Points

In this section we will go through several different types of focal points to evaluate the quality of the generated octree and check if the specific focal point is captured inside.

4.2.1 Direct Light Source

We start with the simplest type of focal point, the light source itself. As a light source, we use an emissive polygon. This means that the light is not analytic. The path tracer does not know the position of the light and cannot sample to it with light source sampling. We do this to demonstrate that the algorithm finds the position of the light source. In the density visualization in figure 4.1, we can see that it indeed found the position of the light source, because the density is much higher there than in the rest of the scene.

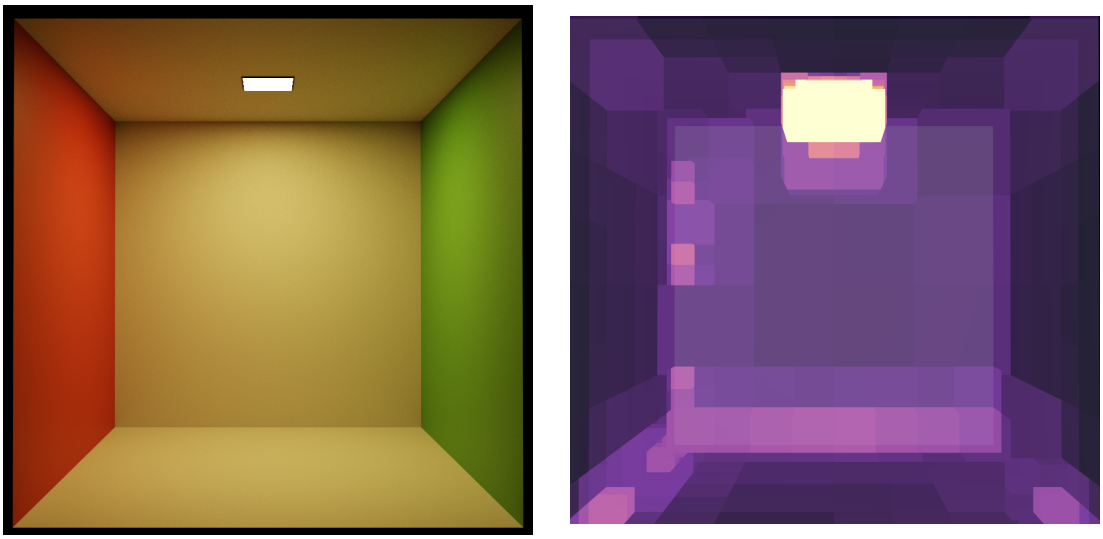
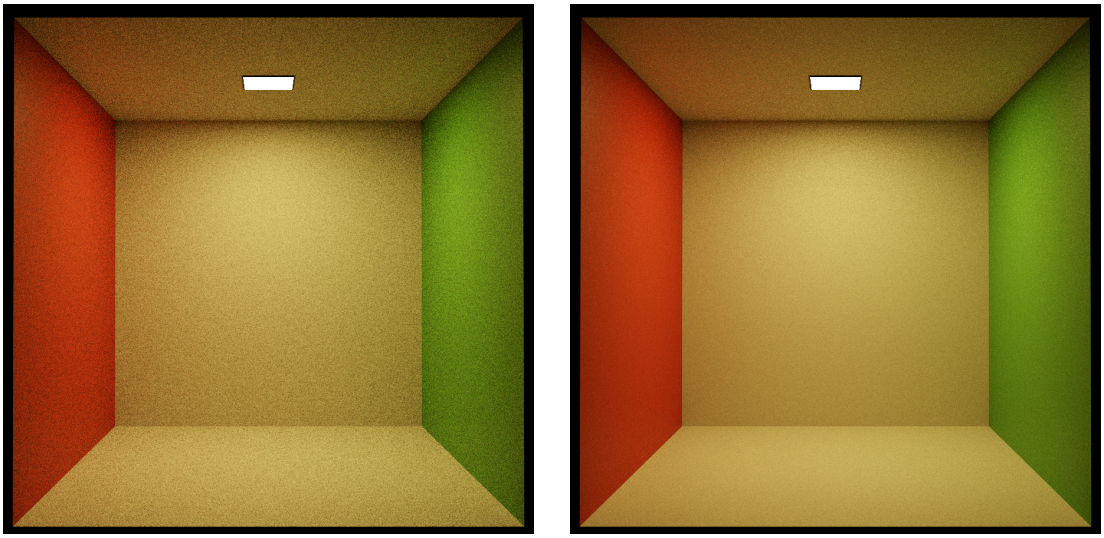


Figure 4.1: Simple Cornell box with just the light source. Reference render on the left. Density visualization on the right.

In the figure 4.2 we can see a render comparison with a classical path tracer. Because the classical path tracer does not know the position of the light, the samples are less effective and result in more noise and higher MSE compared to the focal path guiding.



(a) pt, 20 s, 4800 spp, MSE 2.4e-3 (b) guided, 20 s, 429 spp, MSE 6.5e-4

Figure 4.2: Comparison of classical path tracer (a) and focal path guiding (b) on a simple scene with one emissive light source.

4.2.2 Indirect Light Source

The main light source is made with a hollow cylinder with emissive light source inside simulating a flashlight. The bounced light from the ceiling then creates the indirect light source. In figure 4.3 we can see that both the main emissive light source and the indirect light source are captured in the density octree.

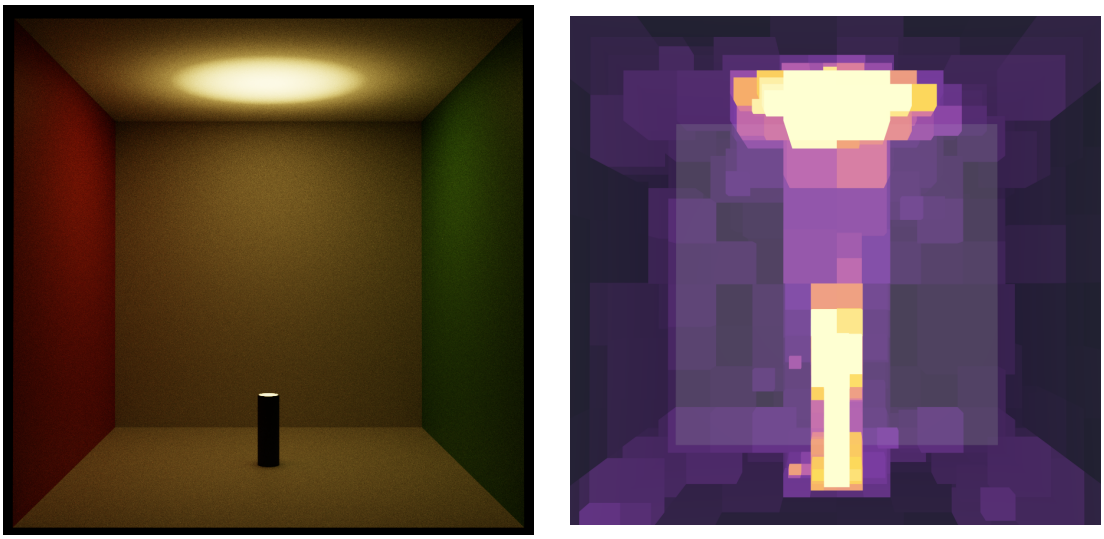
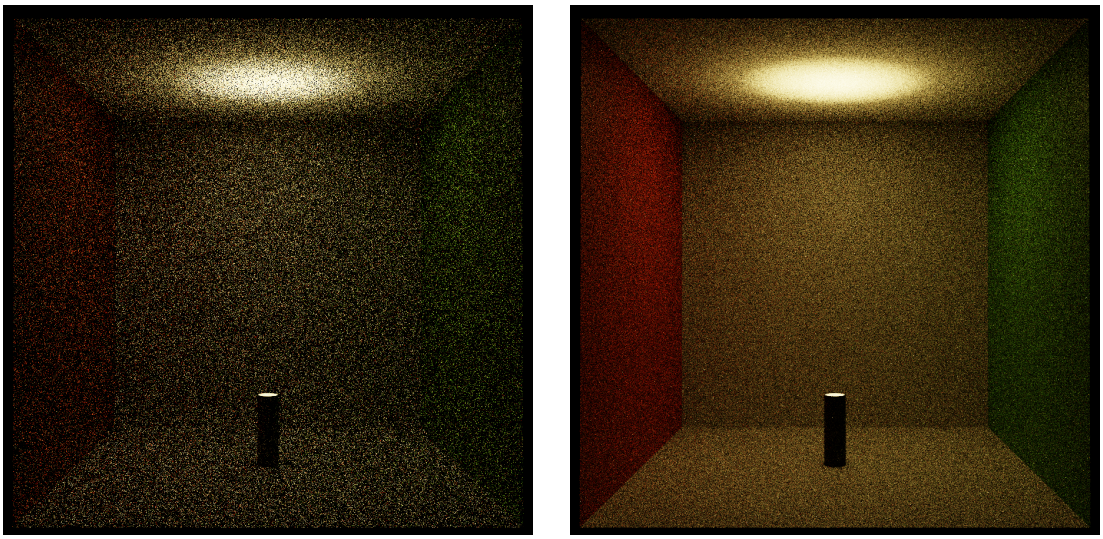


Figure 4.3: Indirect light source. Reference render on the left. Density visualization on the right.

In figure 4.4 we can see the comparison with a classical path tracer. Again, because the classical path tracer cannot use light source sampling, it has more noise and higher MSE.



(a) pt, 20 s, 5154 spp, MSE 4.6e-2 (b) guided, 20 s, 237 spp, MSE 1.1e-2

Figure 4.4: Comparison of classical path tracer (a) and focal path guiding (b) on a scene with indirect light source.

Now we try a similar scenario, but with an analytical spot light source instead of the emissive one. In figure 4.5 we can see that only the indirect light source is captured with the densities. Analytical light samples are not accumulated when building the density octree, because we don't need to detect analytical light sources.

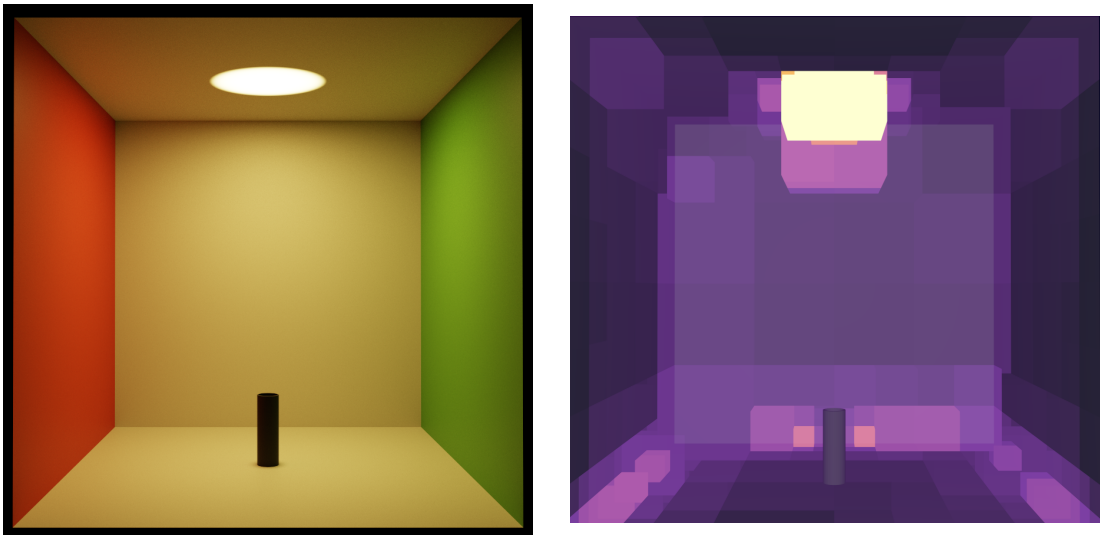
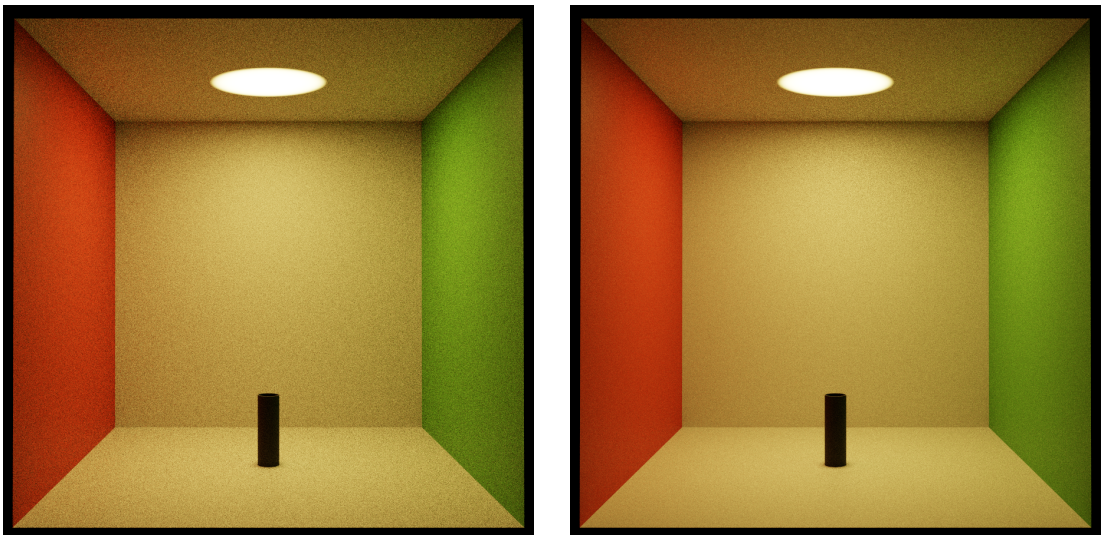


Figure 4.5: Indirect analytical light source. Reference render on the left. Density visualization on the right.

In figure 4.6 we can see the comparison with a classical path tracer. The focal path guiding still has less visible noise and lower MSE. So, in some cases, the focal path guiding method can give better results even with analytical light sources.



(a) pt, 20 s, 2419 spp, MSE 2.0e-3 (b) guided, 20 s, 477 spp, MSE 8.1e-4

Figure 4.6: Comparison of classical path tracer (a) and focal path guiding (b) on a scene with indirect analytical light source.

4.2.3 Lens Focal Point

We use a converging lens placed near the bottom of the scene roughly in distance of its focal length. In figure 4.7 we can see that the focal point of the lens is captured in the density octree.

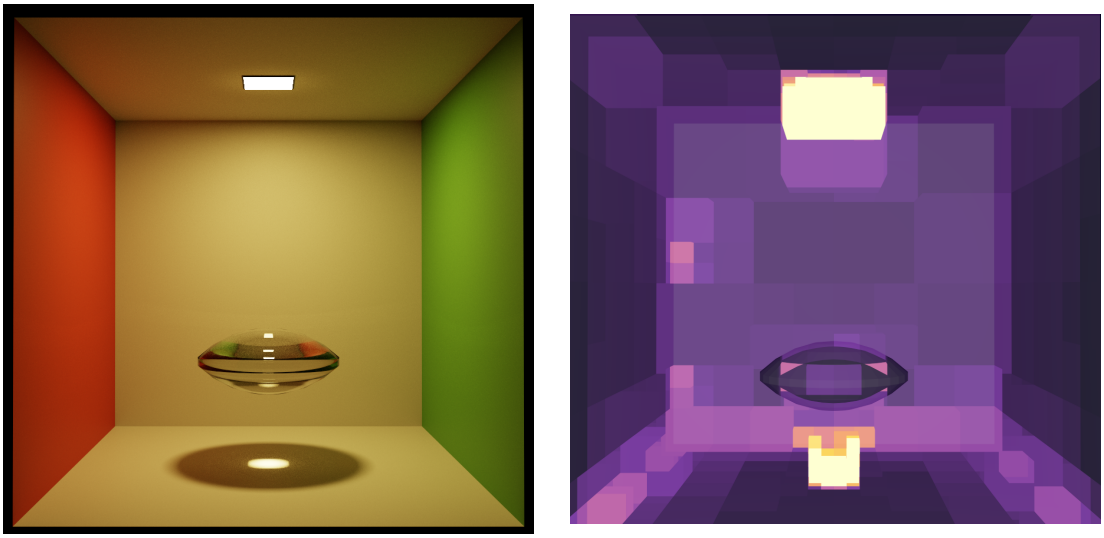
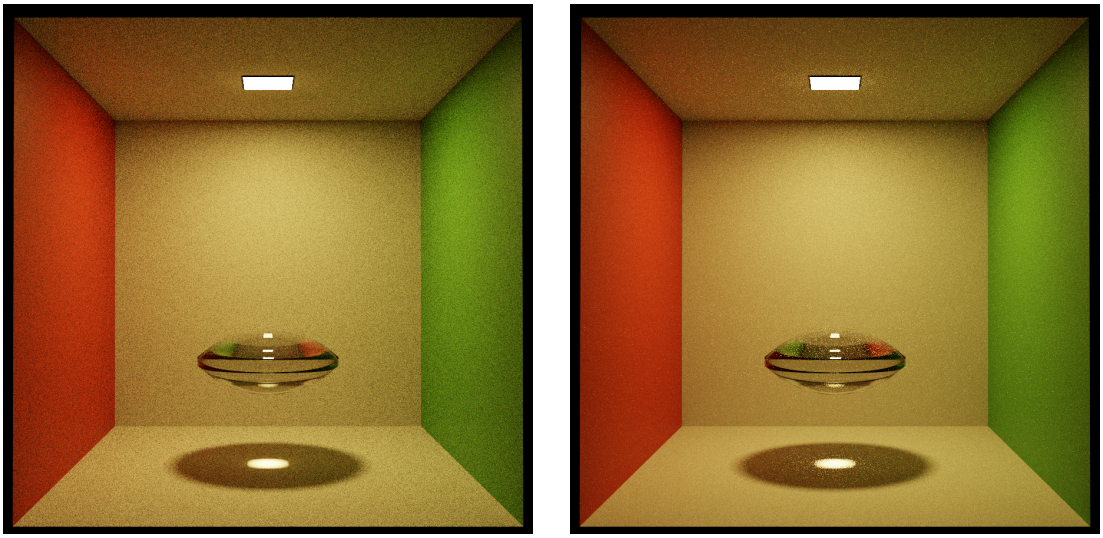


Figure 4.7: Lens focal point. Reference render on the left. Density visualization on the right.

In figure 4.8 we can see the comparison with a classical path tracer. Again, the focal path guiding method appears to be better in terms of overall noise and has a lower MSE. But if we examine closer the lens in the focal path guiding render,

it appears to have more noise on the surface of the lens and in the shadow below compared to the classical path tracer. This could be caused by a combination of perfect refraction of the lens with guided rays being shot more often in the direction of the light source through the lens.



(a) pt, 20 s, 4580 spp, MSE 2.7e-3 (b) guided, 20 s, 370 spp, MSE 1.3e-3

Figure 4.8: Comparison of classical path tracer (a) and focal path guiding (b) on a scene with lens focal point.

4.2.4 Camera Obscura

One of the most impressive result of the original implementation from Rath et al. 2023 [1] is the ability to efficiently render the camera obscura scene. We try to replicate this with a custom scene shown in figure 4.9. In the left room there is a bunny model that we want to project on the barrier located in the right room. The pinhole is located in the middle of the wall that separates the two rooms. The bunny is illuminated with an analytical spot light.

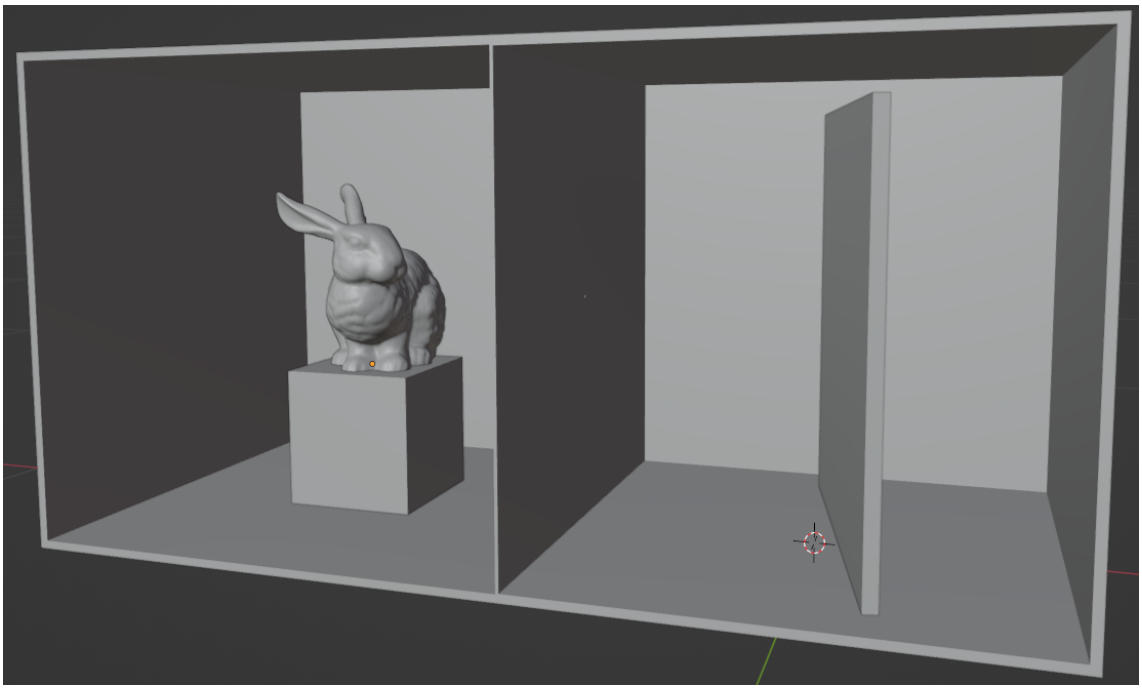


Figure 4.9: Camera obscura scene created in Blender, the pinhole is located in the middle of the wall separating the two rooms.

For purposes of this scene, the render graph settings were changed. It uses 15 iterations of density estimation, first 5 are the initial estimation passes, the other 10 are narrowing passes in combination with splitting. This means that the octree can have an increased depth (each splitting pass can deepen the tree only by one depth level). Due to this, we can accurately represent the focal density of the pinhole.

The render of the first few samples of the focal path guiding method can be seen in figure 4.10. The low number of samples is used intentionally, because the spot light has very high intensity. If we would let it integrate further, the left part would be so bright that we could either set the exposure to make the bunny on the left visible, or set it so that the projection on the right is visible. Figure 4.11 shows visualization of the computed densities. The algorithm successfully found the pinhole.

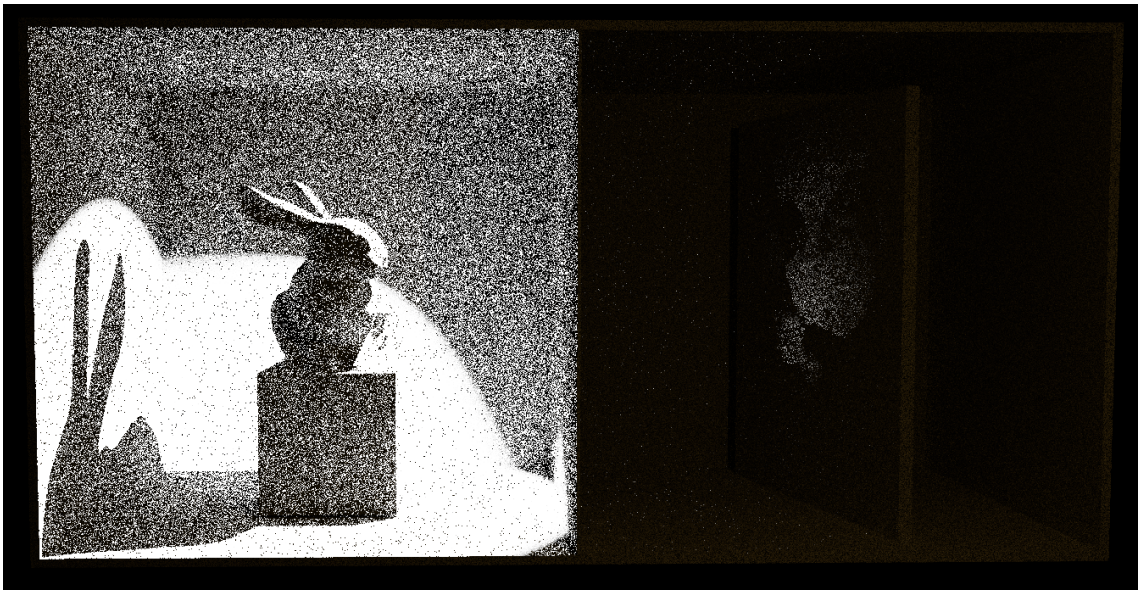


Figure 4.10: First few samples of the camera obscura render.

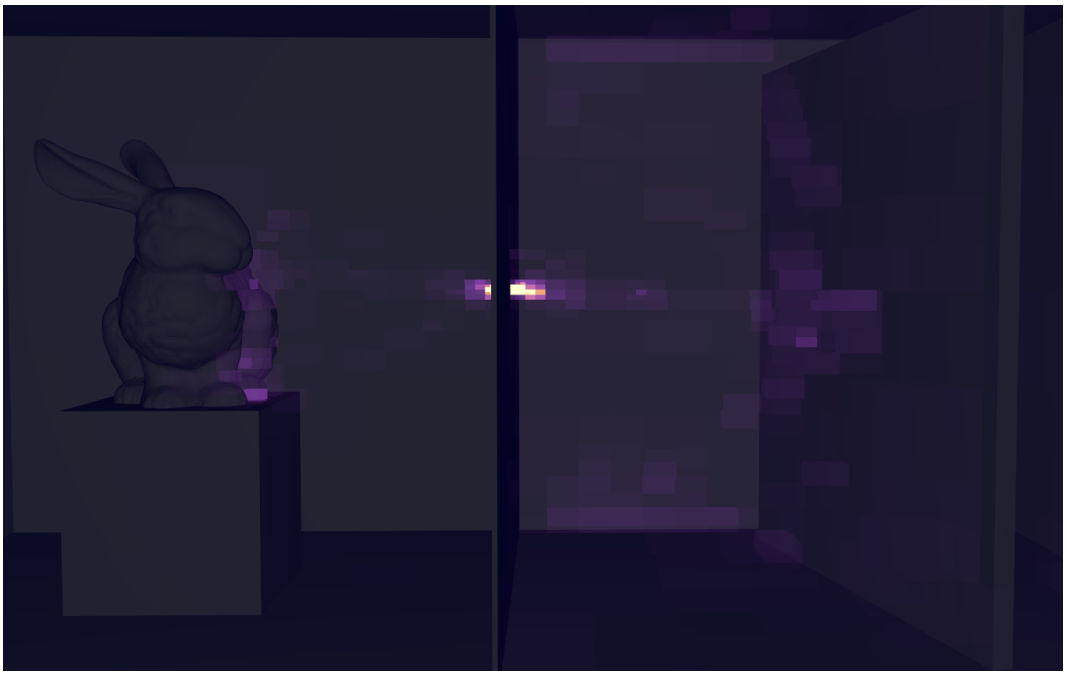
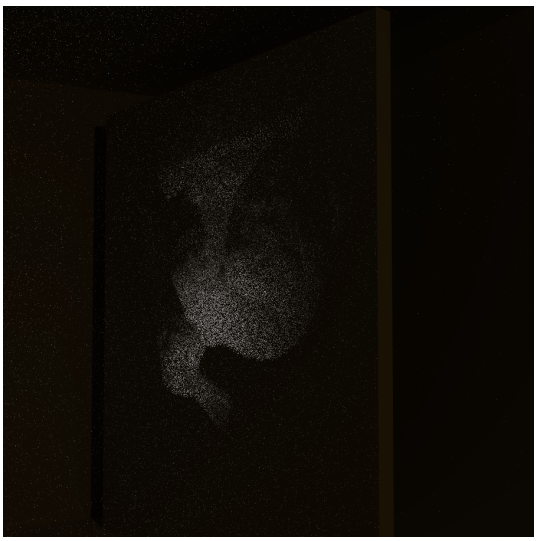


Figure 4.11: Density visualization of the camera obscura scene.

Figure 4.12 contains comparisons with the classical path tracer of the projected bunny image render. In the result of the classical path tracer after 20 seconds of rendering the projection is not much visible, only a few samples managed to pass through the pinhole. Whereas in the 20 seconds focal path guiding render the bunny is clearly visible and the result is even much better than 10 minute classical path tracer render.

(a) pt, 20 s, 1246 spp, MSE 2.5e-3

(b) guided, 20 s, 234 spp, MSE 2.4e-4



(c) pt, 10 m, 71272 spp, MSE 2.0e-3

(d) guided, 10 m, 10235 spp, MSE 7.9e-6

Figure 4.12: Comparison of classical path tracer (a), (c) and focal path guiding (b), (d) on the camera obscura scene.

4.3 Complex Scenes

Now we will test the implementation on more complicated and realistic scenes. We use the Bistro scene from ORCA library [20] and the Dining Room scene from Benedikt Bitterli [21].

4.3.1 Bistro

For this scene, as seen in figure 4.13, the focal path guiding performs much worse compared to the classical path tracer. The reason why is that the scene has analytical light sources and most of the scene has diffuse materials. The computed densities (figure 4.14) are uniformly distributed around the camera view. Therefore, the focal path guiding method does not provide any advantage and the additional overhead causes worse results.

(a) pt, 2 m, 5298 spp, MSE 6.3e-3 (b) guided, 2 m, 924 spp, MSE 1.1e-2

Figure 4.13: Comparison of classical path tracer (a) and focal path guiding (b) on the bistro scene.

Figure 4.14: Bistro scene density visualization. Viewed from outside.

4.3.2 Dining Room

Quite similar to the Bistro scene, the focal path guiding method performs worse than the classical path tracer (figure 4.15). Again, the density octree distributes its values uniformly around illuminated parts of the scene and does not provide much benefit to the quality of the generated samples (figure 4.16).

(a) pt, 2 m, 8324 spp, MSE 3.9e-5 (b) guided, 2 m, 1038 spp, MSE 3.8e-4

Figure 4.15: Comparison of classical path tracer (a) and focal path guiding (b) on the dining room scene.

Figure 4.16: Dining room scene density visualization.

4.4 Summary

From the analysis of scene renders, it can be concluded that the implemented method is beneficial in cases where the scene lighting consists mainly of emissive non-analytic light sources. In such cases, the method finds the locations of these light sources and works as a substitute for light source sampling, which would normally be used when analytical light sources are present.

Other cases where this method is beneficial are when lighting is focused in small area by refraction (lens), reflection (spot light, mirrors) or small narrow gaps (camera obscura). The method successfully locates these focal points and guides the generated paths through them to increase the chance of hitting the original source of light.

However, the method performs poorly for scenes that contain mainly diffuse materials and analytical light sources and the distribution of the light in the scene is not greatly influenced by the mentioned types of focal points. In such cases, the distribution represented by the focal density octree is mostly uniform. Therefore, the sampling does not provide any benefit, because it is equivalent to uniform sampling.

		Path Tracer			Focal Guiding			
Scene	time (s)	spp	FPS	MSE	spp	FPS	MSE	spp ratio
Direct light	20	4800	240	2.4e-3	429	21	6.5e-4	0.09
Indirect e. l.	20	5154	258	4.6e-2	237	12	1.1e-2	0.05
Indirect a. l.	20	2419	121	2.0e-3	477	24	8.1e-4	0.20
Lens Focal p.	20	4580	229	2.7e-3	370	19	1.3e-3	0.08
Camera obscura	20	1246	62	2.5e-3	234	12	2.4e-4	0.19
	600	71272	119	2.0e-3	10235	17	7.9e-6	0.14
Bistro	600	5298	9	6.3e-3	924	1.5	1.1e-2	0.17
Dining room	600	8324	14	3.9e-5	1038	1.7	3.8e-4	0.12
							avg	0.13

Table 4.1: Summary of the comparisons of classical path tracer and implemented focal path guiding method on different scenes. FPS are frames per second when run in the Mogwai application, which is equal to the number of path tracer passes run per second and can be computed from $\frac{spp}{time}$. The spp ratio is the spp of focal path guiding method divided by spp of the classical path tracer.

In table 4.1 are summarized measurements from the presented scene renders. When we look at the spp ratio, the average is 0.13, which means that the focal path guiding method generates roughly $\frac{1}{8}$ of the samples generated by the classical path tracer in the same time. The overhead of the implementation is significant. For comparison, the CPU implementation from Rath et al. 2023 [1] generates roughly $\frac{1}{2}$ of the samples generated by the classical CPU path tracer in the same time. So, there is probably a lot of room for improvement. On the other hand we are more limited by the GPU environment. Possible improvements could be made by trying out some GPU-friendly variant of the octree traversal algorithm from Revelles et al. 2000 [19].

Chapter 5

Conclusion

In the beginning of this thesis we described the problematic of ray tracing, starting with the physical background: BRDF [3] and the rendering equation [4]. Then we continued with several Monte Carlo integration [10] methods used for estimation of resulting radiance from our generated samples. Then we introduced path tracing [4] and described several methods that are its extensions. Starting with ReSTIR [6] and ReSTIR GI [7], which use clever sampling with weighted reservoirs implementing the RIS [5] method. Then we end the chapter with path-guiding methods. Starting with Practical Path Guiding for Efficient Light-Transport Simulation [8], which uses combination of spatial k-d tree and directional quadtree to approximate distribution of radiance in the scene. And finally, we described the Focal Path Guiding for Light Transport Simulation [1] method, which is the main focus of this thesis.

In the implementation chapter, we first gave a quick introduction to the Falcor framework [2]. Then we show the render graph of the implemented render passes, which take advantage of the GPU hardware acceleration. We described the implementation of the octree data structure that stores focal densities. We went through GPU implementation of each render pass, starting with passes for density estimation building the density octree, then the focal path guiding pass, which guides the ray paths according to the density octree and outputs the radiance estimates, which are then used in the final image. In the end, we described the visualization implementation, which is used in many parts of the thesis to get a better idea of values stored in the density octree.

The main goal of reimplementing the focal path guiding method in GPU shaders with Falcor framework was achieved. The implementation gives correct results: when compared with verified standard path tracer, the results are converging together, therefore it is unbiased.

One thing from the original implementation is still missing: handling of diverging focal points. This could be easily implemented in the future by adding a second density octree, with the difference that the rays we would use for PDF computations and density accumulation would be opposite to the ones we use in the original octree. Guiding with this octree would help, for example, with sampling to focal points that are reflected in a mirror. This creates so-called virtual focal point behind the mirror.

From the result renders, we concluded that the method is beneficial for scenes containing non-analytic emissive light sources. The method locates these light sources and can effectively sample in their direction. We also confirmed that it handles well different types of focal points such as lens focal points, intense indirect light sources

and light passing through narrow regions. The most impressive of them is the last one, demonstrated in the camera obscura scene, which is very ineffective to render with classical path tracer.

The method performs poorly in comparison with classical path tracer for scenes that use mainly diffuse materials and analytical light sources. The density octree will be mostly uniform in such cases and the advantages of guided sampling will be lost.

There is a lot of room for improvement. The traversal of the density octree could be more optimized, either by trying out a GPU adapted variant of the algorithm from Revelles et al. 2000 [19], or by considering to switch the data structure itself altogether with, for example, a k-d tree.

Bibliography

1. RATH, Alexander; YAZICI, Ömercan; SLUSALLEK, Philipp. Focal Path Guiding for Light Transport Simulation. In: New York, NY, USA: Association for Computing Machinery, 2023. ISBN 9798400701597. Available from DOI: 10.1145/3588432.3591543.
2. KALLWEIT, Simon; CLARBERG, Petrik; KOLB, Craig; DAVIDOVIČ, Tom'aš; YAO, Kai-Hwa; FOLEY, Theresa; HE, Yong; WU, Lifan; CHEN, Lucy; AKENINE-MÖLLER, Tomas; WYMAN, Chris; CRASSIN, Cyril; BENTY, Nir. *The Falcor Rendering Framework*. 2022. Available also from: https://github.com/NVIDIAGameWorks/Falcor. https://github.com/NVIDIAGameWorks/Falcor.
3. NICODEMUS, Fred E. Directional Reflectance and Emissivity of an Opaque Surface. *Appl. Opt.* 1965, vol. 4, no. 7, pp. 767–775. Available from DOI: 10.1364/AO.4.000767.
4. KAJIYA, James T. The rendering equation. *SIGGRAPH Comput. Graph.* 1986, vol. 20, no. 4. ISSN 0097-8930. Available from DOI: 10.1145/15886.15902.
5. TALBOT, Justin; CLINE, David; EGBERT, Parris. Importance Resampling for Global Illumination. In: 2005, pp. 139–146. Available from DOI: 10.2312/EGWR/EGSR05/139-146.
6. BITTERLI, Benedikt; WYMAN, Chris; PHARR, Matt; SHIRLEY, Peter; LEFOHN, Aaron; JAROSZ, Wojciech. Spatiotemporal reservoir resampling for real-time ray tracing with dynamic direct lighting. *ACM Transactions on Graphics (Proceedings of SIGGRAPH)*. 2020, vol. 39, no. 4. Available from DOI: 10/gg8xc7.
7. OUYANG, Y.; LIU, S.; KETTUNEN, M.; PHARR, M.; PANTALEONI, J. ReSTIR GI: Path Resampling for Real-Time Path Tracing. *Computer Graphics Forum*. 2021, vol. 40, no. 8, pp. 17–29. Available from DOI: https://doi.org/10.1111/cgf.14378.
8. MÜLLER, Thomas; GROSS, Markus; NOVÁK, Jan. Practical Path Guiding for Efficient Light-Transport Simulation. *Computer Graphics Forum (Proceedings of EGSR)*. 2017, vol. 36, no. 4, pp. 91–100. Available from DOI: 10.1111/cgf.13227.
9. VORBA, Jiří; HANIKA, Johannes; HERHOLZ, Sebastian; MÜLLER, Thomas; KŘIVÁNEK, Jaroslav; KELLER, Alexander. Path Guiding in Production. In: *ACM SIGGRAPH 2019 Courses*. Los Angeles, California: ACM, 2019, 18:1–18:77. SIGGRAPH '19. ISBN 978-1-4503-6307-5. Available from DOI: 10.1145/3305366.3328091.
10. CAFLISCH, Russel E. Monte Carlo and quasi-Monte Carlo methods. *Acta Numerica*. 1998, vol. 7, pp. 1–49. Available from DOI: 10.1017/S0962492900002804.

11. LAFORTUNE, Eric P; WILLEMS, Yves D. *Using the modified phong reflectance model for physically based rendering*. Katholieke Universiteit Leuven. Departement Computerwetenschappen, 1994.

12. PHARR, Matt; HUMPHREYS, Greg. Infinite area light source with importance sampling. *Pysically Based Rendering: From Theory to Implementation, PBRT Plugins from the Authors, http://pbrt. org/plugins/infinitesample. pdf*. 2004.

13. VEACH, Eric; GUIBAS, Leonidas J. Optimally combining sampling techniques for Monte Carlo rendering. In: *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*. 1995, pp. 419–428.

14. BLACK, Paul E. *k-dimensional*. 2004. Available also from: `https://xlinux.nist.gov/dads/HTML/kdimensional.html`.

15. ERICSON, Christer. *Real-time collision detection*. Crc Press, 2004.

16. MEAGHER, Donald. *Octree Encoding: A New Technique for the Representation, Manipulation and Display of Arbitrary 3-D Objects by Computer*. 1980.

17. MCCOOL, Michael D; HARWOOD, Peter K. Probability trees. In: *Graphics Interface*. 1997, vol. 97, pp. 37–46.

18. LEFRANCOIS, Martin-Karl; GAUTRON, Pascal. *DX12 Raytracing tutorial*. 2018. Available also from: `https://developer.nvidia.com/rtx/raytracing/dxr/dx12-raytracing-tutorial-part-2`.

19. REVELLES, Jorge; UREÑA, Carlos; LASTRA, Miguel; LENGUAJES, Dpt; INFORMATICOS, Sistemas; INFORMATICA, E. An Efficient Parametric Algorithm for Octree Traversal. 2000.

20. LUMBERYARD, Amazon. *Amazon Lumberyard Bistro, Open Research Content Archive (ORCA)*. 2017. Available also from: `http://developer.nvidia.com/orca/amazon-lumberyard-bistro`.

21. BITTERLI, Benedikt. *Rendering resources*. 2016. Available also from: `https://benedikt-bitterli.me/resources/`.

Appendix

A Source code repository

Implementation of Focal Path Guiding in a fork of Falcor framework:
`https://github.com/shetr/falcor_focal_guiding.git`

B NVIDIA Falcor framework

`https://github.com/NVIDIAGameWorks/Falcor`

C Blender

Used for the creation of some of the testing scenes.
`https://www.blender.org/`