TU WIEN Informatics

Incremental Updates of Path-Traced Scenes during Editing

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Medieninformatik und Visual Computing

eingereicht von

Pascal Dario Hann
Matrikelnummer 01633018

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Mitwirkung: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc
Nina Semmelrath

Wien, 1. Mai 2021

Pascal Dario Hann	Michael Wimmer

TU WIEN Informatics

Incremental Updates of Path-Traced Scenes during Editing

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Media Informatics and Visual Computing

by

Pascal Dario Hann
Registration Number 01633018

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Univ.Ass. Dipl.-Ing. Dr.techn. Bernhard Kerbl, BSc
Nina Semmelrath

Vienna, 1st May, 2021

Pascal Dario Hann	Michael Wimmer

Erklärung zur Verfassung der Arbeit

Pascal Dario Hann

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Mai 2021

Pascal Dario Hann

Danksagung

Ich möchte mich bei Bernahrd Kerbl und Nina Semmelrath für ihre Mitarbeit an dieser Bachelorarbeit bedanken.

Acknowledgements

I would like to thank Bernhard Kerbl and Nina Semmelrath for working with me on this bachelor thesis project.

Kurzfassung

In dieser Arbeit präsentiere ich einen neuen adaptiven Sampling Algorithums für 3D Bearbeitungssoftware, der von mir und meinen Kolleg/Innen entwickelt wurde. Der Algorithums basiert auf der Idee Wissen darüber wie eine Änderung an einer Szene die unterschiedlichen Teile dieser betrifft, für adaptives Sampling zu verwenden. Wir teilen das Bild in Regionen auf und sortieren diese, nach diesem Wissen, von der am meisten betroffenen Region zu der am wenigsten betroffenen. Diese Reihenfolge verwenden wir um das Renderingbudget in erster Linie auf die stärker betroffenen Regionen zu fokusieren und erst danach auf die weniger Betroffenen. Dies geschieht in einem inkrementellen Rendering Prozess. Wir haben diese Methode entworfen um Path-Tracing zum Rendern des Viewports in 3D Bearbeitungssoftware verwenden zu können, ohne mit den Artefakten und Wartezeiten auf ausreichend qualitative Ergebnisse, die diese Technologie üblicherweise mit sich bringt, kämpfen zu müssen. Unser Ziel mit diesem Algorithmus ist es, Benutzer/Innen von 3D Bearbeitungssoftware einen möglichst ununterbrochenen Arbeitsfluss zu ermöglichen und dabei trotzdem nicht auf hochqualitative Rendering Ergebnisse verzichten zu müssen.

Abstract

In this work I present a novel adaptive sampling algorithm for 3D editing software, developed by me and my colleagues. The algorithm is based on the idea of using knowledge about how a given user interaction affects a scene visually. We split the image into regions and order them, according to that knowledge, from most noticeably affected to least. The rendering budget can then be focused on the more affected regions earlier and on the lesser ones later in an incremental rendering process. Although this concept could probably work with other rendering methods, we designed it to be able to use path-tracing as the viewport renderer in 3D editing software without the typical grain-like noise and waiting times for sufficiently smooth rendered images this technology usually comes with. The goal of this work is to offer users of 3D editing software an as uninterrupted workflow as possible while still being able to see their work in high quality.

Contents

CHAPTER 1

Introduction

3D modelling software is essential for a plethora of use cases. Be it creating virtual representations of the physical world for simulations and visualisations or translating human fantasy into media like movies or even interactive media like video games. 3D modelling software is at the centre of all these endeavours and artists and designers rely on these tools to visualise their work. In order to get a good understanding and feel for what they are trying to achieve as early in the process as possible, it is of very first importance to be able to produce high level visuals quickly. Having to undergo a lengthy rendering process in order to see one's work under the final or at least close to final rendering circumstances hinders the design workflow. This leaves artists the choice to either interrupt their workflow regularly and wait for a computationally heavy high quality render or having to work with an obfuscated version of their creation and guessing as to what the final product might or might not look like.

A majority of those computation-heavy steps can be attributed to radiance computation. Lighting a virtual scene is simultaneously one of the most difficult problems in rendering as well as one of the most rewarding ones. Lighting a scene realistically, i.e., in accordance to the real physical world, goes a long way towards creating an illusion of said world. As with many natural phenomena, imitating lighting is exceptionally complicated and algorithms that do so usually are very computational resource intensive.

One straightforward approach, which has become the industry standard for its simplicity and accuracy, is called ray-tracing. The basic idea of the process is to "follow" the rays of light that are present in a scene. Every time a ray hits a surface the impact of that ray on the illumination of that specific intersection is calculated and the ray is consequently followed in its new reflected direction. This is a simplistic view on the real physical phenomenon imitated by ray-tracing. In a real world scenario an infinite amount of light rays is present and most objects are not perfectly reflecting but rather of diffusing characteristics. This means that when a ray hits one such surface, not a single ray gets reflected into a new direction but an infinite amount of new rays get scattered in

every direction around the intersection. However, even when only using a very limited amount of rays to simulate real world radiance, a relatively close approximation of the real behaviour of light in a scene can be achieved. A specialized form of ray-tracing first presented by James T. Kajiya in his milestone publication about the rendering equation [Kaj86], is called "path-tracing". It follows rays shot from the camera into the scene, only considering one new branch at every hit surface, creating a path from the camera to a light source. Which of the many possible ray-directions to choose at an intersection is decided randomly, making this algorithm a so-called "monte carlo" algorithm. Monte carlo algorithms, named after the grand casino in Monaco's capital, use random sampling to achieve numerical results and are often used as an approximation when a complete evaluation of a problem is impractical. Path-tracing achieves images of exceptional quality. Nonetheless the results are heavily dependent on the number of paths traced and for how many intersections they are followed. For a long time existing hardware was not able to compute a sufficient number to achieve acceptable results in real-time. Real-time computation, however, is necessary for interactability in software like 3D modelling applications. In fact, as mentioned before, the shorter the rendering time, the fewer interruptions a user's workflow has to endure.

Even back when hardware was too limited to use ray-tracing in real time, because of its accuracy and simplicity, it quickly became a standard for settings that didn't rely on real-time computation. Animated movies, for example, where the whole movie can be precomputed, later to be viewed with high quality lighting details, were an early adopter of ray-tracing algorithms like path-tracing. The artists creating models for these movies, however, mostly had to work without knowing how their models would look in the final ray-traced context, as a complete render with the technology was too computationally heavy and hence time-consuming for real-time application.
In the last few years, advances in technology have made ray-tracing-based rendering feasible for real-time applications. Using ray-tracing to render a scene while simultaneously editing it in 3D software to get a good understanding of how the final rendered scene will look like has become possible nowadays. See Figure 1.1 for a comparison of different ways to render a scene in the popular 3D editing software Blender [Com18]. The two upper images show techniques that don't create distortions while rerendering after a change to the scene, namely the "Material Preview" mode and the relatively new "Eevee" rendering engine, developed by the Blender Foundation with speed and interactabillity as its focus. Both these methods, however, lack the image quality of ray-traced scenes and the artist can only get an approximation of what the scene is going to look like in the final render context.
The images in the lower half show path-traced renders created with Blenders Cycles rendering engine and the second with NVIDIA Optix denoising, NVIDIA's state of the art AI accelerated denoiser based on [CKS+17a], in addition to that. The denoiser removes any grain-like noise still present in the path-tracing accumulation in an image-reconstruction step. It should be noted, that the difference between pure path-tracing and path-tracing in addition to denoising becomes lesser the more time for accumulation is allocated until the results should be identical as the denoiser has no noise left to remove.

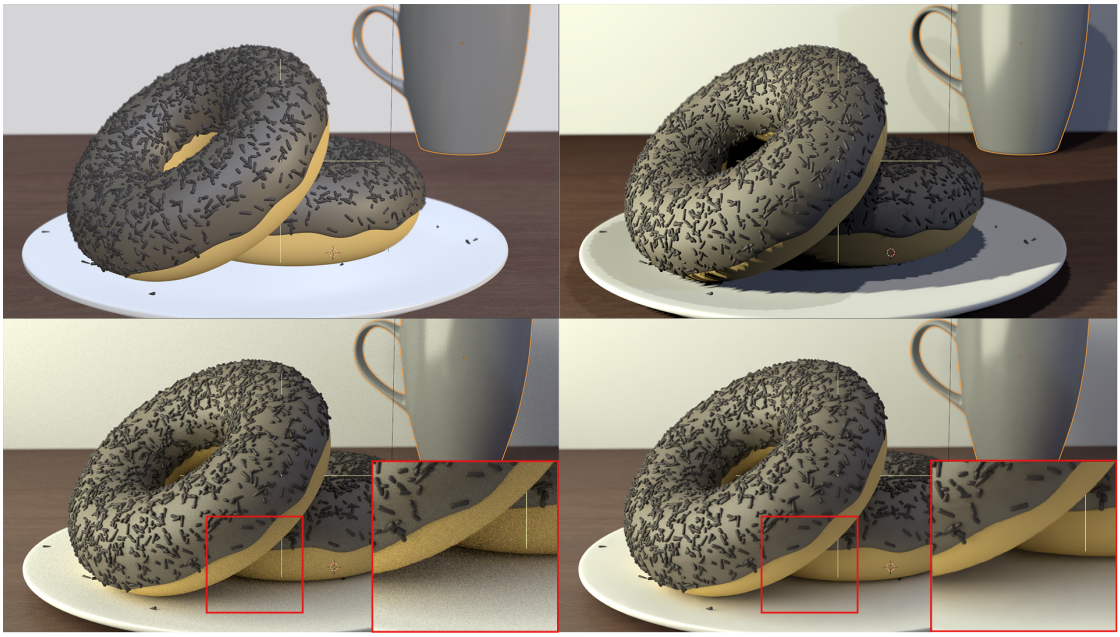

Figure 1.1: A comparison of different options to render one's scene in Blender's viewport. In the upper left, we see Blender's "Material Preview" mode. On the right beside it, the relatively new Eevee renderer developed by the Blender foundation is used. In the lower left corner, path-tracing with the Cycles renderer is applied and in the last image NVIDIA Optix denoising is added on top of that.

This takes a considerable amount of computational resources, however, and the nearly exact same result can be achieved with considerably less effort through denoising.

As can be seen in the comparison images, path-tracing produces results of high quality. The render times, however, while having improved drastically in recent time, are still long enough to interrupt an editing workflow. When rendering a scene with ray-tracing, typically only a limited amount of rays can be computed simultaneously, thus the rendering is not handled as a single action but rather as a continues process of tracing rays and accumulating the samples into a gradually more refined image. The earlier in this process, the more distortion in the form of grain-like noise is present in the image. This noise can be very irritating to the viewer and it takes some time depending on the hardware until the image quality is sufficiently smooth for further work. The usual approach to rerendering with ray-tracing in an interactive editing setting is to discard the accumulation up to the triggering change and to completely rerender the whole image from scratch, indiscriminate to the magnitude of impact that change had on individual parts of the scene.

See Figures 1.2 and 1.3. In these figures one can see how a scene is completely rerendered when moving an object in the background just slightly from its original position. In this example I moved the cup, that can be seen in the background, slightly to the right.

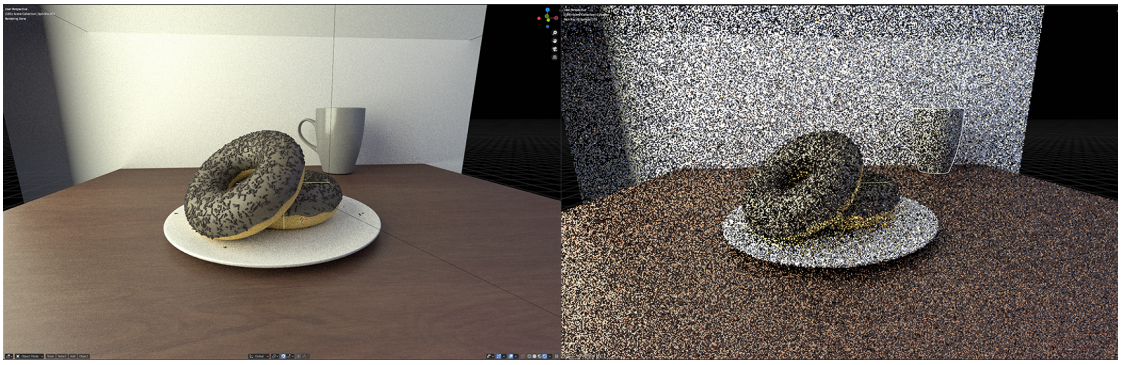

Figure 1.2: Example of how a full rerender is triggered using ray-tracing in the 3D modeling software Blender. The right screenshot was taken a few milliseconds after moving the cup in the background slightly to the right. The whole image gets rerendered from scratch and a significant amount of grain-like noise is present. I used Cycles as the rendering engine for this example.

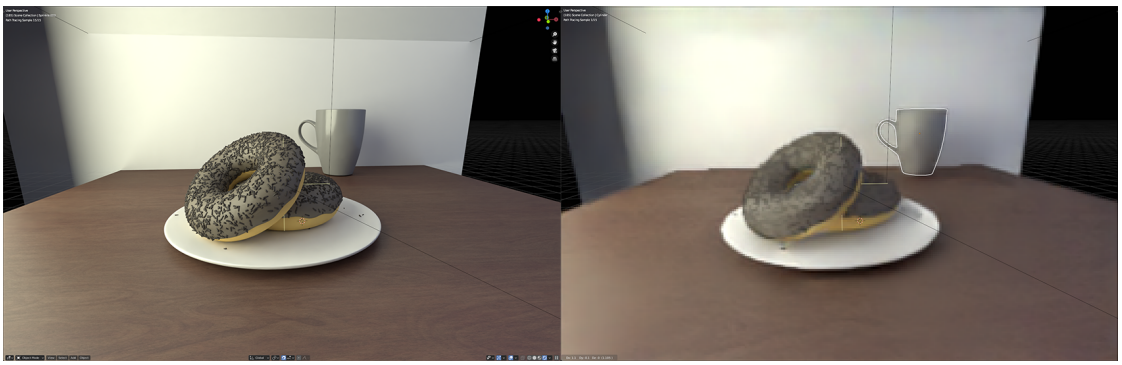

Figure 1.3: Same Example as in Figure 1.2 but with the NVIDIA Optix Denoiser enabled. The denoiser gets rid of the grain-like noise in an image reconstruction step, but the image is still considerably distorted for a few frames until it can get sufficiently refined by the pipeline.

How big of a noticeable impact could this change possibly have on the donuts in the foreground or the majority of the wall and table seen in the scene? Presumably very little. The user is most likely interested in how the cup looks and how light is reflected in its new position. There is no immediate need to rerender image regions containing the donuts, most of the table and wall. The purpose of this work is to present an alternative approach to rerendering the scene after changes to it, that gets rid of noise while preserving the advantages of being able to see the models users are working on under realistic rendering circumstances. We aim to achieve this by using information about how a given user interaction affects the scene in a 3D editing context. We want to determine an order of affected image parts, from most noticeably affected to least, and focus the available rendering budget on the most affected regions immediately, while relying on the accumulated rendering up to a triggering change to represent the less affected regions. Eventually all regions get updated in an incremental manner, when the rendering ressources become available to do so.

CHAPTER 2

Related Work

3D modeling software has existed with the trade-off between performance and visual accuracy since its first iterations. Artists, on one hand, need to see their work in high quality to get an understanding of how the finished product is going to look. On the other hand, interactability is also an issue. The longer an artist has to wait for a render after a change to a scene, the less fluent their workflow is going to be. To improve performance modern approaches usually employ a combination of adaptive sampling and image reconstruction.

2.1 Sampling

In general, to render a scene one needs to extract information like the position of geometry, lightsources, materials etc. from it. The process of retrieving one piece of such information is called sampling. In that regard the scene can be seen as a continuous signal that gets digitized into a discrete finite one through sampling [Mit90].
If a continuous signal cannot be sampled at a sufficiently high rate this leads to irreparable distortions in the signal known as aliasing [Sha49]. Since computational power is limited, sampling algorithms constantly have to battle the problem of aliasing.

Adaptive sampling means that instead of uniformly sampling a signal, one chooses different sample rates for different parts of the input. Usually the goal is to defer the most resources to regions with high frequency, like edges of objects in a scene, while using only a minimum of the computational budget on uniform areas with lower frequency. This way resources can be used optimally to achieve the best possible trade-off between quality and performance.

In [Whi05], Whitted describes how he used an adaptive sampling strategy to perform simple image reconstruction through low-pass filtering on high frequency parts of the image only. He uses this approach to counteract aliasing resulting from his suggested

recursive ray-tracing algorithm. In this early evolution of adaptive sampling, the image gets sampled uniformly as a regular spaced grid first and then the squares of the grid get subdivided recursively if the intensity in their corners differ significantly.

While being adaptive, Whitted's algorithm lacks an element of randomness which leads to structural aliasing. Later publications built on this approach and introduced a stochastic element [Mit87, Mit91]. Instead of structurally sampling points, these algorithms choose sample points at random with higher density at high-frequency areas of the signal. The advantage these approaches offer is that they yield aliasing in the form of high-frequent noise which can be effectively countered with a low-pass filter.

These first, stochastic adaptive sampling strategies are an effective tool to reduce the amount of resources needed for basic ray-tracing, however, they struggle with more complex effects like depth of field or motion blur. The reason is that these effects introduce new dimensions, outside of image space, like the position of a moving object at a given time, to the sampled signal.
Mitchell [Mit91] describes how to evaluate the quality of sampling patterns for higher dimensional problems and [HJW$^+$08] built on the work of Kajiya in his pioneering publication about the rendering equation [Kaj86] to present a multidimensional adaptive sampling strategy. They start with a stochastic initial set of samples and store them in a kD-tree, like Kajiya explored in his work. They then use an altered version of Mitchells [Mit87] contrast algorithm to find regions with high frequency within the multidimensional signal. Using that information they refine the regions stored in the kD-tree.
The basic approach does not differ all that much from Whitted's early adaptive sampling strategy: Start with a coarsely distributed sample set, find regions within the signal with high frequency and subdivide these regions to repeat the process.
The drawbacks of this technique are that adaptive sampling this way provides diminishing returns with increasing dimensionality and the image reconstruction step, which is needed afterwards, grows exponentially to the number of dimensions in cost. Overbeck also notes that this approach produces blocky artifacts for problems of high dimensionality [ODR09]. These factors considered, Hachisuka et al.'s algorithm works effectively for problems with a low number of dimensions and expensive shading cost.

Not only areas of high frequency are of interest when it comes to keeping the number of samples needed as low as possible. In ray-tracing scenarios one also should take into consideration that not all rays contribute the same to the radiance of a given point. If one casts a ray into a scene for instance and it hits a non-perfectly reflecting surface, one would have to cast an infinite amount of new rays from that point to perfectly calculate its intensity. Not all of these rays contribute the same to the result, however, since the more parallel a ray gets to the hit surface the more attenuated it becomes. Algorithms that take this knowledge into account and try to weight samples accordingly are called importance sampling.
Different approaches use different information, for instance the used bidirectional reflectance distribution function (BRDF), the function describing how light is reflected

from an opaque surface [LRR04], or the environment map or both [CJAMJ05], as criteria for importance sampling. Importance sampling can be combined with the other adaptive sampling strategies explained until now.

A form of importance sampling, called Metropolis light transport (MLT) [VG97] starts with a set of randomly traced paths from light sources to the lens and then randomly mutates them. The mutated paths get accepted or rejected depending on how much the ray contributes to the ideal image, determined by a probability function. This strategy offers a cost efficient, unbiased adaptive sampling approach that works for multiple dimensions [KSKAC02]. However, MLT does not work for blurry effects such as motion blur and depth of field.

The evolved form of a technique called lightcuts [WFA+05], multidimensional lightcuts [WABG06], offers an effective way to compute illumination under the presence of complex effects like motion blur or participating media. It works by discretizing the signal into two point sets: light and gathering points. Light points are placed at the location of the light sources in the scene and gathering points are located at the camera. The integrals of the rendering equation can then be approximated by evaluating all pairings of light and gather points. Since computing all pairings would be impractically expensive, Walter et al. expand the technique by weighting the pairings, forming a hierarchy called the product graph. The hierarchy is furthermore split into clusters adaptively. By importance sampling the most contributing clusters, also called lightcuts, the signal can be effectively approximated.

A family of algorithms analyse the signal not on a pixel basis but in the domain of so-called wavelets, a group of wave-like functions [CDF92]. The advantage of wavelets over pixels is that they offer a multi-scale representation of the image, thus being able to represent both hard edges and smooth regions[SN96].
Bolin and Meyer use the wavelet representation to introduce a sophisticated visual error metric by analyzing natural images [BM98]. Rather then necessarily focusing on areas of the signal with high frequency they use their metric to find regions where errors more noticable to the human visual system are more likely to appear and divert more resources to them. This kind of adaptive sampling allows them to ignore areas which other approaches would put high emphasis on, although the human eye would not even be able to notice the error in them.
Insipired by [CJAMJ05] success in using wavelets for importance sampling, Overbeck et al. publicated a paper on adaptive wavelet rendering [ODR09]. They suggest rendering the final image directly into a wavelet basis. By sampling the signal in wavelet form, they can find edges through the wavelet basis scale coefficients. These edges do not have to be in the image domain but can also be of high value in other dimensions of the rendering equation. Distributing more samples to these edges allows them to get an adaptive sampling strategy for multiple dimensions producing good quality images with significantly fewer samples than other approaches.

Another group of researchers explored the idea of analyzing the input signal directly in the domain of frequency. [DHS+05] prove that complex effects of light transport such as

occlusion, propagation, reflection, caustics and others can be effectively detected in the frequency domain. Others built on this research to detect and adaptively sample effects like motion blur [ETH+09] and depth of field [SSD+09].

The classic problem of monte carlo ray-tracing like path-tracing is the visual grain-like noise that it produces. The lower the sample count the more prevalent this effect becomes. [KS13] propose an adaptive sampling algorithm that predicts areas of the input signal which are more likely to produce higher levels of noise than others. Their work is similar to [BM98] where they both do not directly focus on frequency as a metric for adaptive sampling but another factor. Where Bolin and Meyer focus on the human visual system, Kalantari et al. focus on the intricacies of how typical ray-tracing algorithms work.

As time progressed, researchers realized that having adaptive sampling and image reconstruction techniques operate independently of each other was not optimal and a waste of resources. A number of algorithms emerged, which operated on the same common idea: Starting with an initial render pass, arbitrarily assigning samples, they choose a most optimal filter for reconstructing the current sampled image. Next, they calculate the reconstruction error of the image and adaptively assign more samples to regions with a higher error for subsequent render passes. Different papers have been published relying on this concept and using Gaussian [RKZ11], non-local means [RKZ12], local linear regression [MCY14] and polynomial [MMMG16] filters as a basis.
Another aspect sampling and reconstruction codependent techniques can differ in, is the reconstruction error estimation metric used. While Rousselle et al. initially used a greedy minimization algorithm [RKZ11], others and they themselves experimented with the Stein unbiased risk estimator (SURE) later on and found that they were able to use more effective filter kernerls for reconstruction as a result [RMZ13, LWC12].
[BEEM15] propose that instead of arbitrarily sampling the input signal for the initial error estimation, one should sample fewer regions of the image but those more densely in return. This way the resulting error estimates become more sparse but also more robust, ultimately leading to a more accurate selection of suitable filters for reconstruction.
To further speed up the algorithm, [MIGYM15] came up with the idea to perform the costly model reconstruction and filter optimization only for a few pixels and predicting the filtered values for the others with linear prediction models.
Ultimately, however, these algorithms all suffer from their reliance on the quality of the initial sampled image for the error estimation. If this initial step is done with to few samples they fail to reliably compute the reconstruction error, thus being unable to produce an acceptable result.

More recently, research has emerged on using machine learning [Sch15] for adaptive sampling algorithms. [KBS15], [BVM+17] and [CKS+17b] explore machine learning for removing noise from monte carlo ray-traced renders, these use however uniformly sampled signals as their input. Motivated to introduce adaptive sampling to machine learning based algorithms and combine it with image reconstruction, [KKR18] developed deep adaptive sampling. They use two convolutional neural networks (CNN) responsible for adaptive sampling and denoising respectively. The algorithm starts with sampling the

input signal with one sample per pixel monte carlo ray-tracing. Afterwards the CNN responsible for adaptive sampling assigns additional samples adaptively. The two CNNs are trained in conjunction with the goal of minimizing denoising error, thus more samples are assigned to regions of the signal the denoiser would struggle with otherwise.
[HMS+20] expand on the work of Kuznetsov et al. by introducing time as a factor for the training of their CNNs. The temporal feedback allows them to increase the effective sample count. On top of that, their network responsible for adaptive sampling is able to predict which regions of the signal are going to need more samples with the temporal reprojected data from previous frames and the geometry buffer of the current frame alone. This allows them to get rid of the initial sampling step, which [CKS+17b] had to use.

2.2 Image reconstruction

Image reconstruction is the inverse operation of sampling. Its the process of transforming a digital image back into a continuous form [Mit90]. In most cases, image reconstruction means applying some kind of filter to repair aliasing artifacts resulting from digitising the input signal. Stochastic sampling techniques, like monte carlo ray-tracing typically produce high-frequency noise. The fewer samples are used the higher the density of the noise. A basic approach to counteract this high frequency noise is a low-pass filter [Mit87, Mit91].

At the early stages of image reconstruction all the commonly used filters where linear filters. Those are filters that typically simply average the sampled values. These kind of kernels suffer from a lot of serious drawbacks, however. Too narrow filters can actually increase the amount of noise in some sections of the image, broad or wide band filters can lead to smeared or broadened boundaries and outliers (pixels whose value differs greatly from its neighbours) spread to adjacent regions through averaging. See Figure 2.1 for an illustration of this matter.

For these reasons, [LR90] suggest using non-linear filters instead. More specifically they use median and alpha-trimmed mean filters to address the problems of linear filter kernels. While these non-linear filters can produce nearly noise-free images, they introduce their own set of problems. The median filter is not energy preserving, meaning it can offset edges [McC99] or shift the colors of an image [DWR10] . The alpha-trimmed mean kernel removes all outliers even though they might be a valid part of the image [Sch13]. To avoid these issues, [JC95] apply non-linear filters only on indirect diffuse lighting of the image. This dodges the aforementioned problems but leaves open the issue of not being able to apply image reconstruction to the rest of the image.

[RW94] introduce a set of energy preserving non-linear filters and show how applying them with adaptive kernel width reduces blurring. Since the filters designed by them are energy-preserving they do not lead to color shifts like the median filter used by Jensen et al. Locally adapting the bandwidth of a filter kernel is an interesting idea to preserve high-frequency features like outliers and hard edges, but since the filters themselves do

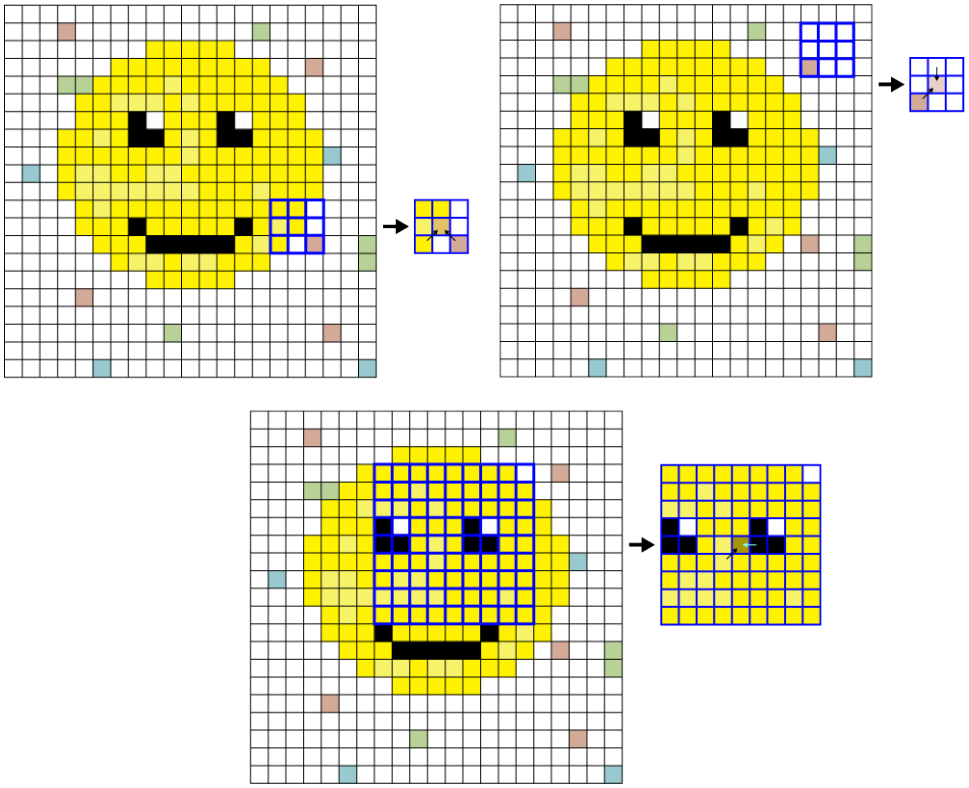

Figure 2.1: This image is meant to illustrate from the upper left to the lower one: 1) An outlier, in the form of noise, shifts the color and intensity of the regions lying around it when averaged over with a linear mean filter. The effect becomes more drastic with a narrow filter and more widespread with wide one. 2) In a narrow filter outliers like noise have a heavy impact and can spread the noise to adjacent pixels. 3) A too wide linear mean filter averages over many pixels for every filtered value which can lead to distortions reaching from blurry features to a homogeneous image where every pixel has the same color.

not respect such features, they still can get spread out through averaging leading to blurry edges and similar artifacts.

Instead of applying the reconstruction filters as a preprocessing step, like most methods did at the time, [SW00] opted to apply them during image formation. They used a variable bandwidth strategy similar to Rushmeier et al. and suffer from similar problems as in that their method also blurs regions of high frequency, which is more noticeable the fewer samples are used.

[McC99] explores the possibility of using a technique called anisotropic diffusion for noise reduction produced by monte carlo ray-tracing. It's a filter based on an interpretation of Gaussian scale-space [WRV98] and it offers the great advantages of being energy- and edge preserving. However, this method has proven to be difficult to implement into a

progressive rendering pipeline since its incremental nature proves too costly to perform every frame [Sch13].

[XP05] use bilateral filtering [TM98] for noise reduction in monte carlo ray-traced images. Bilateral filtering is a method based on the idea that two pixels are related to each other not only in spatial distance to each other but also in their intensity. To respect that idea two filters are used in combination for bilateral filtering: a spacial and a range filter. As a result applying the combined filter means that, each pixel is replaced with a weighted average of its neighbours. The values of the neighbours are weighted by a spatial component that favors nearer pixels over more distant ones and a range component that in turn favours neighbors closer in intensity. This leads to nearby, similarly intense neighbors contributing the most to the filters result for the current pixel. This way the filter can smooth an image, removing noise while preserving edges.

Outliers are a big problem for filtering techniques, they can lead to objectionable artifacts if spread by an averaging filter. At the same time simply removing them leads to a biased result since they could be a valid part of the sampled scene. [DWR10] formulate an outlier rejection algorithm that rejects outliers based on their joint density, meaning, similar to bilateral filtering they consider two pixels to be close when they are both close in spatial location and intensity. Removing the visually perceived outliers this way, does reduce the introduced bias and conventional filtering methods can effectively remove the remaining noise. Still the resulting Image is biased. [PBP11] present a similar method using density estimation to reject bright spots from monte carlo ray traced images.

As mentioned before, [KBS15], [BVM$^+$17] and [CKS$^+$17b] present how to use machine learning for image reconstruction. They train deep convolutional networks to learn the highly complex relations between a noisy image and its ground truth to then predict optimal filter kernels for reconstruction.

The work that we present in this writing does not incorporate image reconstruction, however, it could be combined with the techniques mentioned in this section in order to improve performance.

CHAPTER 3

Method

The technique we present attempts to use information about the user input in a 3D editing context for adaptive sampling. We aim to determine which parts of a scene are noticeably affected by a given user interaction, in order to use this information to prioritize these parts during an incremental rendering algorithm. Instead of wasting resources on regions of a scene that have not been drastically changed by an interaction, we want to focus on the parts that actually look different after the change. In order to achieve this goal we apply the following steps: Starting from a base render of a scene, the initial render before any user interaction has been made, after a user changes part of it, we incrementally rerender the regions most noticeably affected by the change first, leaving the other parts of the image unchanged from the base render. We rerender the image step by step from most to least affected image region, replacing the base render whenever a whole region is done. The aim of this technique is that a user gets a quick high quality render of the regions actually affected by his or her change, while less affected regions, which rerendered likely look close or even identical to the render before the interaction, get updated slowly in the background. This way the user should be able to work with less interruption.
Having explained the general idea behind it, it has to be stated, that the main focus of this work lies clearly on presenting a method on how to use an order of image regions, most affected to least, for an incremental, adaptive and interactive rerendering algorithm, not as much on how to generate such an order. Because of that, we make reasonable assumptions for select interactions on how to generate the order of impact to act as a proof of concept and leave a more thorough research of the matter open for further work.

For our algorithm to work, two main steps have to be performed for every user interaction: First, for every image region, the magnitude of impact of the interaction on it has to be determined, so that the regions can be ordered from most to least affected.
Secondly, when the order of regions has been determined, we immediately invest all available rendering resources on rendering the most affected region first. Then we render

the second most affected part and so on. Basically, the image has to be incrementally rerendered according to the order of impact. Until a region gets updated, we rely on previously accumulated color information to present the region. For this to work of course, an initial render of the whole scene must be made to fall back on during the first incremental rerender.

Because of it's incremental nature, the effectiveness of our method relies on the assumption, that image regions further down in the order of impact, have been little to not at all affected by a given user interaction, so that it won't be noticeable to the viewer if we don't update them immediately. That means, the more uniformly an interaction affects the whole scene, the less effective our approach becomes. It works best for small local changes, where only a small portion of the scene is affected noticeably, while its effectiveness diminishes the more of the image is affected, as in those situations it will be increasingly noticeable to the viewer that we update regions further down in the order later. Formulated differently, the effectiveness of our method scales directly in proportion to the spread of the magnitude of impact in the image region order. Ideally, the first few regions in the order are heavily impacted by a change. Then the magnitude of the impact falls of rapidly going further down the order, until the last regions are not affected at all. The consequence of this, is that our approach does perform suboptimal in situations where the image regions are affected uniformly. Interactions, such as changes to the camera's angle or orientation, which affect the whole image uniformly throughout are the worst case scenario for this method. It would produce distorted looking results, as the difference between already updated regions and the ones not, could be potentially very stark, leading to a split image, where one part is updated and shows the new scene, while the other is still in its old form. See Figure 3.1 for reference. The effect is comparable to old monitors which had no image buffering feature. The image would get updated in a way of a scan-line running from top to bottom, overwriting the old image with the new one. To accommodate this problem, we implemented logic to handle interactions uniformly affecting the whole scene differently than non-uniformly affecting interactions.

3.1 Splitting up the image into regions

For our approach it is essential that we divide the picture into uniformly big parts that we can render individually, we will call them tiles. We adapted a standard path-tracer in such a way that we can load the same tile into it multiple times, basically instead of the full viewport we load in a buffer of the same size, filled with only a few different tiles repeated multiple times. See Figure 3.2 for reference. Doing so, we can then let the path-tracer sample this buffer the same way as it would a complete scene of the same size, effectively meaning we render the same few tiles multiple times concurrently at a relatively low sample rate. We use a compute shader afterwards to combine all these low sample results of each unique tile into one, yielding a few high sample tiles in roughly the same time as it would take to render a whole scene of the same size at a low sample rate. We can only do this, because we used an unbiased renderer as the base for the implementation of our adapted path-tracer. An unbiased renderer produces the same

Figure 3.1: This image is meant to illustrate how an incremental rerender performs poorly for changes affecting a large portion, or even all of the scene heavily or even uniformly. I rotated the ground plane in the scene and captured the screen midway through the update. The red line splits the image into two parts, the upper being the updated one, while the lower still shows the old image. It can be seen, that because of the change affecting virtually the whole scene heavily, no regions cant be effectively isolated and rendered later than others without causing visual disturbances.

result wether you let it run N times with one sample per pixel and average the results or run it once with N samples per pixel over the same scene [KA91]. This is exactly what we are doing, made possible because of the lack of bias in our renderer. Doing the same with a biased sampler would lead to distortions since it would produce potentially vastly different results doing it one way or the other.
We deliberately implemented the logic to handle individual tiles this way over adapting the path-tracer to render single tiles successively at a high sample rate, because the architecture of modern GPUs is optimized to handle a large number of simple jobs concurrently, like it has to with our approach, rather than a few jobs with a heavy workload consecutively, which would be the alternative.

3.2 Determining most to least affected regions of the image

Optimally entire papers could be devoted to finding an optimal metric for recognizing how noticeably a specific image region is affected by any given user interaction. Using such a metric, one could then build a sophisticated list of image regions, ordered from most to least affected. This list could then be used for our incremental rendering method

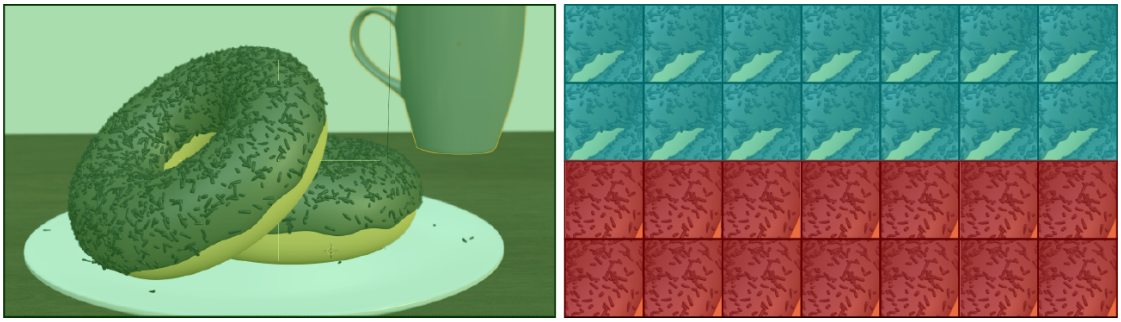

Figure 3.2: This figure illustrates two different representations of buffers of the same size for a path-tracer. On the left, in green, we see the whole scene being loaded into the buffer once. This is how a standard path-tracer would typically operate. On the right we see our method of loading in only a few tiles into the buffer, repeated multiple times. In blue we see the first tile being repeated and a second one in red.

to rerender highly affected regions first and little affected ones last.

As I mentioned, however, this is not the main focus of this work, so we simply assume one point of the image as the epicenter of the change. I will call it the point of change (PoC) from hereon out. We define the point of change as the most affected point of the scene or at least the center of the most affected region. A suitable candidate for the PoC could for instance be the center of the bounding box of an object in the scene that has been transformed (moved for example) by the user.
We simply assume that the relative spatial position of an image region relative to the PoC of a given interaction, directly corresponds to how much it is affected by that change. This means the closer an image region lies to the PoC, the more affected we consider it. Following this assumption, we implemented two ways of ordering the image regions: a base update queue and a spiral update queue. The base queue simply orders the tiles from the upper most left region to the lower most right. We use this ordering method for global changes, like the cameras position, where it makes no difference in which order we prioritize the parts of the image, as they are all affected the same.
The spiral queue gets built every time a non uniformly affecting user interaction occurs. It spirals outwards from the PoC of that interaction. See Figure 3.3. This outwards spiraling order corresponds to our assumption of spatial proximity equaling rate of affection. We use this queue for our incremental rerendering approach.

3.3 Building the spiral queue

The spiral queue gets built by traversing the tiles in an outwards spiral from the point of change, which has been normalized by dividing it through the image dimensions. The spiral motion works by stepping in one direction for a certain number of steps (step size), changing the direction every time the step size has been reached. Every second step completion, the step size gets incremented.

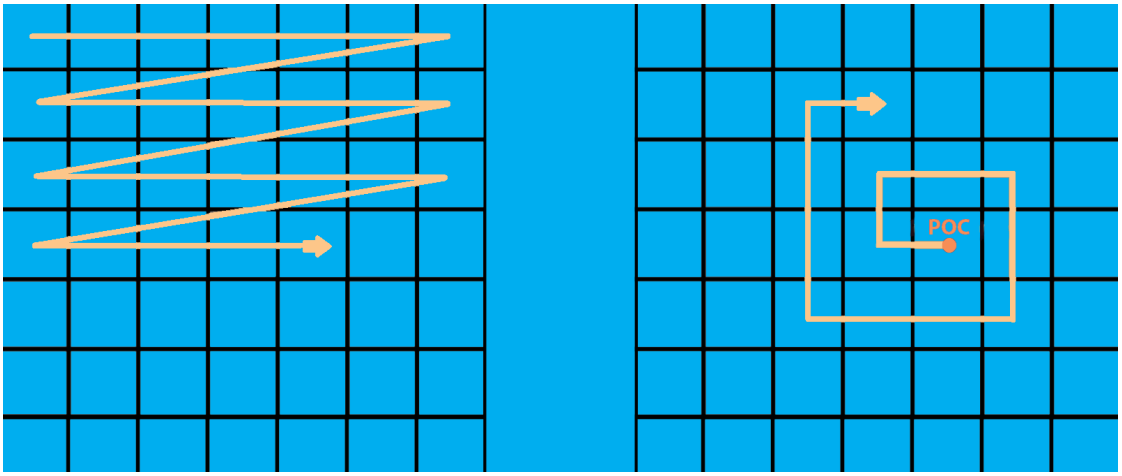

Figure 3.3: The basic default image update queue vs the spiral queue. The basic queue traverses the tiles from the upper most left corner to the lower most right, while the spiral queue spirals outwards from the point of change after an appropriate change.

We want to have every tile placed exactly once within our queue, as completing the queue marks our incremental rendering approach having updated the image once completely in high quality and we can switch back to a continuous refining rendering mode. In that regard a problem we are faced with, is that most of the time the viewport won't be a perfect square and the PoC will probably not lie at its center. Because of that, blindly traversing the tiles in an outwards spiral from the PoC will inevitably lead to hitting the viewports boarders without having reached every tile once. To solve this we use a brute force approach: we pretend that the rectangular viewport is part of a larger square with the PoC at its center, we then traverse this square's imaginary tiles in an outwards spiral from the center, until we have reached every tile of the viewport exactly once. At each tile, we check if its an actual tile of the viewport or just a theoretical one from the surrounding assumed square, so that we only place actual tiles of the scene into the queue. See Figure 3.4. We know when to stop the building algorithm, because we know how many tiles there are at the time we start building the queue and this algorithm should never visit one tile more than once. So we can simply stop building when we have a number of tiles in our queue equal to the total number of tiles present at the start of the process.

3.4 Switching modes

We designed our program to operate in 3 distinct modes. These modes mainly differ in two ways: Firstly the number of copies per tile we place into the path-tracers buffer, I will call it the number of repetitions(NoR). We use a fixed sample rate and tile size and thus only control the number of samples per pixel that should be dedicated to a single tile per render through the NoR. If we want individual tiles to get rendered at high

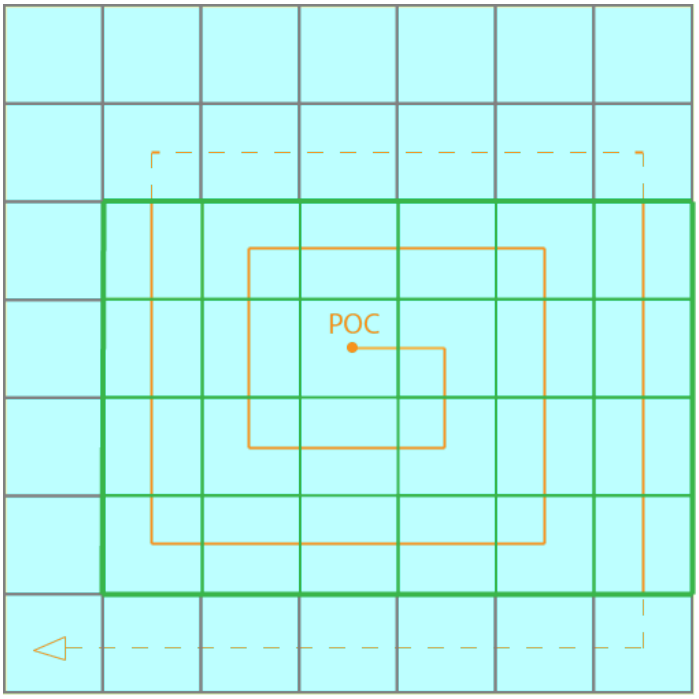

Figure 3.4: This figure visualizes the assumption we make, when the PoC is not at the center of a square viewport. We pretend the PoC (orange) lies at the center of a bigger square (gray) which encloses the rectangle of the viewport (green).

quality we set the NoR high and low if we want only a rough render. The NoR directly determines how many tiles get rendered in one rendering step. As both the size of the path-tracers buffer and the size of one tile is fixed, the number of how many tiles get updated in one rendering step calculates as the division of the buffer size through the tile size multiplied with the NoR. So for example if we used a buffer size of 400x400 pixels, a tile size of 10x10 and a NoR of 4 the number of tiles completed in one rendering step would calculate as 10. We use a fixed sample rate of one path traced per pixel for the whole buffer, so the NoR incidentally equals the number of samples per pixel, per tile. For instance if we decide to repeat each unique tile 200 times, each copy gets rendered with one path per pixel, so 200 paths per pixel of the tile get traced. If we used a higher sample rate the number of samples per pixel, per tile would result from the multiplication of this global sample rate and the NoR. To stay in the previous example of an NoR of 200, if we used a global sample rate of 10, that would mean we would trace 2000 paths per pixel of each unique tile loaded into the path-tracer.
The second part the modes of operation differ in, is the strategy to accumulate the color and intensity samples taken by the path-tracer per pixel. This accumulation is done in an extra step once per frame. As I explained, our incremental rerendering method potentially produces distorted looking results for uniformly affecting user interactions. Thus, for global changes, like changing the camera, we throw away the image data accumulated up

to this point and start rendering the scene completely from scratch, which equates to no accumulation at all. In these base renders we quickly render the image with a NoR of 1, which means every tile gets rendered exactly once in one rendering step, to achieve a coarse image as fast as possible with the basic default update queue. After we have completed this initial coarse image to work with, we switch to the second mode, which continuously refines the image by accumulating the samples until another user interaction occurs. It does so by repeating the first mode over and over, combining the results into a continuously more refined accumulation of color and intensity values. These two modes in conjunction basically emulate a standard whole-image path-tracer.
When however, a user interacts with the scene in a non uniformly affecting manner, we switch to mode number three, incremental rerendering. We determine the PoC, build the spiral queue and start traversing it outwards from that point. We do so using a high NoR limiting the number of unique tiles rendered by the path-tracer in one step to only a few, to achieve that those get rerendered in high quality. During this incremental rerendering mode we replace only those tiles that have been completed in this manner and keep the other's accumulation up to the point when the process of the incremental rerendering had started intact. This way the most affected regions of the image get updated in high quality as soon as possible, while less affected regions get rerendered later. As soon as the spiral has been traversed once, we switch back to the second mode that continuously refines the image. See Figure 3.5 for an illustration of the 3 different modes.

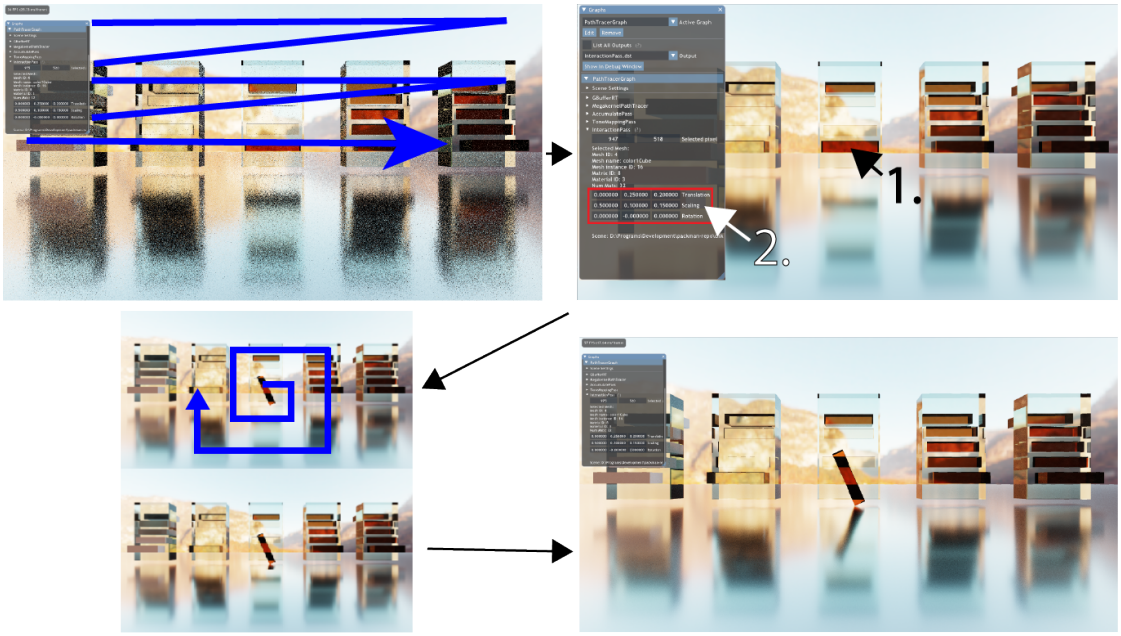

Figure 3.5: Starting from nothing, a base render is produced using the base queue, no accumulation and a NoR of 1 (upper left). This is the first mode our program operates in. It is used whenever a uniformly affecting user interaction is applied to the image and for the initial render at the start. After the first render we switch to the second mode, continious refinement, which repeats the first mode continuously and accumulates the results to refine the image (refined image in upper right). In this example, I then select the red cube in the center of the scene by right-clicking it. Then I change its rotation in the GUI on the left (upper right image). This is an user interaction where our third mode, incremental rerendering, is applicable. So we use it to rerender the image with the spiral queue and a high NoR, replacing only finished tiles (lower left). Finally, the program switches back to continuous refinement (lower right).

CHAPTER 4

Implementation

In the following chapter, I will explain the more technical details of how we implemented our suggested incremental rerendering method into NVIDIAs rendering framework Falcor.

4.1 Software environment

Initially, we tried to implement our approach into the open source 3D modeling software "Blender", specifically using the "Cycles" ray-tracing renderer developed and integrated into Blender by the Blender Foundation and Cycles Team [Com18]. We wanted to tie our logic into the workflow of using Cycles as the real time renderer in the work-space of Blender. That way we would have been able to leverage the powerful editing tool-set of Blender and could have achieved a high level of comparability to Blenders existing approach for our users.
Unfortunately, we ran into many problems working with the Blender source code and had difficulties implementing our desired logic into this kind of monolithic code base. We came to the conclusion that the amount of work needed to complete the project in the Blender environment was infeasible for the desired scope. A switch to a different environment, NVIDIA's Falcor open-source real-time rendering framework was made [BYC+20].
Falcor is a light-weight framework, designed with modularity and expandability in mind. That made it easier for us to implement our logic. We specifically used the included "Mogwai" software as a graphical user interface and the "PathTracer" render graph as a starting point. The drawback of NVIDIA's code base is that it does not come with a 3D editing tool-set. As Falcor is geared more towards pure rendering projects, there is per default no way to edit scene geometry. Because of this, we had to implement rudimentary editing capability ourselves.

The Falcor framework focuses on the concept of render graphs and render passes. A render graph consists of any number of render passes, which in turn stand for one single step in

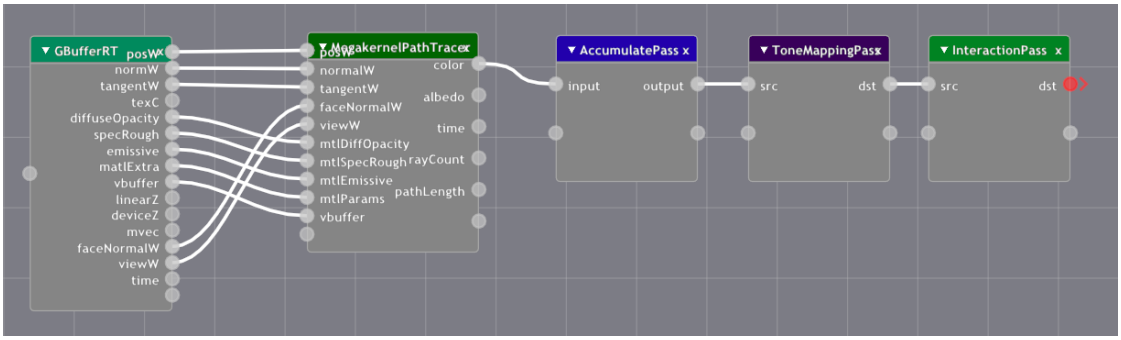

Figure 4.1: Updated render-graph, now including the new InteractionPass.

the rendering pipeline, implemented by that graph. Starting with the "PathTracerGraph" render graph we made the following three major changes and additions:

- Adapting the path-tracer
- Adapting the AccumulatePass render pass
- Adding a custom render pass for handling user interaction

The "mega-kernel path-tacer" is the implementation of a path-tracer that comes with the Falcor framework. We adapted it and use it as our sampler to retrieve color and intensity information form the scene. The "AccumulatePass" is used to accumulate the samples of the path-tracer. Before our adaptions to it, it basically always operated in the first two modes described in the corresponding section of this work: Section 3.4. It threw away all accumulation if any change to scene was made, triggering a full rerender and then stayed in continuous refinement mode until another change occurred. We implemented the third accumulation strategy needed for our method and logic to switch between the 3 distinct modes. We added our new "InteractionPass" at the end of the pass loop of the PathTracerGraph. The mega-kernel path-tracer and the AccumulatePass were adapted to react to triggers from the InteractionPass, but only if it is present and they still function independently without the other passes, to keep Falcor's design principle of modularity intact. See Figure 4.1.

4.2 Adapting the path-tracer

Falcor's mega-kernel path-tracer had to be adapted to work on a queue of tiles rather than the whole viewport. As I explained before we achieved this by implementing a way to load a buffer of viewport size into it, holding a few distinct tiles repeated a number of times determined by the NoR, instead of the whole scene. In Addition to that we had to implement a compute shader that combines all the copies of the same unique tiles, rendered simultaneously by the path-tracer with one sample per pixel in one rendering

step, into high quality renders with NoR times global sample rate of traced paths per pixel.
We use a square tile size of 16 by 16 pixels and a global sampling rate of one path traced per pixel, where one path is allowed a maximum of 3 bounces. We keep the NoR at 1 for the first two modes of operation, to emulate a standard path-tracer and only change it to a value of 256 during the incremental rerendering mode.
We needed to implement logic that switches between the base and the spiral queue and the right NoR according to the current mode of operation. For that, the path-tracer queries triggers placed into shared memory by the InteractionPass to determine which is the current mode. If the Interactionpass is not present within the rendergraph the path-tracer only operates within the first two modes and performs basically the same as a standard whole image path-tracer. The spiral queue building logic also resides within the source code of this render pass. It uses the PoC, also placed into shared render pass memory by the InteractionPass. Lastly the path-tracer passes along the result of every frame further down the rendering graph, in our case to the accumulation pass.

4.2.1 The spiral queue

```
1  void buildSpiralQueue(uint2 point_of_change, uint2 gridDim)
2  {
3      uint2 grid_PoC = point_of_change / uint2(tileSize, tileSize);
4
5      spiralQueue.push(grid_PoC);
6
7      int x = grid_PoC.x;
8      int y = grid_PoC.y;
9      int steps = 1;
10     int direction = 0;
11
12     while (spiralQueue.size() < gridDim.x * gridDim.y)
13     {
14         for (int j = 0; j < steps; j++) {
15             switch (direction)
16             {
17                 case 0: x++; break; //RIGHT
18                 case 1: y++; break; //DOWN
19                 case 2: x--; break; //LEFT
20                 case 3: y--; break; //UP
21             }
22
23             // brute force solution for the fact that the PoC may not be centered
24             // and the grid may not be symmetrical
25             if (x >= 0 && y >= 0 && x < (int)gridDim.x && y < (int)gridDim.y)
26             {
```

```
27                  spiralQueue.push(uint2(x, y));
28              }
29          }
30          direction = (direction + 1) % 4;
31
32          // every two turns the step size increases
33          if ((direction % 2) == 0)
34              steps++;
35      }
36  }
```

Source Code 4.1: The method that builds the spiral queue written in C++.

As can be seen in the code snippet: Source Code 4.1, the building of the spiral queue boils down to a simple loop, which ends after every tile has been added to the queue. Dividing the point of change through the tile size normalizes it for a normalized tile grid. The grid is given by the parameter "gridDim", a two dimensional point with the width(x) and height(y) of the grid. The gridDim gets computed by dividing the viewport dimensions through the tile size. This normalization process allows us to move inside a grid where every point from (0, 0) to (gridDim.x, gridDim.y) corresponds to the index of a tile. To get the position of the tile in screen space one simply needs to multiply the index with the tile size afterwards. Using this normalized grid, the algorithm traverses it in a second loop, which performs a number of increments or decrements on the current index according to the step size per step of the overarching while loop. According to the current direction, either the x or y axis index get incremented or decremented.
To handle the problem of non-square viewports and non-centered PoCs, described in Figure 3.4, we check each index if the corresponding theoretical tile actually lies within the image or is just part of the assumed square encompassing it, before inserting it in the spiral queue.
After the step size loop has finished, the direction gets incremented and modulated by four so that the order switches every repetition between right, down, left and upwards. Lastly, every second step completion the step size has to be incremented as the steps of an outwards spiral get bigger after every second direction change.

4.2.2 Switching modes

As I touched upon, the algorithm runs in continues refinement mode per default and uses a NoR of 1 per Tile. If a change to the camera occurs, all accumulation up to that point gets thrown away and the refinement process starts anew with a single run through of mode 1 and then reverting back to mode 2 until another interaction is made by the user. If the point of change has been set by the InteractionPass, it means a non-uniformly affecting user interaction was made and the path-tracer switches to incremental rerendering. For that, it uses the spiral queue and a NoR of 256 per tile. This queue gets rendered exactly once, if not interrupted, then the path-tracer switches back to the base

queue and a NoR of 1 per tile, for continues refinement. If the completion of the spiral queue gets interrupted by another user interaction before it could finish, the queue gets replaced by the new spiral queue and the path-tracer starts to render this one instead. Since the incremental logic updates the most affected parts of the image first and does so at a high quality, the image should stay in a satisfactory state even when the user keeps interrupting the path-tracer with rapid interaction chains, as long as it gets to finish a minimal amount of tiles per interaction.
Reference Figure 3.5 from the method chapter for an illustration of the different modes.

4.3 Adapting the accumulate pass

The accumulation pass accumulates samples of the path-tracer in three different accumulation modes. As outlined, the default mode is to continuously accumulate samples into a more and more refined image. The other two modes are only employed when a user interaction occurs. Global, uniformly affecting changes trigger the accumulation up to that point to be thrown away, while all other interactions trigger the incremental rerendering process, where the accumulation gets replaced tile by tile as they get completed. After the image has been completely rendered once in either mode, the logic returns to its default continuous refinement mode.
To determine the correct mode the accumulation pass first queries the Falcor scene. Falcor offers methods to query information about recent scene updates since the last query. Through this interface we determine if a camera change, jitters excluded, has occurred since the last frame. If that's the case we have to reset the scene. If no change has been made to the camera, the pass queries a parameter, which is set by the mega-kernel path-tracer, if it should use incremental replacement accumulation or continuous refinement.

4.4 Interaction pass

Since Falcor offers no functionality to support user interaction, beyond moving and rotating the camera, at the time we worked on this thesis, we developed a custom render pass, the "InteractionPass", to react to user input. So far we handle two different kinds of interaction. The first is when the user clicks on the viewport in Mogwai. In that moment we use a custom shader to trace a single ray from that pixel into the scene. The first piece of geometry this ray hits gets marked as the currently selected object. When an object is selected we display a set of information about it in a designated GUI element implemented into Falcor's render pass UI. Among it we display the object's position, scaling and rotation in Euler angles. The second user interaction we track, is the user manipulating these values, by either inputting new ones via the keyboard or clicking and dragging the cells directly. When the user changes the values for position, scaling or rotation, we apply the corresponding transformation to the object. See Figure 4.2.

The second responsibility of the interaction pass is passing on information to the path-

tracer and the accumulation pass. Whenever the user manipulates an object, we trigger the path-tracer and accumulation pass to switch to the spiral queue incremental rendering mode. The interaction pass currently passes along the selected pixel, which is the last pixel a user clicked on during object selection, as the point of change for building the spiral queue.

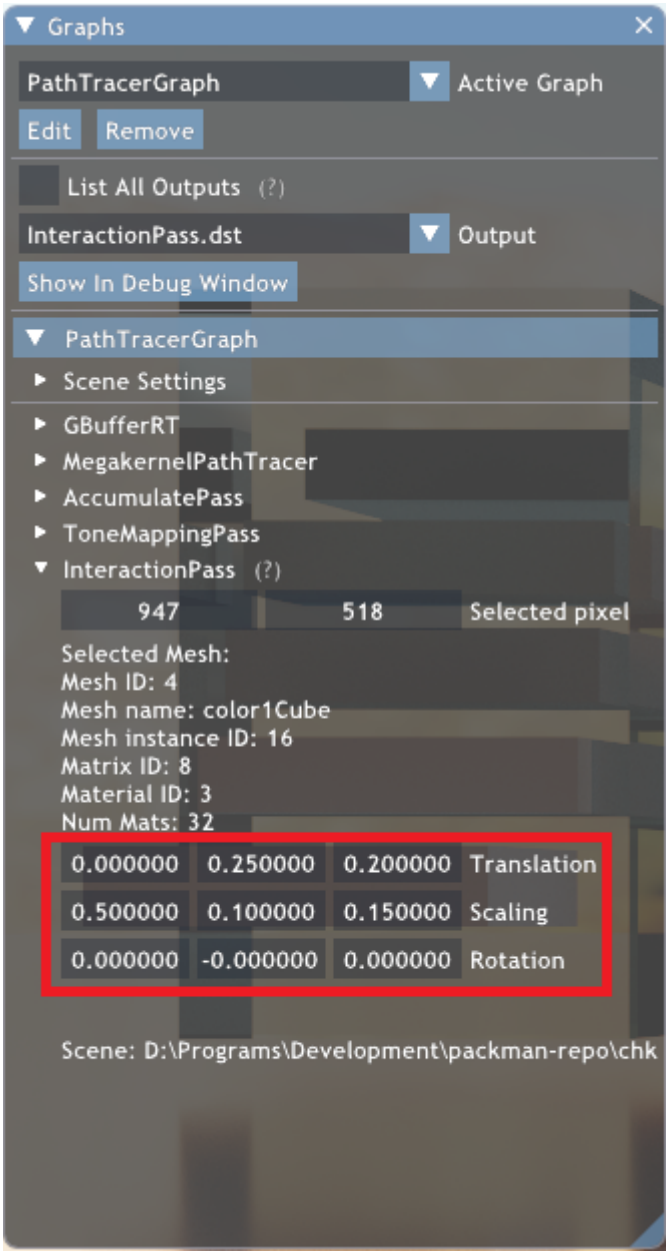

Figure 4.2: The user interface we implemented into Mogwais own render pass UI. It displays a selection of information about the selected object. The section marked in red, shows position, scaling and rotation of this object in Euler angles. Its values can be modified either by clicking and dragging or by directly altering them with the keyboard.

CHAPTER 5

Evaluation

In this chapter I will try to compare our method with classic path-tracing, evaluate the effectiveness of our concept and discuss strengths, weaknesses and how our algorithm could be improved upon in future work.

5.1 Hardware environment

The hardware I worked with on this project and that I used to run all the tests, evaluations and representations shown, is as follows:

- CPU: AMD Ryzen 7 3700X
- GPU: NVIDIA GeForce RTX 2070 Super
- RAM: 16 GB DDR4
- OS: Windows 10 Pro x64

5.2 Results

Since we aimed to minimize visual disturbance in-between user interactions, the question if our incremental rerendering approach is less, similarly or even more intrusive to the viewer than conventional path-tracing remains a highly subjective matter. The best way to evaluate a work such as this one, would be a user study among typical users of 3D editing software, however, this is beyond the scope of this bachelors thesis, so I will instead provide basic use case examples and apply common sense reasoning to evaluate them. An appropriate user study could be subject of future work on the matter.
I am going to give a comparison between our approach and Falcors standard path-tracing. I used the scene "Nested_Dielectrics" included in the Falcor repository. It offers complex

reflections and caustics with a significant number of partially translucent objects. The program runs at 60 frames per second on my test hardware.

For the first comparison I shrank a relatively small cube. This user interaction impacts a comparatively small area of the scene. See Figures 5.1 and 5.2.

With classic path-tracing, typical grain like noise is instantly noticeable. The more frames pass after the interaction the smoother the image becomes again. Notice after 10 frames the image is still noticeably noisier than before the interaction.
Our approach shows no grain like noise at all. The image gets gradually updated with all newly rendered parts already in high quality. After only 10 frames the image is basically completely updated and ready for further work.

For the second evaluation I rotated another relatively small cube along the z axis. This user interaction impacts a moderate area of the scene. See Figures 5.3 and 5.4.

Similar observations than in the first example can be made, but because a larger area of the scene has been affected, our approach takes longer to achieve a complete update of all the noticeably affected image regions. Notice that after 10 frames the immediate area around the cube has been updated completely while the reflection of the object in the reflecting ground surface, which is further away from the point of change, has not.

For the third comparison I moved a larger cube along the x axis. Since the sheer size of the cube takes up a large portion of the image, the change affects a comparatively big part of it. See Figures 5.5 and 5.6.

With this change affecting such a large part of the image, the classic path-tracing performs considerably better than in the previous examples. After 15 frames the classic approach has reached a noise level not too far off from our method showing the whole image in its new state, while our algorithm only manages to update the parts of the image containing the cube itself after 30 frames, still not correctly showing the reflection in the ground. This is partially because of our suboptimal method of determining the order from most to least affected part of the image simply by spatial distance to the PoC, but it shows that our program gets less effective the more parts of the image are considerably affected by a given user interaction.

To offer a more numerical comparison between classic path-tracing and our incremental method, I made an user interaction that affected a moderately big part of the image, namely rotating the red cube in the center of the scene once again, like I did in the second example above. I rendered a very high-quality ground truth (GT) of the scene with standard path-tracing after this change was made. See Figure 5.7. Then I compared this GT with the results of classic path-tracing and our method using different NoRs, after 5, 10 and 30 frames. I calculated the normalized root mean squared error (rmse) between the GT and each of these results: See Table 5.1.

At first glance, it can be seen that our method performs significantly better in all captured cases, going purely by the rmse values. An interesting observation is that while the values go down quite linearly with classic path-tracing the more frames have been rendered,

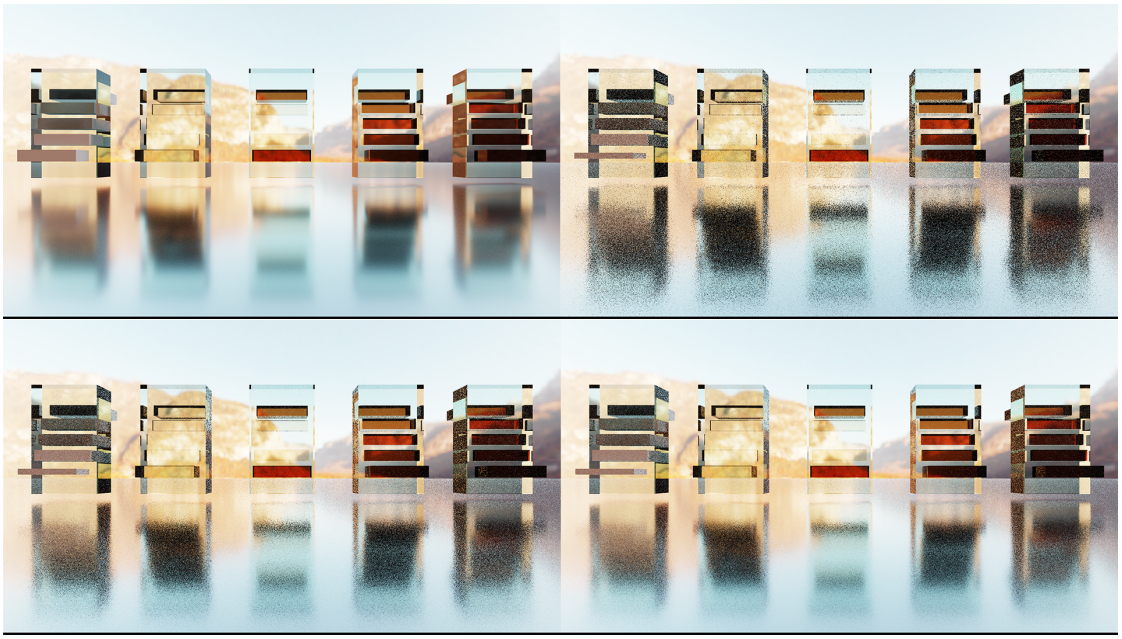

Figure 5.1: Shrinking of a relatively small object, the grey cube centered vertically at the left of the scene, rendered with classic path-tracing. The viewport was captured, in this order, 0, 2, 5 and 10 frames after the user interaction.

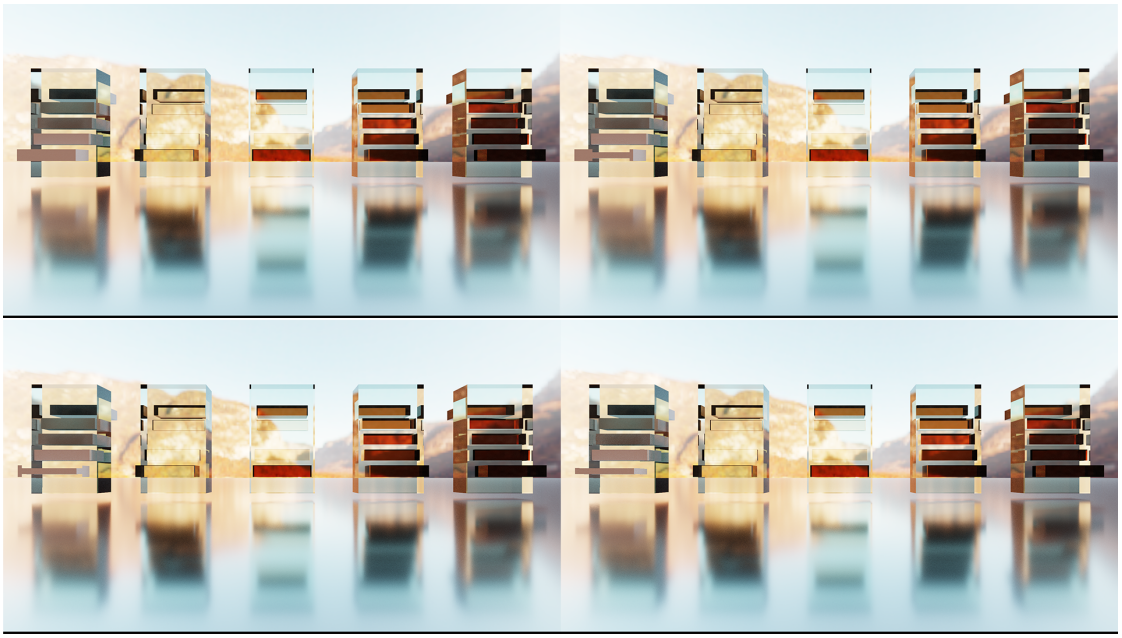

Figure 5.2: These images show the viewport captured at the same interval, after the same user interaction as in 5.1, but rendered with our incremental path-tracing.

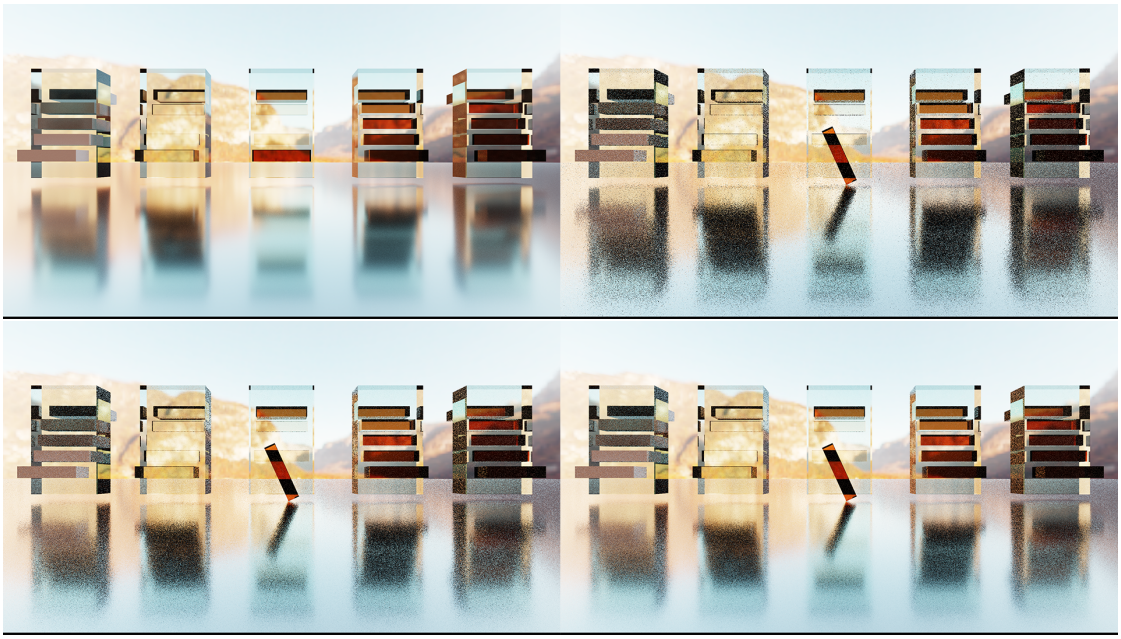

Figure 5.3: Rotation of a relatively small object, the red cube at the center of the scene, rendered with classic path-tracing. The viewport was captured, in this order, 0, 2, 5 and 10 frames after the user interaction.

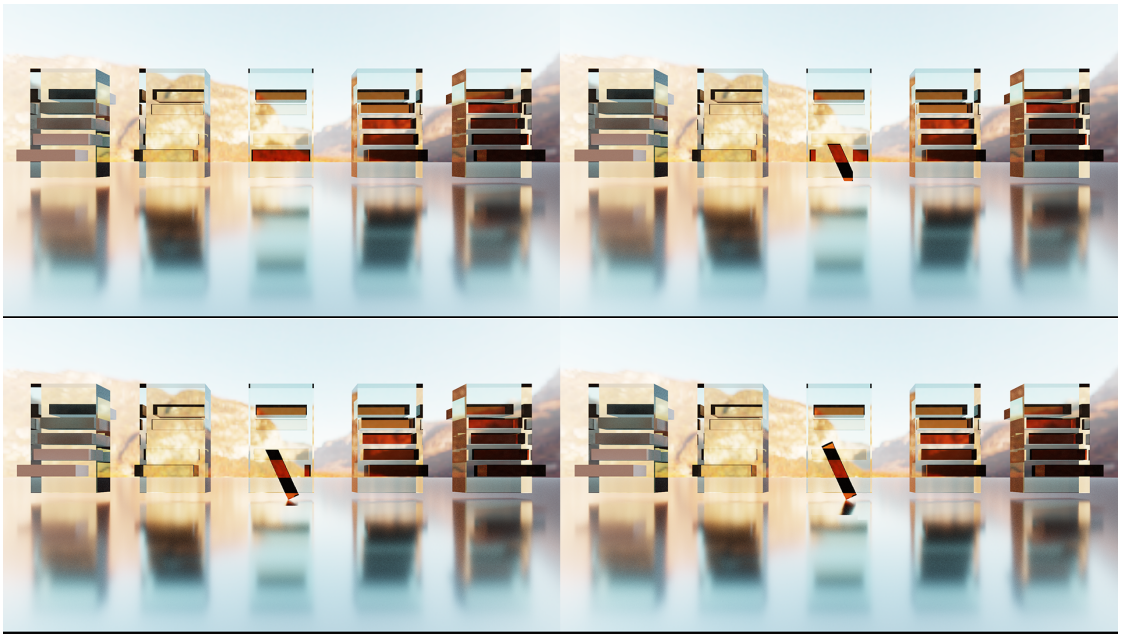

Figure 5.4: These images show the viewport captured at the same interval, after the same user interaction as in 5.3, but rendered with our incremental path-tracing.

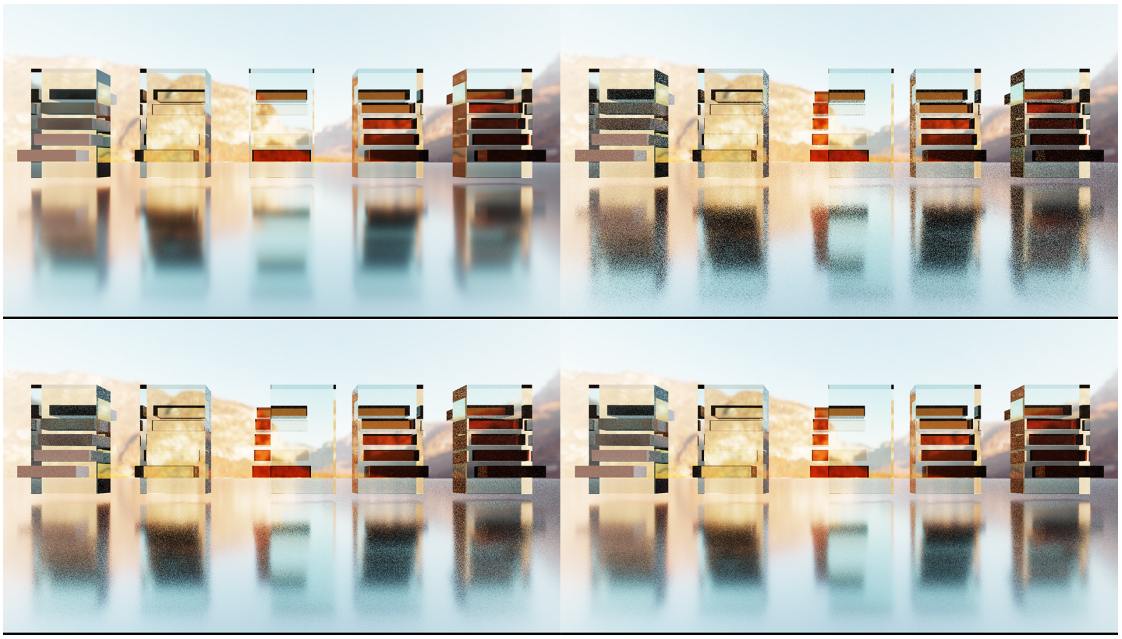

Figure 5.5: Translation of a relatively large object, the large translucent cube at the center of the scene, rendered with classic path-tracing. The viewport was captured, in this order, 0, 5, 15 and 30 frames after the user interaction.

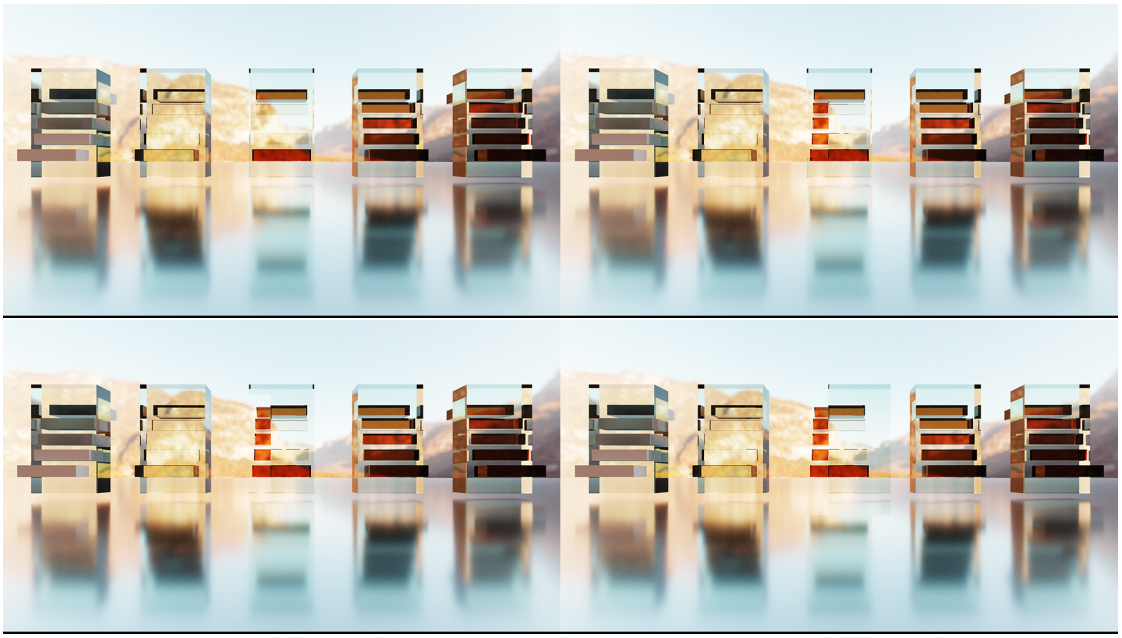

Figure 5.6: These images show the viewport captured at the same interval, after the same user interaction as in 5.5, but rendered with our incremental path-tracing.

Figure 5.7: The ground truth for a roation along the z axis of the red cube at the center of the scene.

	Standard PT	NoR 64	NoR 128	NoR 256
5 frames	0.11519 (∼12%)	0.01523 (∼2%)	0.01335 (∼1%)	0.01506 (∼2%)
10 frames	0.08106 (∼8%)	0.01788 (∼2%)	0.01539 (∼2%)	0.01425 (∼1%)
30 frames	0.04632 (∼5%)	0.02885 (∼3%)	0.02048 (∼2%)	0.01628 (∼2%)

Table 5.1: This table shows the normalized root mean squared error (rmse) between the GT (Figure 5.7) after rotating the red cube at the center of the test scene and the results of standard path tracing and our incremental method using different NoRs, after 5, 10 and 30 frames.

that is not the case for our incremental rerendering. The values fluctuate and generally even minimally increase after more frames. This is explained by the fact, that depending on the NoR used, which in this case is the same as the number of samples used per pixel, per tile, the quality of the newly rendered tiles is lower than the high quality GT and minimal levels of noise are present in these compared to the GT. In this example, after 5 frames, the immediate area around the interaction has been updated, while the rest of the image is still taken from the high quality accumulation up to the rerender. So after 5 frames, the tiles that look most different in the GT compared to the accumulation up to the point of the interaction have been updated already, while at the same time minimal noise has been introduced as only a few tiles have been updated with renders using the lower sample rate. After more frames have passed, more tiles have been updated with less samples, so the quality difference between the GT and the incremental render shows

in the rmse values. Generally the value will drop for the first few frames, depending on how much of the scene a given user interaction has affected, while the most affected tiles are being rerendered, then increase slightly as little to not changed regions get updated in lower quality than the accumulation up to the rerender. Nevertheless, the values will stay relatively low throughout the whole rerendering process.

The trade-off between our approach and classic path-tracing is that with classic path-tracing the viewer sees the whole image in its new state faster but rendered at a lower quality, while our approach offers high image quality throughout the whole rerendering process, but areas handled as less affected by the change get updated later than with the classic approach. This confirms my argumentation that our method scales in proportion to the spread of the magnitude of impact in the tile update order. When only a few tiles are affected heavily, they can be placed at the top of the order and rerendered at high quality immediately. The user won't notice that the other tiles are updated later as they are not, or at least not heavily noticeably affected. On the other hand, the more uniformly and more tiles are affected by an interaction the less effective our approach becomes compared to classic path tracing, which rerenders the whole image indiscriminately.
One definitive advantage of our method over classic path-tracing is that there is no grain like noise present at any point during the rerendering. So if a user sees this noise as highly irritating our approach definitely seems favourable.

5.3 Further work

For the sake of simplicity and keeping the scope of this work manageable, some compromises had to be made. I will list some points of this project that could be improved upon here. For one thing, I explained in the related work section, that image reconstruction techniques often get combined with adaptive sampling to achieve better results and performance nowadays. We, however, chose to not use any particular of these algorithms. The performance of our method probably could be improved by implementing an image reconstruction algorithm into it.

Another point is our strategy to determine the affection rate of a region within a scene by a given user interaction. We simply assume that regions spatially closer to the interaction are affected more than those more apart. This assumption is likely wrong for some interactions, for instance moving a light source could affect a strongly reflecting surface spatially relatively far from the actual position perceivable more than the air around the light-source. For this reason our idea could be improved by designing a metric to more accurately classify the affection rate of a region by a given user interaction, in order to change the order our incremental rendering algorithm then can process these regions in. As part of this, regions that have not been affected at all could be identified and get completely omitted from the update queue, preventing them getting rerendered in a quality that is likely lower than the accumulation up to the point of rerender. It would even be thinkable to use different metrics for different interactions.

The "point of change" as I have named it, also is up for discussion. Defining a single pixel as the epicenter of change triggered by a user interaction is rather difficult. Our approach of simply using the last selected pixel, while very practical, is suboptimal for a wide variety of use cases. For example, a user rotates a relatively large object he or she has selected by clicking on one of the outer corners. With our strategy of rendering outwards from the PoC this pixel would be suboptimal since opposing sides of the object would get updated rather late. A more suitable point for this example would maybe be the center of the objects 2D bounding box. This problem could possibly be counteracted by a better affection rate metric and updating order, but a combination of a better PoC selection and order of impact would probably achieve the best results.

Finally less conceptual improvements of our project would be to support a wider range of user interactions, like material changes or deformations, a better user interface/interaction system or incorporating our idea into an existing software with a good user interaction experience like Blender or Maya.

CHAPTER 6

Conclusion

In this work I presented a novel approach to rerendering scenes after an user interaction within a real-time ray-tracing context for 3D editing. It builds upon the basic idea, that most user interactions only affect limited parts of the scene noticeably to a viewer. Our technique incrementally renders the image after an interaction, going from most to least affected image region. We implemented our algorithm into NVIDIA's Falcor rendering framework alongside basic editing functionality. This framework is light-weight software built with modularity and expandability in mind. It is designed for research projects such as this one, but it doesn't offer a very sophisticated user interface nor any editing capability by default. As a result this initial implementation of our idea is more a proof of concept rather than a finished product.
I evaluated our approach against classic full image path-tracing and confirmed that our method eliminates the typical grain like noise. In addition it works increasingly well the more localized an interaction affects the scene. However, I also recognised that our method doesn't necessarily always work better than full image rerendering. If significant amounts of a total image are heavily affected by a change, our incremental approach yields diminishing results. A classic full image path traced rerendering approach can arguably deliver better results in shorter time in some cases. How well our method performs for such, less optimal scenarios, depends on the chosen parameters of global sample rate, the NoR for the incremental rerendering, the tile size, the metric to determine the magnitude of effect an interaction has on a given image region and what pixel is choosen as PoC. In the limited time and scope we had to work on this thesis, we achieved a proof of concept but definitely not the limit of how far this approach could be optimized through careful consideration and tests of these parameters.
In conclusion, I would say that we were able to prove the potential of this method and that further work in this direction is warranted. If the method's parameters could be further optimized and if it could be combined with other existing state of the art algorithms like modern image reconstruction methods, implemented into a sophisticated

3D editing software like Blender or Maya, it could offer a new level of uninterrupted 3D editing workflow.

List of Figures

List of Tables

Bibliography

[BEEM15] Pablo Bauszat, Martin Eisemann, Elmar Eisemann, and Marcus Magnor. General and robust error estimation and reconstruction for monte carlo rendering. In *Computer Graphics Forum*, volume 34, pages 597–608. Wiley Online Library, 2015.

[BM98] Mark R Bolin and Gary W Meyer. A perceptually based adaptive sampling algorithm. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, pages 299–309, 1998.

[BVM+17] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony Derose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. *ACM Trans. Graph.*, 36(4):97–1, 2017.

[BYC+20] Nir Benty, Kai-Hwa Yao, Petrik Clarberg, Lucy Chen, Simon Kallweit, Tim Foley, Matthew Oakes, Conor Lavelle, and Chris Wyman. The Falcor rendering framework, 08 2020. `https://github.com/NVIDIAGameWorks/Falcor`.

[CDF92] Albert Cohen, Ingrid Daubechies, and J-C Feauveau. Biorthogonal bases of compactly supported wavelets. *Communications on pure and applied mathematics*, 45(5):485–560, 1992.

[CJAMJ05] Petrik Clarberg, Wojciech Jarosz, Tomas Akenine-Möller, and Henrik Wann Jensen. Wavelet importance sampling: Efficiently evaluating products of complex functions. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, page 1166–1175, New York, NY, USA, 2005. Association for Computing Machinery.

[CKS+17a] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Trans. Graph.*, 36(4), jul 2017.

[CKS+17b] Chakravarty R Alla Chaitanya, Anton S Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. *ACM Transactions on Graphics (TOG)*, 36(4):1–12, 2017.

[Com18] Blender Online Community. *Blender - a 3D modelling and rendering package*. Blender Foundation, Stichting Blender Foundation, Amsterdam, 2018.

[DHS+05] Frédo Durand, Nicolas Holzschuch, Cyril Soler, Eric Chan, and François X. Sillion. A frequency analysis of light transport. *ACM Trans. Graph.*, 24(3):1115–1126, July 2005.

[DWR10] Christopher DeCoro, Tim Weyrich, and Szymon Rusinkiewicz. Density-based outlier rejection in monte carlo rendering. In *Computer Graphics Forum*, volume 29, pages 2119–2125. Wiley Online Library, 2010.

[ETH+09] Kevin Egan, Yu-Ting Tseng, Nicolas Holzschuch, Frédo Durand, and Ravi Ramamoorthi. Frequency analysis and sheared reconstruction for rendering motion blur. In *ACM SIGGRAPH 2009 Papers*, SIGGRAPH '09, New York, NY, USA, 2009. Association for Computing Machinery.

[HJW+08] Toshiya Hachisuka, Wojciech Jarosz, Richard Peter Weistroffer, Kevin Dale, Greg Humphreys, Matthias Zwicker, and Henrik Wann Jensen. Multidimensional adaptive sampling and reconstruction for ray tracing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, New York, NY, USA, 2008. Association for Computing Machinery.

[HMS+20] Jon Hasselgren, Jacob Munkberg, Marco Salvi, Anjul Patney, and Aaron Lefohn. Neural temporal adaptive sampling and denoising. In *Computer Graphics Forum*, volume 39, pages 147–155. Wiley Online Library, 2020.

[JC95] Henrik Wann Jensen and Niels Jørgen Christensen. Optimizing path tracing using noise reduction filters. 1995.

[KA91] David Kirk and James Arvo. Unbiased sampling techniques for image synthesis. *SIGGRAPH Comput. Graph.*, 25(4):153–156, July 1991.

[Kaj86] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986.

[KBS15] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. A machine learning approach for filtering monte carlo noise. *ACM Trans. Graph.*, 34(4):122–1, 2015.

[KKR18] Alexandr Kuznetsov, Nima Khademi Kalantari, and Ravi Ramamoorthi. Deep adaptive sampling for low sample count rendering. In *Computer Graphics Forum*, volume 37, pages 35–44. Wiley Online Library, 2018.

[KS13] Nima Khademi Kalantari and Pradeep Sen. Removing the noise in monte carlo rendering with general image denoising algorithms. In *Computer Graphics Forum*, volume 32, pages 93–102. Wiley Online Library, 2013.

[KSKAC02] Csaba Kelemen, László Szirmay-Kalos, György Antal, and Ferenc Csonka. Simple and robust mutation strategy for the metropolis light transport algorithm. eurographics 2002/g. drettakis and h. *P. Seidel*, 21(3), 2002.

[LR90] Mark E Lee and Richard A Redner. A note on the use of nonlinear filtering in computer graphics. *IEEE Computer Graphics and Applications*, 10(3):23–29, 1990.

[LRR04] Jason Lawrence, Szymon Rusinkiewicz, and Ravi Ramamoorthi. Efficient brdf importance sampling using a factored representation. *ACM Trans. Graph.*, 23(3):496–505, August 2004.

[LWC12] Tzu-Mao Li, Yu-Ting Wu, and Yung-Yu Chuang. Sure-based optimization for adaptive sampling and reconstruction. *ACM Trans. Graph.*, 31(6), November 2012.

[McC99] Michael D McCool. Anisotropic diffusion for monte carlo noise reduction. *ACM Transactions on Graphics (TOG)*, 18(2):171–194, 1999.

[MCY14] Bochang Moon, Nathan Carr, and Sung-Eui Yoon. Adaptive rendering based on weighted local regression. *ACM Trans. Graph.*, 33(5), September 2014.

[MIGYM15] Bochang Moon, Jose A. Iglesias-Guitian, Sung-Eui Yoon, and Kenny Mitchell. Adaptive rendering with linear predictions. *ACM Trans. Graph.*, 34(4), July 2015.

[Mit87] Don P Mitchell. Generating antialiased images at low sampling densities. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 65–72, 1987.

[Mit90] Don P Mitchell. The antialiasing problem in ray tracing. *Advanced Topics in Ray Tracing*, 1990.

[Mit91] Don P. Mitchell. Spectrally optimal sampling for distribution ray tracing. In *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '91, page 157–164, New York, NY, USA, 1991. Association for Computing Machinery.

[MMMG16] Bochang Moon, Steven McDonagh, Kenny Mitchell, and Markus Gross. Adaptive polynomial rendering. *ACM Transactions on Graphics (TOG)*, 35(4):1–10, 2016.

[ODR09] Ryan Overbeck, Craig Donner, and Ravi Ramamoorthi. Adaptive wavelet rendering. *ACM Transactions on Graphics (SIGGRAPH ASIA 09)*, 28(5), December 2009.

[PBP11] Anthony Pajot, Loïc Barthe, and Mathias Paulin. Sample-space bright spots removal using density estimation. In *Graphics Interface*, pages 159–166, 2011.

[RKZ11] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive sampling and reconstruction using greedy error minimization. *ACM Trans. Graph.*, 30(6):1–12, December 2011.

[RKZ12] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive rendering with non-local means filtering. *ACM Transactions on Graphics (TOG)*, 31(6):1–11, 2012.

[RMZ13] Fabrice Rousselle, Marco Manzi, and Matthias Zwicker. Robust denoising using feature and color information. In *Computer Graphics Forum*, volume 32, pages 121–130. Wiley Online Library, 2013.

[RW94] Holly E Rushmeier and Gregory J Ward. Energy preserving non-linear filters. In *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 131–138, 1994.

[Sch13] Karsten Schwenk. *Filtering techniques for low-noise previews of interactive stochastic ray tracing*. PhD thesis, Technische Universität Darmstadt, 2013.

[Sch15] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[Sha49] Claude Elwood Shannon. Communication in the presence of noise. *Proceedings of the IRE*, 37(1):10–21, 1949.

[SN96] Gilbert Strang and Truong Nguyen. *Wavelets and filter banks*. SIAM, 1996.

[SSD+09] Cyril Soler, Kartic Subr, Frédo Durand, Nicolas Holzschuch, and François Sillion. Fourier depth of field. *ACM Trans. Graph.*, 28(2), May 2009.

[SW00] Frank Suykens and Yves D Willems. Adaptive filtering for progressive monte carlo image rendering. 2000.

[TM98] Carlo Tomasi and Roberto Manduchi. Bilateral filtering for gray and color images. In *Sixth international conference on computer vision (IEEE Cat. No. 98CH36271)*, pages 839–846. IEEE, 1998.

[VG97] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76, 1997.

[WABG06] Bruce Walter, Adam Arbree, Kavita Bala, and Donald P. Greenberg. Multidimensional lightcuts. In *ACM SIGGRAPH 2006 Papers*, SIGGRAPH '06, page 1081–1088, New York, NY, USA, 2006. Association for Computing Machinery.

[WFA+05] Bruce Walter, Sebastian Fernandez, Adam Arbree, Kavita Bala, Michael Donikian, and Donald P. Greenberg. Lightcuts: A scalable approach to illumination. In *ACM SIGGRAPH 2005 Papers*, SIGGRAPH '05, page 1098–1107, New York, NY, USA, 2005. Association for Computing Machinery.

[Whi05] Turner Whitted. An improved illumination model for shaded display. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, page 4–es, New York, NY, USA, 2005. Association for Computing Machinery.

[WRV98] Joachim Weickert, BM Ter Haar Romeny, and Max A Viergever. Efficient and reliable schemes for nonlinear diffusion filtering. *IEEE transactions on image processing*, 7(3):398–410, 1998.

[XP05] Ruifeng Xu and Sumanta N Pattanaik. A novel monte carlo noise reduction operator. *IEEE Computer Graphics and Applications*, 25(2):31–35, 2005.