Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Graphics and Interaction

Progressive spatiotemporal variance-based path tracing filtering

Bc. Jan Dundr

Supervisor: doc. Ing. Jiří Bittner, Ph.D.
Field of study: Computer graphics
May 2018

ČVUT
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Dundr** Jméno: **Jan** Osobní číslo: **406761**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačové grafiky a interakce**

Studijní program: **Otevřená informatika**

Studijní obor: **Počítačová grafika a interakce**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Filtrování globálního osvětlení pro interaktivní aplikace

Název diplomové práce anglicky:

Global Illumination Filtering for Interactive Applications

Pokyny pro vypracování:

Prostudujte existující metody pro filtrování globálního osvětlení použitelné v interaktivních aplikacích a vytvořte rešerši těchto metod. Následně implementujte nedávno publikovanou metodu pro filtrování globálního osvětlení využívající odhadu variance a reprojekce [1]. Implementaci realizujte v systému Virtual Reality Universal Toolkit (VRUT) jako rozšíření stávající implementace metody sledování cest. Proveďte důkladné testování implementace na nejméně dvou testovacích scénách. Zaměřte se na rychlost algoritmu, schopnost odstranění stochastického šumu a vizuální artefakty způsobené filtrováním.

Seznam doporučené literatury:

[1] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi. Spatiotemporal variance-guided filtering: real-time reconstruction for path-traced global illumination. In Proceedings of High Performance Graphics (HPG '17).

[2] Mathias Zwicker, Wojciech Jarosz, Jaakko Lehtinen, Bochang Moon, Ravi Ramamoorthi, Fabrice Rousselle, Pradeep Sen, Cyril Soler, and S-E Yoon. 2015. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. Computer Graphics Forum 34, 2 (2015), 667?681.

[3] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz. 2012. The State of the Art in Interactive Global Illumination. Comput. Graph. Forum 31, 1 (February 2012), 160-188.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Jiří Bittner, Ph.D., Katedra počítačové grafiky a interakce

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **15.02.2018** Termín odevzdání diplomové práce: ____________

Platnost zadání diplomové práce: **30.09.2019**

doc. Ing. Jiří Bittner, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Ing. Pavel Ripka, CSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

25. 5. 2018
Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

Děkuji velice Jiřímu Bittnerovi i Antonínu Míškovi za skvělé rady a porady a za možnost vyzkooušet filtr na VRUTu. Vřelé díky také patří rodině a všem spřízněným duším v okolí, bez Vás bych opravdu nic nenapsal! :)

Dodatečně také děkuji pánům Marko Dabrovic a Frank Meinl za model Sponzy, uživateli "Leo" z `https://archive3d.net` za model židle, který jsem vložil do Sponzy, neznámému studentovi, opět Tondovi Míškovi a Matějovi Vydrovi za 3D scénu křižovatky, na které pořád a neustále testuji (jsou to tu ty obrázky s autem) a nakonec také Standovi Štěrbovi za naměření rychlosti na své bleskové NVidii 1060.

Prostě díky!

Declaration

Prohlašuji, že jsem předloženou práci vypracoval samostatně, že jsem uvedl veškerou použitou literaturu a že jsem na sebe při psaní vylil kafe jen dvakrát.

V Praze, 28. května 2018

Abstract

Rendering 3D scenes with both fast and accurate global illumination is still a very tough problem: there is a large number of methods giving approximate results, each one with different compromise. Path tracing is, given it's accurate and unbiased nature, often the method of choice here. It is still, however, very hard to do in real-time due to its complex recursive light interactions: even the newest GPUs struggle to keep real-time frame rates for more than just a few samples per pixel, which can lead to a very noisy output or even no useful data for some more problematic areas.

A recent approach to processing the resulting image is described and experimented with in this thesis: diffuse light samples are accumulated from previous frames using reprojection and subsequently filtered by a fast bilateral filter constrained by normals, depth, and variance (which will stop blurring valuable details). This results in a temporally stable noise-free output which converges in a few frames.

We implemented the method using OpenGL and incorporated it in an existing high-performance CPU-based path tracer VRUT. We extended the approach by putting it in the progressive rendering framework, where initially less than one sample per pixel is shot to increase interactivity. We evaluate the performance and visual quality of this algorithm in several test cases with diffuse illumination.

Keywords: computer graphics, path tracing, global illumination, real-time rendering, bilateral filtering, temporal filtering, OpenGL

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

Abstrakt

Vykreslování 3D scén se zároveň rychle vypočítatelným a přesným globálním osvětlením je ještě pořád komplikovaný problém: existuje mnoho metod, které jsou schopny výsledek v určitých situacích odhadnout velmi přesně a v jiných zase selhat. Sledování cest (path tracing) je díky své přesnosti a nevychýlenosti často používaná metoda, stále je ale příliš pomalá pro výpočet v reálném čase kvůli velkému množství komplikovaých interakcí paprsků se scénou: i nejnověší GPU mají problém udržet si rychlost výpočtu v reálném čase pro víc než jen několik snímků za sekundu. To znamená, že obrazový výstup je všude z velké míry zatížen šumem a místy ani nemusí obsahovat žádnou informaci.

V této práci je popsána a otestována jedna z novějších metod: vzorky difúzní složky světla jsou s pomocí reprojekce nashromážděny z minulých snímků a následně je na ně aplikován bilaterální filtr využívající normálu, hloubku a rozptyl v daném místě k zastavení rozmazání cenných detaiů např. kolem hran a okrajů stínů. Tímto způsobem dostaneme časově stabilní výstup bez šumu (který nám dokonverguje hned po několika snímcích).

Tudo metodu jsme implementovali pomocí OpenGL jako součást existujícího rychlého vykreslovacího systému na CPU postaveného na sledování cest. Metodu jsme upravili tak, aby si poradila i se vstupem, kde v každém pixelu ještě nemusí být přítomen vzorek z vystřeleného paprsku (kvůli udržení interaktivní snímkové frekvence). Nakonec jsme zhodnotili kvalitu a rychlost výstupu tohoto algoritmu.

Klíčová slova: počítačová grafika, sledování cest, globální osvětlení, vykreslování v reálném čase, bilaterální filtr, časový filtr, OpenGl

Contents

1 Introduction **1**

1.1 Summary 2

2 Theory **5**

2.1 Global Illumination 5

2.2 Methods simulating light transport 6

2.2.1 Path tracing 6

2.2.1.1 Different versions 7

2.2.2 Photon mapping 9

2.2.2.1 Instant radiosity 9

2.2.3 Finite elements 9

2.3 Methods in rasterization pipeline 10

2.3.1 Ambient light 10

2.3.2 Prebaked diffuse lighting 11

2.3.3 Light probes 11

2.3.4 Screen-space methods 11

2.3.5 Voxel methods 12

2.3.5.1 Voxel cone tracing 12

3 Related work **17**

4 Algorithm **19**

4.1 Basic algorithm 19

4.1.1 Quick summary 19

4.1.2 Input 19

4.1.3 Accumulation 20

4.1.3.1 Variance 22

4.1.4 Spatial filtering 24

4.1.4.1 Bilateral filter 25

4.1.4.2 Normal weighting function 25

4.1.4.3 Depth weighting function . 26

4.1.4.4 Luminance weighting function 27

4.1.4.5 Combining the weights . . . 28

4.1.4.6 Edge-Avoiding À-Trous Wavelets 28

4.1.5 Final bledning 32

4.1.6 Tuning the algorithm 33

4.2 Fewer samples per pixel 34

4.2.1 Missing normal and depth information 34

4.2.2 Overblurring from first wavelet level accumulation 35

4.2.3 Overblurring caused by too high ϵ in w_L 35

4.2.4 Reprojecting a fraction of samples per pixel 35

4.2.5 Handling occlusions with holes 36

4.2.6 Temporal AA 36

4.3 More samples per pixel 37

4.4 Spatial variance estimate 37

5 Implementation and results **47**

6 Conclusion **51**

6.1 Limitation: no scene or light movement 51

6.2 Limitation: diffuse component only 52

6.3 Limitation: no transparency 54

6.4 Discussion 55

A Bibliography **57**

Figures

1.1 Tennis for two was a first visual videogame running on an osciloscope (1957), image source: http://nr.news-republic.com . . . 2

1.2 Juggler was a first real-time raytracing demo running on Amiga (1987), image source: https://commons.wikimedia.org . 3

1.3 We can see how our filter (in red) fits into a path tracing pipeline as a post-processing filter: it works purely with image buffers as its input. The green parts are optional: additional normal, depth and albedo buffers might be generated by a rasterizer instead of coming straight from the path tracer and any kind of post-processing might be additionally performed on the output of our filter. 4

1.4 Before and after our algorithm is applied: the noise is gone! 4

2.1 The camera (eye) is looking at a point X on the surface with a normal N, the incoming vector between the eye and the point on the surface is $D0$. The incoming ray $D0$ can reflect in any direction around hemisphere H with a probability given by BRDF (see [Nic65]) at a given point. Two examples of possible outgoing directions are $D1$ and $D2$. 6

2.2 rendering equation is valid for every point on the surface: every surface affects all other surfaces 7

2.3 path tracing approximates rendering equation recursively: rays that hit a surface continue reflecting around the scene up to some depth, image source: http://blender3d.cz 7

2.4 bidirectional path tracing connects subpaths from camera and light and converges faster, image source: http://www.maw.dk 8

2.5 Metropolis light transport is able to converge in short time even very complex cases of light transport, image source: https://cg.tuwien.ac.at/ 8

2.6 Two cropped parts of path-traced Sponza images are shown. The first column shows a shadow under a chair formed by direct light from a small sphere-shaped light; the second column represents an area with indirect light. The four rows represent images after $1, 4, 16, 64$ samples. The image is still noisy even after 64 samples and many features in indirect lighting are missing before that point. Most GPUs are only capable of rendering around 1 sample per pixel in real-time today. 13

2.7 PM has two passes: photon distribution trough the scene and querying photon map for photon density 14

2.8 pixel brightness in simple lighting in rasterization pipeline is calculated as cosine of angle between light's direction and surface normal clamped at zero 14

2.9 First SSAO implementation as it appeared in Crysis. Note the middle gray areas which shouldn't get occluded: the hemisphere visibility term wasn't implemented correctly, halos around borders and distance scaling. More modern SSAO implementations don't have these issues. Image source: https://en.wikipedia.org 15

2.10 Voxel cone tracing rasterizes the scene into a mipmapped 3D voxel grid, image source: http://leifnode.com 15

3.1 Architecture overview of a Deep Recurrent Neural Network used by Chaitanya et al. Image source: [CKS+17] 18

3.2 Screenshot from NVidia's realtime raytracing Volta demo Reflections. Image source: NVidia, Youtube... 18

4.1 This diagram illustrates the inner workings of the algorithm (it zooms out on the red "filter" part of the diagram 1.3): the blue numbers relate to the three stages described in the section 4.1.1................. 20

4.2 Additional input buffers, together forming something similar to a GBuffer. They are normal, depth (in top row) and albedo buffer (depth buffer has been scaled for preview). Normal buffer is packed into 2-component form using a spheremap transform (discussed later in section 4.2.1) to allow both linear interpolation and memory-friendly storage of normalized normals. Additional optical flow or in-pixel variance buffers might be required, depending on the version of the algorithm. 21

4.3 A single frame of our one-sample-per-pixel input. The scene is lit by a singe sphehe-shaped light source. Shadow of the chair formed by a direct lighting is noisy, but discernible, the soft indirect bounces on the lion and in the back are, however, only very sparsely sampled....................... 22

4.4 The same scene as in 4.3 after a few frames of accumulation: new detail emerges, but there's still a lot of noise. The noise would start to disappear only after seconds into the accumulation (as we saw in figure 2.6). 23

4.5 Some pixels weren't visible in previous frame and it isn't possible to blend them with a prpevious frame. This creates visible artifacts in the accumulation: there are some places along edges with visibly less accumulated samples. This additional noise gets filtered by our spatial bilateral filter during the next algorithm stage 4.1.4. 24

4.6 The accumulated scene is filtered by our bilateral filter with normal and depth weights driven by normal and depth buffers pictured below the image. We can see the geometry-based edges have been preserved, but all important structure introduced by path tracing has been lost by too much blurring (like the chair's shadow).......... 26

4.7 Variance buffer: lighter color represents more variance at the currect pixel. Variance highly correates with luminence, but differs in important places: the floor is very bright, but with smaller variance (most rays simply hit the light) and shadow edges have higher variance than their surroundings because secondary rays at their pixels ended up in more different places. More noise is present in these images compared to accumulated diffuse because variance is calculated using terms with a square power, amplifying all errors. 27

4.8 The result after applying our bilateral filter with weights calculated using normal, depth and variance buffers (pictured below the image). Our bilateral filter blurs more values of high variance together (such as the shadow borders) resulting in sharp shadows. 29

4.9 Illustration of multi-scale convolution used for accelerating the bilateral filtering: a wide convolution kernel is approximated by multiple smaller, faster convolutions in exponentially decreasing skip size ($2^2 - 1, 2^1 - 1, 2^0 - 1 = 3, 1, 0$ pictured). 30

4.10 A few stages from bilateral filtering of a car model. Top left image is unfiltered, next three in reading order are three stages from our 6-stage bilateral filter. The scales are applied in a decreasing order to subsequently filter high-frequency artifacts introduced in a past low-frequency stage. These artifacts can be seen as "boxy" patterns in the noise. It can happen the pixel is filtered only by earlier stages and the artifact patern appears in the output: this is, however, noticeable only at very low sample counts and right after disocclusions, when one sample distributes too far. 31

4.11 The hierarchical wavelet filter was applied in increasing order of scales (high-frequency first, low frequency later). The resulting artifacts are clearly visible as repeated borders along edges. 32

4.12 Default accumulation buffer is shown on the top and the accumulation buffer with first-wavelet previous frame lookup on the bottom. The bottom one certainly has much less noise and speeds up convergence this way. It, however, increases bias as we can see in form of a slightly blurred shadow edges in the magnified area around the chair's legs. The bright speckles are new, yet un-spatially-filtered samples: technique using less samples per pixel was used (more about this version of the algorithm is described in section 4.2). 39

4.13 Variance buffer after being filtered by bilateral filter's weights two times. Only prominent areas are weighted significantly at this stage: only pixels with high variance are allowed to be filtered in later scales of the bilateral filtering to prevent overblurring (the variance weighting is scale-awave). 40

4.14 Temporal antialiasing smooths both jagged lines (easily visible with complex geometry) and noisy low-sampled areas (like the han rail here). Top image is without temporal antialiasing, bottom one with it. . . 41

4.15 Accumulation ratio r is important for detail preservation. Top image is rendered using 16 samples/pixel with our filter off, second one using unbounded r and the third one with $age = 5$. The shadow's shape is correct even in the third row, but penumbra detail has been lost due to too aggresive blending. The missing slight brown tint in the middle picture is caused by changing material properties. 42

4.16 The result after the final blending phase: albedo is multiplied back in and themporal antialiasing is applied by blending with a previous frame. 43

4.17 This diagram illustrates the changes (highlighted in red) we need to make to our algorithm to support even less than one sample per pixel as input. All modifications are discussed in section 4.2. 43

4.18 GLSL code implementing encoding and decoding normal between its normalized 3-component and packed 2-component form. Code source: 44

4.19 It can occur previous frame gets reprojected under new sparse samples that are actually closer to the camera and should cover the old ones, resulting in holes in objects cleaning up only after many new frames. Our solution is using a min-filter-based rejection described in section 4.2.5. 44

4.20 Illustration of a eroded depth buffer: all holes with incorrect surfaces (much further compared to the eroded depth buffer) get rejected. 45

6.1 This is what would the accumulation buffer look like without any reprojection applied (the camera is moving to the left). A similar smearing effect would be visible on shadows in a scenario a light or an object would move: uncorrelated pixels would get blended unless rejected using some other mechanism. 52

6.2 Implementing the algorithm and ignoring reflections yields incorrect resuls. Top image shows a correctly pathtraced car with a reflection, the bottom one the same scene when our filter is applied: reflection is overblurred and it has albedo's color instead of a reflected one. The path to the correct solution is described in section 6.2. 53

Chapter 1
Introduction

People were interested in computer graphics ever since there were computers (or at least very early on): having something visually nice and interactive on the screen was always very appealing. One of the first visually fully interactive experience appeared already in the 1950s: American physicist William Higinbotham developed a sports video game Tennis for two in 1958 (see figure 1.1). It ran on an oscilloscope as a fun attraction at an exhibition, and its circuitry took up space as big as a microwave oven.

The popularity of computer graphics gained a lot of momentum afterward: a lot of arcade games with better and better graphics appeared in the 70s and 80s, interactive computer graphics became mainstream with home computers. Ray tracing (like a juggler in figure 1.2), path tracing, rendering equation and other crucial 3D rendering concepts appeared at that time as well. GPUs with hardware Transform and Lighting accelerators were introduced in the late 90s later developing in vertex and pixel shaders, enabling a large number of graphics algorithms. They keep evolving to this day: we can render more and more elaborate lighting phenomena in real-time.

It's slowly narrowing, but there is still a significant quality gap between offline-rendered and real-time computer graphics. Shading models are becoming more physically accurate, but most production real-time rendering pipelines heavily depend on rasterization and a few additional manually-coded visual approximations (shadow maps, screen-space lighting, light probes, etc.). Offline rendering for visualization and movies, on the other hand, keeps using path tracing: its convergence gets better and better by using smarter probability distributions and accelerated tree structures, but the basic concept stayed the same for a long time. GPUs can be used to accelerate path tracing performance rapidly (mostly for coherent rays and scenes that fit in their video memory), but we are still not able to produce anything near noise-less images in real-time (at least 30 fps). Some approaches raytracing screen-space or mipmapped voxel data are emerging even in the rasterization world (raytracing reflections, shadows, GI, etc.), but their use case is either

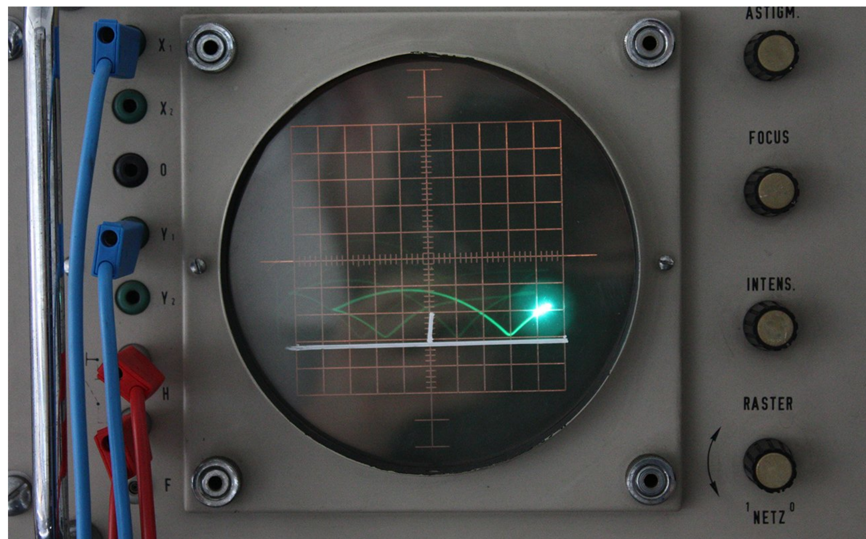

Figure 1.1: Tennis for two was a first visual videogame running on an osciloscope (1957), image source: `http://nr.news-republic.com`

very limited in scope compared to accurate path-traced result or too slow for current hardware.

Recently, another approach to tackling the real-time path tracing problem surfaced: using the quickly computed noisy data and approximate global illumination using only them, essentially rapidly denoising the image in some clever way. Different approaches like this include a machine learning approach ([CKS$^+$17]) using a pre-trained neural network, real-time indirect light approximation using light probes ([SL17]) and spatiotemporal filter ([SKW$^+$17]) with emphasis to sample variance experimented with in this thesis. We will talk about these works in more detail in chapter 3.

The filter is applied as a post-processing filter: it needs only a few image buffers generated by path tracing as an input and is in no way dependent on 3D scene complexity or exact path tracing implementation, as we can see in a diagram 1.3. The resulting rendering pipeline could, if functional, have advantages of both worlds: accurate (based on path tracing) and fast (only a small number of samples is rendered and quickly filtered each frame). The noise/detail ratio can be balanced using three simple parameters introduced in section 4.1.6.

1.1 Summary

The chapter Theory (2) is going to focus on fundamental concepts used in 3D global illumination rendering, some of which are input to this filter or are used as a part of it. Chapter Related work (3) goes through similar

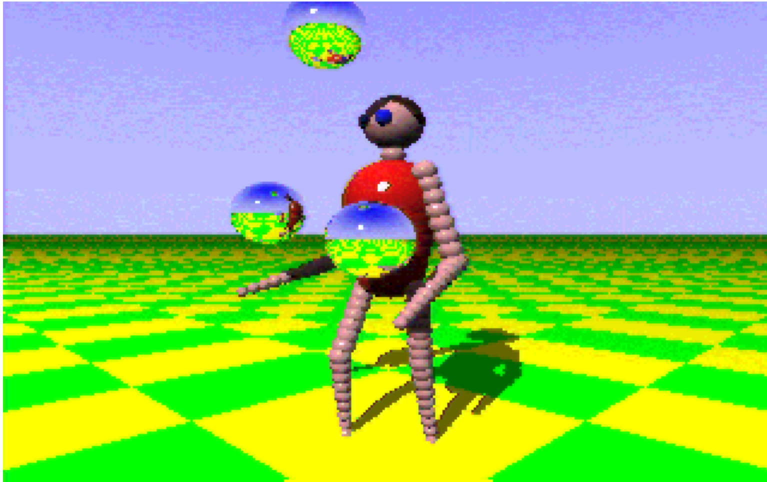

Figure 1.2: Juggler was a first real-time raytracing demo running on Amiga (1987), image source: `https://commons.wikimedia.org`

published approaches. Next, I will go through the elementary stages of the filter and later dive deeper into it in a chapter Algorithm (4). I'll talk about the implementation and summarize the algorithm's quality and speed in the Implementation and results (5). Finally, I will go through the filter's limitations and possible future extensions in the chapter Conclusion (6).

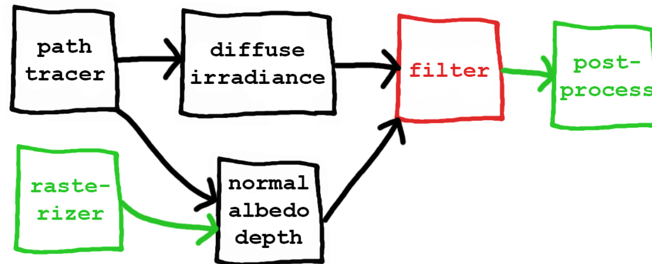

Figure 1.3: We can see how our filter (in red) fits into a path tracing pipeline as a post-processing filter: it works purely with image buffers as its input. The green parts are optional: additional normal, depth and albedo buffers might be generated by a rasterizer instead of coming straight from the path tracer and any kind of post-processing might be additionally performed on the output of our filter.

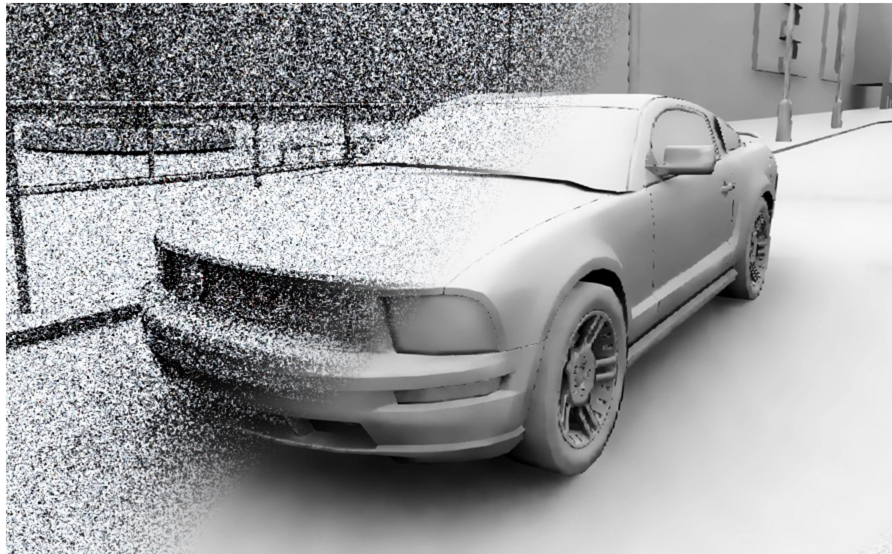

Figure 1.4: Before and after our algorithm is applied: the noise is gone!

Chapter 2

Theory

2.1 Global Illumination

The term Global Illumination ([RDGK12]) (GI) applies to any rendering technique that attempts to produce realistic looking images using not only direct interaction between light and a surface but also light propagating and bouncing through the whole scene. All of them follow the idea of rendering equation ([Kaj86]):

$$L_o(x, d) = L_e(x, d) + L_r(x, d)$$

It states that outgoing radiance L_o from location x in direction d is sum of emitted (L_e) and reflected (L_r) radiance. We can find the reflected radiance at our point by shooting a ray into the scene. The amount of light is then adjusted by a cosine of an angle between the ray and out surface normal (dot product) and by a special per-material function describing how much light is transferred depending on incoming and outgoing ray directions. More formally, reflected radiance is computed as:

$$L_r(x, d) = \int_h L_i(x, d_i) f_r(x, d_i, d) max(0, n \cdot d_i) dd_i$$

where h is hemisphere on a location x oriented by its normal n, L_i is incident radiance determined by shooting a ray into the scene and f_r is Bidirectional Reflectance Distribution Function (BRDF, see [Nic65]) determining the amount of light for different incoming and outcoming directions. Dot product describing angle between normal and incoming light direction is clamped at zero so that directions facing each other add up light and they never subtract it. We can see the illustrative situation on figure 2.1.

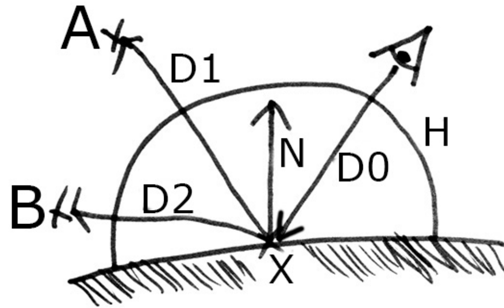

Figure 2.1: The camera (eye) is looking at a point X on the surface with a normal N, the incoming vector between the eye and the point on the surface is $D0$. The incoming ray $D0$ can reflect in any direction around hemisphere H with a probability given by BRDF (see [Nic65]) at a given point. Two examples of possible outgoing directions are $D1$ and $D2$.

This is not a straightforward or easy problem to solve. There are many possible locations in every 3D scene where rendering equation has to be valid, and they all depend on each other recursively, as illustrated in figure 2.2.

2.2 Methods simulating light transport

2.2.1 Path tracing

Path tracing ([Kaj86]) approximates integral from the rendering equation by Monte Carlo sampling. Primary rays are shot from camera origin through the screen, and all the surface's rendering equations are solved by randomly sampling rays through their hemispheres. A probability of a ray being shot in a certain direction respects the rendering equation with its cosine law and BRDF (importance sampling) and continues evaluating other surfaces it hits in the scene recursively, as seen on figure 2.3.

The solution is initially very noisy but converges to a correct solution after enough rays have been propagated through the scene. It's an unbiased method: the result it gives us is always exact with some amount of noise. Path tracing is a golden standard method for computing a high-quality GI, a significant advantage of this method is its generality and robustness: a lot of otherwise complicated light phenomena appear from the relationship of rendering equation and BRDF by themselves, there is no need to add them manually.

Figure 2.2: rendering equation is valid for every point on the surface: every surface affects all other surfaces

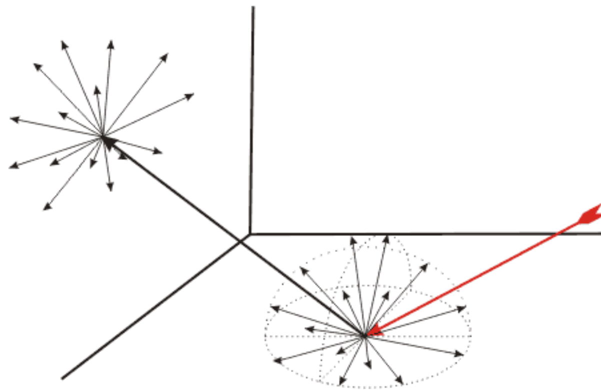

Figure 2.3: path tracing approximates rendering equation recursively: rays that hit a surface continue reflecting around the scene up to some depth, image source: `http://blender3d.cz`

2.2.1.1 Different versions

While the naive path tracing is correct in its essence, some lighting situations might need a very long time to converge due to the structure of the algorithm: sending rays from the screen and letting them bounce around randomly. A small, but bright source of light (like a light bulb or a sun (which we might indeed consider small by its angular size)) would have a very minuscule probability of being hit by an incoming ray. It would, however, have a large impact on an amount of reflected light: the output could get noisy to the point of not hitting the light at all for thousands of samples per pixel. Next event estimation solves this problem by sampling predetermined bright light sources more often: all known lights are usually tested for visibility at every bounce.

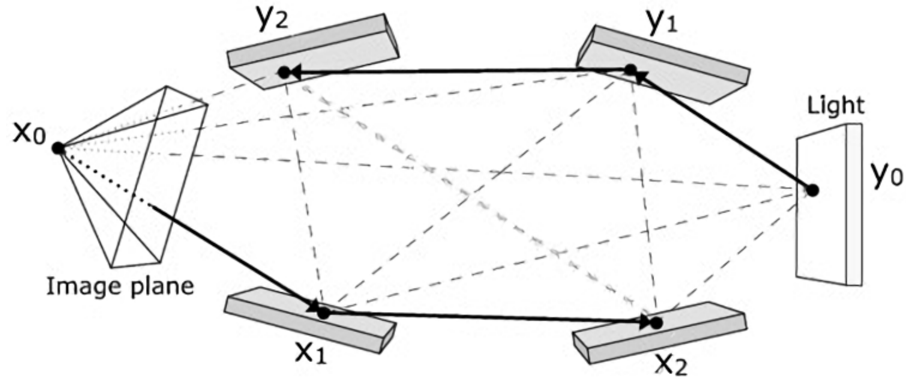

Figure 2.4: bidirectional path tracing connects subpaths from camera and light and converges faster, image source: `http://www.maw.dk`

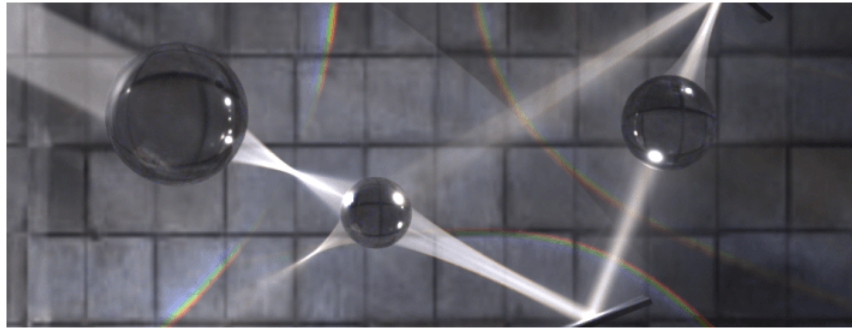

Figure 2.5: Metropolis light transport is able to converge in short time even very complex cases of light transport, image source: `https://cg.tuwien.ac.at/`

Caustics (bright spots light likes to make when traveling through a medium with a high refractive index like glass or water) converge very slowly as well because traditional path tracers trace the rays from the eye rather than from a light source. Bidirectional path tracing ([LW93]) is something like next event estimation on steroids: it traces a few light bounces both from a surface and light and connects each hit point on the surface with each other by shooting additional rays (as seen on figure 2.4). Both next level estimation and bidirectional path tracing need to be carefully weighted by probabilities the light will travel in the ray's direction: too much noise would be introduced into the output otherwise.

Many new methods for faster converging path tracing emerge all the time. There are probabilistic connections for bidirectional path tracing ([PRDD15]) reusing lighting subpaths and introducing caching. Metropolis light transport ([Vea98], see 2.5) converges quickly even with scenes with very hard to find light paths. Adaptive sampling based on the remaining variance (noise) in the image at a given pixel ([TJ97]) is possible, same as milking temporal

coherence ([HDMS03]) and many more. The performance of any ray tracing approach depends on the acceleration structure used: that's a whole another universe of problems we will not tackle in this thesis.

Any of the path tracing versions could be used as an input to our filter: we experimented with a simple next level estimation path tracer with sample order given by a Halton sequence to fill up space quickly and nicely.

While path tracing is beautiful, unbiased and widespread in offline rendering, it's still very slow for real-time usage today (how quickly a very simple path tracer converges can be seen in figure 2.6). It can take a lot of time to converge, depending on the exact algorithm used and scene complexity: seconds, minutes or hours. That's why a plethora of other GI solutions exist: each one is suitable for a specific GI feature but less for others.

2.2.2 Photon mapping

Photon mapping ([Jen96]) (PM) is similar to a bidirectional path tracing in its concept. It computes the result in two passes refsk2: a large number of photons is shot from the light sources, bounced around the scene and stored into a photon map at each hitpoint. Pixel brightness is then computed in a second pass by shooting one ray and estimating photon density using a quick search in a photon map. This approach can in some conditions get very close to real-time by using smart search structures (like KD-trees) and is an excellent solution for caustics simulation because of its direct "from light to scene" approach. It's biased, but the bias disappears with more samples used.

2.2.2.1 Instant radiosity

The first pass of instant radiosity ([Kel97]) is identical to photon mapping: but the second pass doesn't lookup photon density, but treats prominent photons as a virtual point light (VPL). The scene is then lit by these VPLs with imperfect shadow maps. These shadow maps are only imperfectly sampled and a very low resolution (32x32) so many of them can be processed at once.

2.2.3 Finite elements

Finite elements method by Goral et al. ([GTGB84]) is based on a basic idea from the rendering equation: every surface affects every other surface (triangles in a scene expressed by a mesh, for example). We can construct a linear system of equations representing these relationships and solve it by a least squares method.

Not every surface needs to be directly linked to every other: we can skip links for surfaces that are not directly visible. The resulting system is easier to solve, and light will propagate to them anyway. This method can only approximate smooth diffuse lighting; no sharp reflections are taken into account. Deep GBuffers can be used to implement this method in real-time on GPU: there is no linear system being solved there, light propagation is iterated temporally between frames.

2.3 Methods in rasterization pipeline

Real-time GPU accelerated rendering tackles the rendering problem differently from the methods we talked about until now: we're not shooting rays into the scene and the geometry is rendered directly: triangles are drawn on the screen by rasterization. GPU has special hardware just for this task and is capable of rasterizing millions of triangles in a very short time. Simple direct lighting model computation for a directional light can look like this (see figure 2.8):

$$val(P) = \sum_{l=1}^{L} max(0, N \cdot D)$$

where $val(P)$ is a resulting brightness of a pixel P. It's computed as a sum over all lights l from L lights as a dot product between triangle normal N and light direction D so that surface is wholly lit when its normal is precisely opposite to the light direction and not lit at all when perpendicular to it. The value is clamped to zero so that facing away from the light source doesn't subtract light. An additional shadow map might be pre-rendered and looked up to check if the light is visible at the current surface.

Implementing a practical GI approach in this constrained real-time environment has always been a challenge, the algorithms can be very simple or mightily complex.

2.3.1 Ambient light

Sometimes, an ambient light is sufficient. It (quite crudely) approximates GI by lighting every fragment by a constant value: there can never be an unlit place in a scene. This technique works best with textures: the contrast is lost otherwise.

2.3.2 Prebaked diffuse lighting

GI can be computed by any slow but accurate method beforehand and baked into the scene as a part of a texture or another texture channel ([LTH+13]). This approach supports only static light but enables quality lighting for virtually no additional cost, that's why it has been widely used.

2.3.3 Light probes

Light probes ([MMNL17]) can be stored at some important locations through the scene: they capture indirect lighting information that is later used to light objects by interpolating between the probes. It can be combined with a similar probe-using approach for reflection using environment maps. Probes can be placed by an artist or automatically, even generated on-the-fly. This approach is popular but suffers from poor spatial resolution.

Many-light approach presents a light probe extension using the same VPL algorithm as is described in the Instant radiosity section ([Kel97]).

2.3.4 Screen-space methods

Screen-space methods, mainly Screen Space Ambient Occlusion (SSAO) ([RGS09]) gained a massive popularity since it was used in Crysis (see figure 2.9) for the first time since it's a simple method with a good visual appearance. It was, however, very often overused and its effect exaggerated too much.

Ambient occlusion is a simple, yet nicely-looking approximation to true GI: the point on the surface is darker if it's occluded from incoming light by its surroundings. The screen-space variant works with a depth buffer and optionally a normal buffer: it searches a small neighborhood around current point and approximates occlusion of the current fragment by its surroundings. The fragment is darker when it has a lot of occluding pixels around it. SSAO is often sampled at half resolution and filtered in multiple passes to reduce noise since it's technically an approximate form of screen-space raytracing.

Many modifications of the basic methods were implemented during the years to mitigate disocclusion artifacts (even using two-layered GBuffer by Mara et al. [MMNL16]), increase directionality, bounce colors ([RGS09]) and even approximate accurate GI ([MMNL16]). It's, however, always limited to the on-screen information: any attempt to sample further points would result in a visible pop-in.

2.3.5 Voxel methods

GI is a very 3D spatial phenomena, so it makes sense that many voxel-based methods exist. They usually inject the direct lighting information in a voxel grid first, then propagate light through them in some way and then interpolate them to light their scene. This approach is quite fast using 3D texture lookups but, again, suffers from bad spatial resolution and light bleeding.

2.3.5.1 Voxel cone tracing

Cone tracing by Crassin et al. ([CNS+11]) is a very exciting real-time approach, similar to approximate single-bounce diffuse path tracing in the voxel space with a twist. The scene is first rendered with direct lighting in a voxel grid using rasterization, multiple levels of detail of this voxel grid are computed, and cones are shot from screen fragments into the voxel grid to determine indirect lighting. Cones are like rays, but they have an increasing width: voxel grid is sampled at the lower level of detail according to the cone's diameter at the hitpoint (as seen in figure 2.10). The voxel data are directional (6 directions for a voxel cube) to increase directional resolution.

This is a surprisingly good real-time approach to GI approximation and has been already implemented a lot of times in a gaming world already: as a part of Unreal Engine, for example. It, however, suffers from bleeding artifacts (no good solution for accurate visibility at lower voxel resolutions) and use a lot of memory for higher-resolution voxel data, which are necessary for more complex GI phenomena. More compact tree representation would solve the memory issue, but the lookup would become too slow.

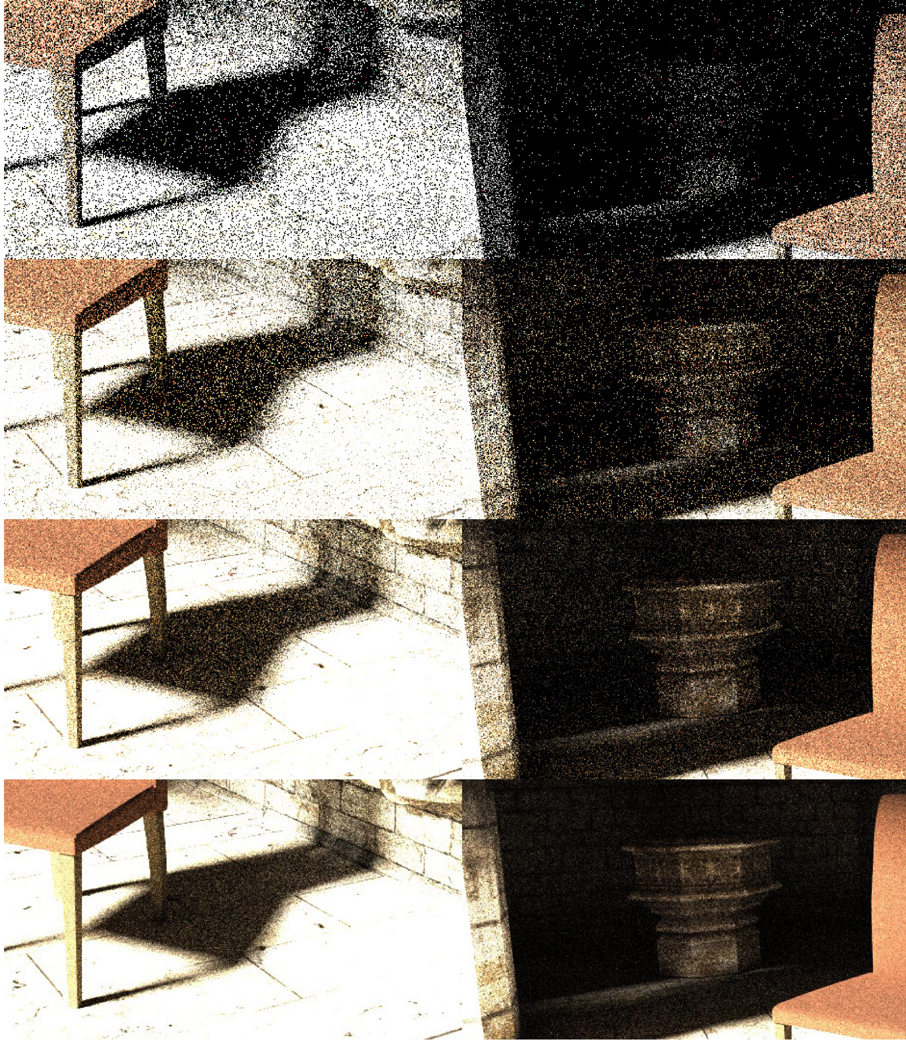

Figure 2.6: Two cropped parts of path-traced Sponza images are shown. The first column shows a shadow under a chair formed by direct light from a small sphere-shaped light; the second column represents an area with indirect light. The four rows represent images after $1, 4, 16, 64$ samples. The image is still noisy even after 64 samples and many features in indirect lighting are missing before that point. Most GPUs are only capable of rendering around 1 sample per pixel in real-time today.

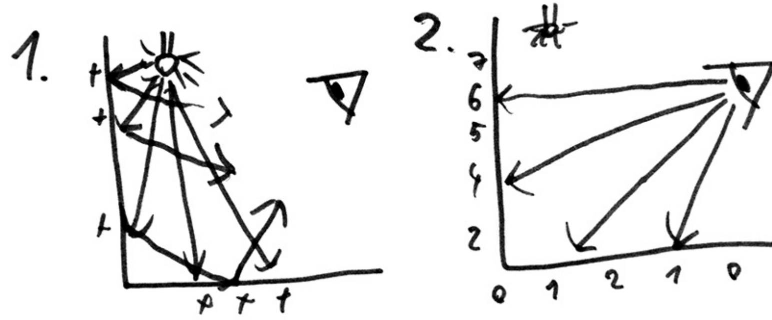

Figure 2.7: PM has two passes: photon distribution trough the scene and querying photon map for photon density

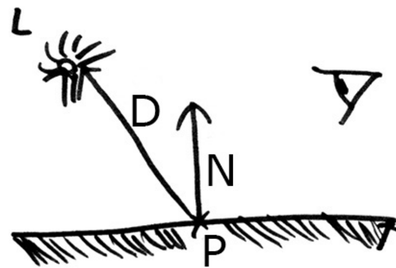

Figure 2.8: pixel brightness in simple lighting in rasterization pipeline is calculated as cosine of angle between light's direction and surface normal clamped at zero

Figure 2.9: First SSAO implementation as it appeared in Crysis. Note the middle gray areas which shouldn't get occluded: the hemisphere visibility term wasn't implemented correctly, halos around borders and distance scaling. More modern SSAO implementations don't have these issues. Image source: `https://en.wikipedia.org`

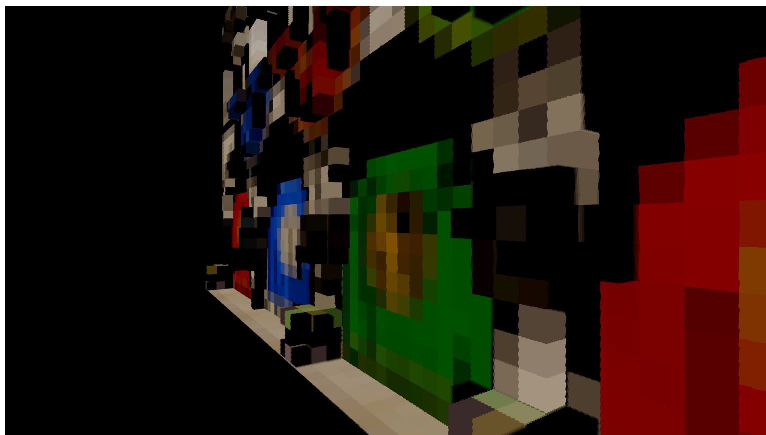

Figure 2.10: Voxel cone tracing rasterizes the scene into a mipmapped 3D voxel grid, image source: `http://leifnode.com`

Related work

Many methods using the low-sample quickly computed noisy data as a basis for a noiseless biased approximation surfaced lately.

This thesis is based on an approach introduced in a paper from Shied et al. ([SKW+17]) published in at HPC 2017: it filters path tracing output accumulated over multiple previous frames constrained by auxiliary buffers and sample variance. This thesis experiments with this approach, modifies some of its parts and extends it to be able to use any number of samples per pixel. The same technique is used by Frostbite to bake global illumination quickly.

Machine learning approach from Chaitanya et al. ([CKS+17]) solves the same problem, but using a different tool: machine learning, more accurately a recurrent denoising autoencoder. A recurrent deep neural network (see 3.1) is trained on flythroughs of three complex scenes to take a sequence of noisy one-sample per pixel data and return a filtered result. The most significant advantage of this method is the reduced warm-up time: the complex model can extract more of structural information from the noisy data early on, but our filter needs to accumulate the structure of the noise through its variance during the first 2-3 frames. This approach is used in Optix, using new NVidia's tensor cores Volta hardware and some form of AI denoising is used in Octane renderer as well.

Our filter, however, runs much faster (5): just a few milliseconds on any GPU compared to 54 ms on Titan Pascal (both at 720p). The execution could be, however, faster and separate from other tasks using new NVidia's tensor cores: it would, however, take up space from other operations on the tensor cores. Moreover, our filter scales to any number of samples per pixel easily without any retraining. Some work exhaustively comparing performance/quality tradeoffs of these two methods could be very interesting.

Another paper ([MMBJ17]) with seemingly comparable results by Mara uses very similar methods. Their filtering is, however, limited to the use of

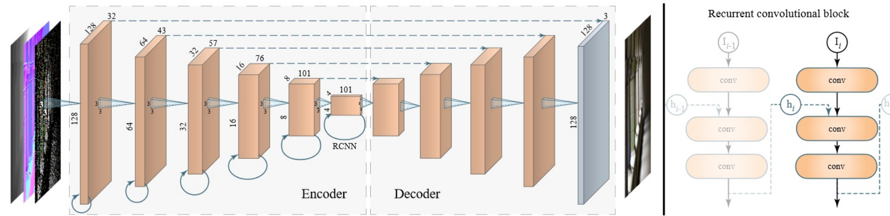

Figure 3.1: Architecture overview of a Deep Recurrent Neural Network used by Chaitanya et al. Image source: [CKS+17]

Figure 3.2: Screenshot from NVidia's realtime raytracing Volta demo Reflections. Image source: NVidia, Youtube

a bilateral filter without using any variance: this might hurt the method's performance. A study comparing our algorithm with Mara's would be very interesting as well.

Another recent intriguing paper from Silvennoinen and Lehtinen ([SL17]) uses irradiance probes placed into a scene to approximate diffuse indirect lighting by using very accurate mutual visibility approximations from sparse data. Results are very similar to accurate path-traced images and available from the first frame, where our filter needs some warm-up time. It's, however, constrained only to irradiance probes and it's unclear how they would behave under animation.

Some form of similar denoising is advertised as a part of upcoming NVidia's DXR real-time raytracing framework (for example 3.2), it's unclear what technique is used exactly. A very similar method is very likely used in an AMD ProRender denoiser.

Chapter 4

Algorithm

4.1 Basic algorithm

4.1.1 Quick summary

The algorithm can be divided into three main stages (their relationships are illustrated by diagram in figure 4.1):

1. Firstly, our filter accumulates the diffuse irradiance data. The current frame is blended with a previous accumulation output reprojected using our velocity buffer: some part might be left unblended because of disocclusions. Variance is accumulated and blurred at this stage as well. This process is described in section 4.1.3.

2. Next, a bilateral filter is applied to the accumulated result. Its weight is driven by normal, depth, material id, and variance buffers to prevent blurring over important image features. We go trough the workings of this filter in section 4.1.4.

3. Finally, albedo modulation and tonemapping is applied. The resulting frame is blended with a reprojected previous final one once again: this introduces some additional noise filtering and temporal antialiasing. This operation is described in section 4.1.5.

4.1.2 Input

Let's assume the basic version of the algorithm, which works with an incoming input sequence of new images. We have frames of 1 sample/pixel of diffuse irradiance as an input, as we can see on figure 4.3 (only a diffuse lighting component will be filtered to simplify our reprojection). We have additional

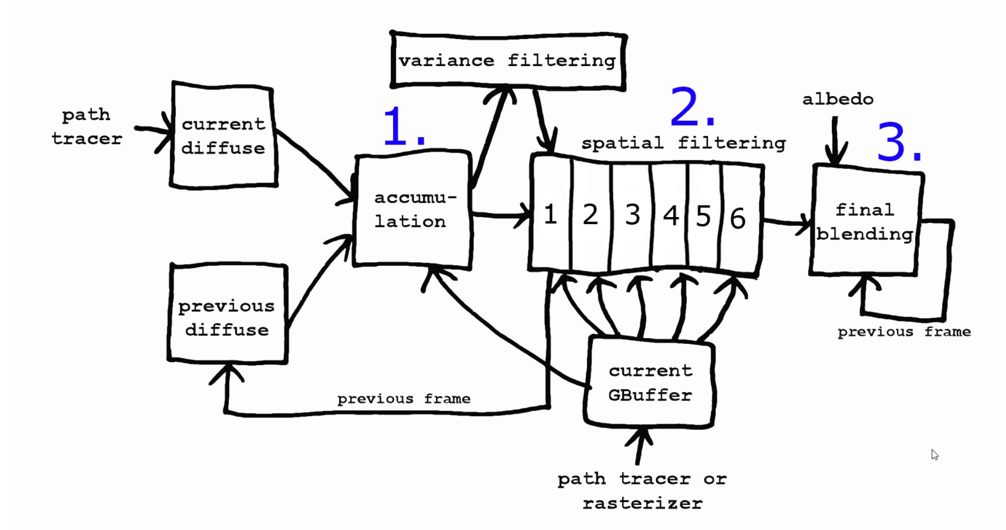

Figure 4.1: This diagram illustrates the inner workings of the algorithm (it zooms out on the red "filter" part of the diagram 1.3): the blue numbers relate to the three stages described in the section 4.1.1.

image buffers to constrain our filtering, similar to a GBuffer in a deferred rendering (see [ST90]): normal, depth and material id. Albedo buffer is used to modulate our filtered diffuse lighting by color information. The buffers are illustrated in a figure 4.2.

To summarize, we have these image-sized buffers as an input to our algorithm: diffuse irradiance, albedo, material id, normal, depth and velocity buffer.

All of these can be supplied by our path tracer, but it's possible to use a GPU-accelerated rasterizer take of all primary rays and output all the additional image buffers. Computing primary rays by rasterization should be a lot faster than raytracing in most typical cases; it can, however, become slower for very complex scenes. And sometimes, the implementation cost of the additional rasterizer on top of your path tracing pipeline might be too much work.

4.1.3 Accumulation

Current light buffer (path tracing output) is blended with the one from the previous frame. The previous frame is reprojected into current one using velocity buffer, which records optical flow between two frames: it's a screen-space offset between last frame and current frame fragment positions. Alternatively, one can use only current and previous frame's matrices to the pixel offset if there are no dynamic objects. A fragment is discarded on disocclusion, this is determined by a difference between predicted reprojected

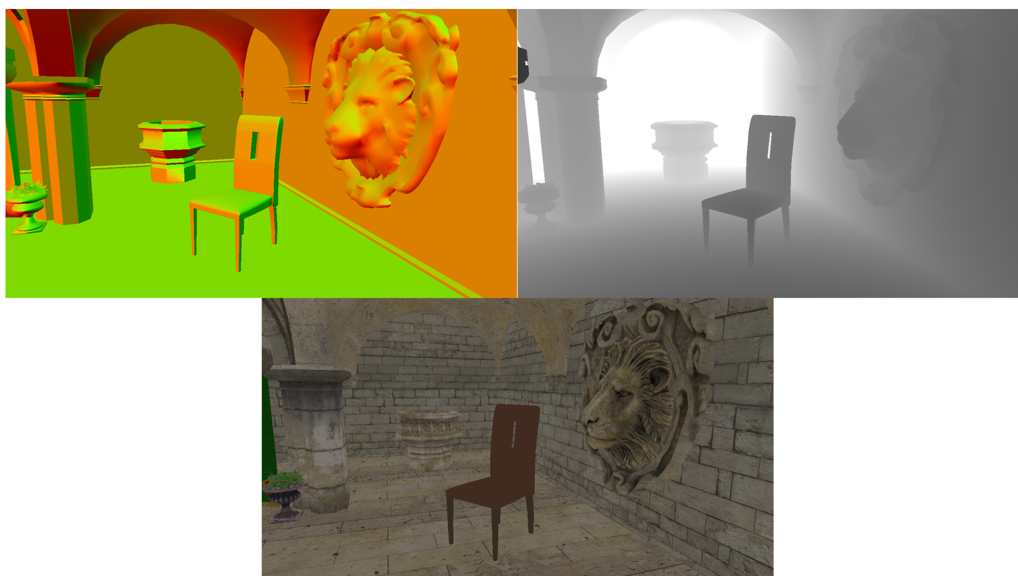

Figure 4.2: Additional input buffers, together forming something similar to a GBuffer. They are normal, depth (in top row) and albedo buffer (depth buffer has been scaled for preview). Normal buffer is packed into 2-component form using a spheremap transform (discussed later in section 4.2.1) to allow both linear interpolation and memory-friendly storage of normalized normals. Additional optical flow or in-pixel variance buffers might be required, depending on the version of the algorithm.

depth and actual depth in depth buffer. Only current frame light data is accumulated in a case of a disocclusion. Otherwise, the formula is:

$$c = cr + p(1 - r)$$

where c is a current frame and p is a previous one. We can control how much of current and previous frame gets blended by a ratio r in a range $0 \ldots 1$. Using a constant ratio r results in a current frame being an exponential sum of all past frames (previous contributions have an exponential decay):

$$c = cr + p_1 r^2 + p_2 r^3 + p_3 r_4 + \ldots$$

where p_i is a previous frame i frames past. This might be exactly what we want, but frames blended in this way will never converge, there might be too much current frame's contribution. We can set up ratio r to give us a mean value of a sequence instead like this:

$$r = \frac{1}{age}$$

Figure 4.3: A single frame of our one-sample-per-pixel input. The scene is lit by a singe sphehe-shaped light source. Shadow of the chair formed by a direct lighting is noisy, but discernible, the soft indirect bounces on the lion and in the back are, however, only very sparsely sampled.

where *age* is a number of frames since disocclusion or a first frame. This always converges to a perfect mean, but lefts oáll possible reprojection errors as a part of an image. We can combine these two approaches instead:

$$r = max(r_0, \frac{1}{age})$$

We can compute the age-adjusted r at first (so new samples contribute more when there is not enough information yet), but resort to exponentially-fading ratios later to gradually remove any potential errors by enforcing a minimum current frame ratio r_0. We can se ethe result of the accumulation in figure 4.4, with some possible reprojection artifacts seen in figure 4.5.

4.1.3.1 Variance

As multiple frames arrive, our path tracer always shoots a new ray in a random direction (according to BRDF (see [Nic65]) or at least a cosine with the surface normal). These different directions always result in a different value in the output pixel: there are multiple probability distributions. These distributions typically have a different variances: a completely lit surface's variance is close to zero (all rays ended up in a light), but a smooth, wide shadowed area has much more variance to it (only some rays end up in a light).

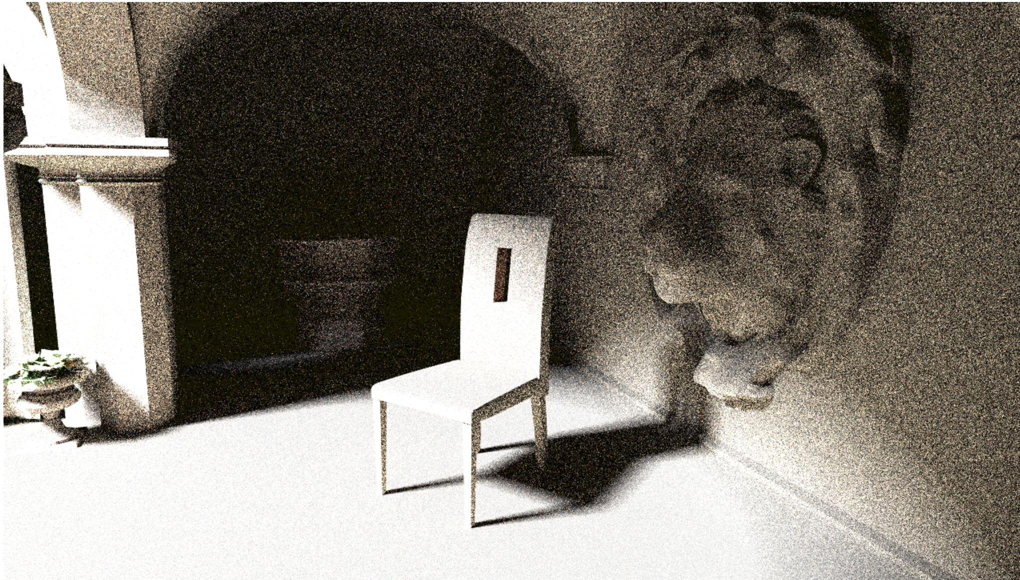

Figure 4.4: The same scene as in 4.3 after a few frames of accumulation: new detail emerges, but there's still a lot of noise. The noise would start to disappear only after seconds into the accumulation (as we saw in figure 2.6).

This variance is very important to the bilateral filter at the next stage (it recognizes shadow boundaries and avoids blurring them), so we need to approximate it here. We convert RGB values to luminance before computing variance using formula:

$$grayscale = 0.299R + 0.587G + 0.114B$$

This saves us 4 valuable float values in video memory. It's possible this configuration might overblur some edge cases of different-colored shadows (we haven't tested for that yet), but the extension to using per-channel variance would be trivial if it's ever needed.

The definition formula for variance computation looks like this:

$$var(x) = E[(x - \mu)^2]$$

where x are grayscale values of a pixel at different samples and μ is their mean value. This tells us what variance is: an expected value (average) of a squared deviation of samples from their mean. However, computing this value requires having all past grayscale values in memory while we might have infinitely many of them. Accumulating this term is impossible (we don't know the μ beforehand) but luckily, we can compute variance in other, mathematically equivalent way:

Figure 4.5: Some pixels weren't visible in previous frame and it isn't possible to blend them with a prpevious frame. This creates visible artifacts in the accumulation: there are some places along edges with visibly less accumulated samples. This additional noise gets filtered by our spatial bilateral filter during the next algorithm stage 4.1.4.

$$var(x) = \sum x^2 - (\sum x)^2$$

We just need to store two float values this time ($\sum x$, $\sum x^2$) and accumulate them using the same ratio r as with our diffuse buffer. We just need to compute their difference later to acquire our variance.

Variance acquired this way can be very noisy, which can lead to some artifacts. We can alleviate this problem by blurring variance buffer by a small 3x3 kernel: we don't lose much spatial information this way because of the noise and the results are much better, especially around sharp shadow's edges.

4.1.4 Spatial filtering

We accumulated a number of samples from a few previous frames and computed their variance; now we can filter them spatially. We will constrain the blurring by normal, depth and variance buffers from earlier using a bilateral filter and three similarity weight functions defined later.

4.1.4.1 Bilateral filter

Bilateral filter is an edge-preserving smoothing filter (see [TM98]): it blurs the image but sums similar pixels more (what is similar is defined by some metric). Our bilateral filter uses a Gaussian window; this is what a simple Gaussian blur looks like:

$$I_f(x) = \sum_{i \in d(x)} W_i I_i, \quad W_i = g_i$$

where I_f(x) is a final image color at a pixel x, $I(x)$ is an image color at a pixel position x, $i \in w(x)$ are pixels i in a window d around a pixel x and finally, $W_(i)$ is a weight of a current pixel sample in a window, which is set to some gaussian kernel $g(i)$. This is a Gaussian blur which simply sums a neighboring pixel's contributions using a Gaussian kernel.

A bilateral filter differs from Gaussian in its weighting function $W(i)$. It depends on the actual pixel values as well:

$$W_i = g_i w_{xi}, \quad \sum_{i \in d(x)} W_i = 1$$

The extra weighting factor w_{xi} represents this dependence; it weights similarity of two pixels w, i in some way. The renormalization is needed now for the filter to preserve energy. How exactly is w_{xi} derived is presented in following pages. The w, i might be omitted for clarity; the meaning is obvious from the context.

4.1.4.2 Normal weighting function

We can use our normal buffer input to blend similarly-facing surfaces: that is, pixels with similar normals. A good measure of angle similarity is taking their dot product: dot products get large for same angles and zero for orthogonal. Assuming N_0 to be the first normal and N_1 offset normal being blended, both of them normalized ($\sqrt{x^2 + y^2 + z^2 = 1}$):

$$w_N = max(0, N_0 \cdot N_1)^o N$$

Fragments with similar normals are blended this way. Exponent oN controls how similar normals need to be to be mixed. Values around 64 work well.

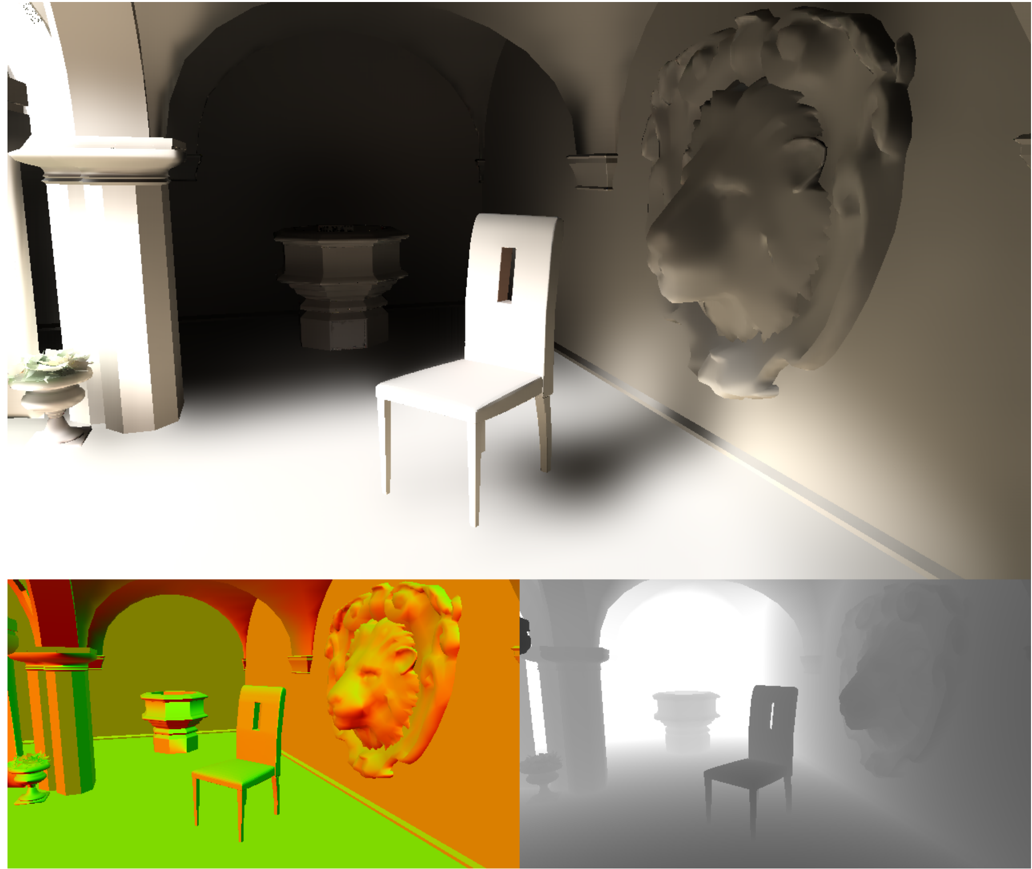

Figure 4.6: The accumulated scene is filtered by our bilateral filter with normal and depth weights driven by normal and depth buffers pictured below the image. We can see the geometry-based edges have been preserved, but all important structure introduced by path tracing has been lost by too much blurring (like the chair's shadow).

4.1.4.3 Depth weighting function

Depth information is a bit harder to weight because it can differ wildly when sampled in screen space. We need to blend pixels together both on a nearly flat surface and at steep slopes very collinear to camera direction, a simple depth difference is not enough. We approximate a local slope given by screen-space depth derivatives and define our weight function by a deviation from this slope:

$$w_D = \exp \frac{-|D_0 - D_1|}{|oDgrad \cdot off| + \epsilon}$$

D_0 is the first depth, D_1 is the offset depth being blended into it, $grad$ is the gradient approximated from screen-space derivatives, off is screen-space

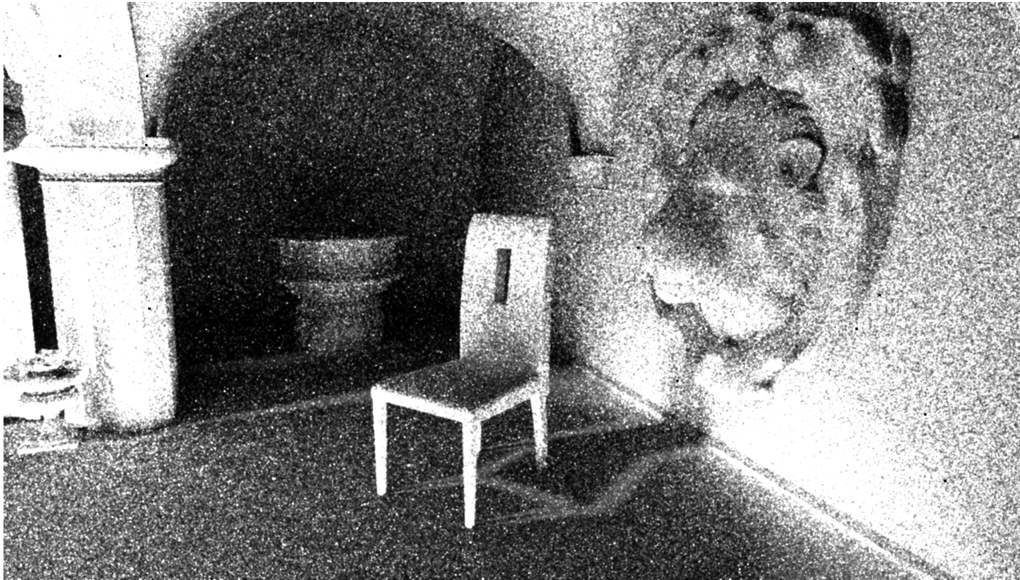

Figure 4.7: Variance buffer: lighter color represents more variance at the currect pixel. Variance highly correates with luminence, but differs in important places: the floor is very bright, but with smaller variance (most rays simply hit the light) and shadow edges have higher variance than their surroundings because secondary rays at their pixels ended up in more different places. More noise is present in these images compared to accumulated diffuse because variance is calculated using terms with a square power, amplifying all errors.

offset of D_1 from D_0. ϵ is a small value to avoid division by zero (depends on the scale of depth values used, around 0.005 works fine for a normalized depth buffer) and exponent oD controls a threshold of quantization artifacts.

This depth weight function works fine in most of the cases except very steep and narrow sides. Very narrow and steep parts of the depth buffer have a very little information to filter through and result in a visible noise which is not very aesthetically blended into the following frames. That's why we turn off the depth weight function for too steep gradients: normal weight is sufficient in this case. This is accomplished by setting $w_D = 1$ when $|grad| > c$ where c is a very small scene-dependent constant.

4.1.4.4 Luminance weighting function

At this stage, our image filtering respects edges given by normal and depth weights w_N, w_D, but shadow detail generated by path tracing is still being overblurred (as we can see in figure 4.6). We need some luminance weight, where only similar colors are being blended. That's not easy, because our accumulated buffer still is still very noisy and boundaries aren't readily visible. Luckily, we have prepared our variance buffers 4.7: we can blend pixels with

high variance together even if their colors are different (there is more noise, so we need to blur them more). Our luminance weight looks like this:

$$w_L = \exp \frac{-|L_0 - L_1|}{(oLvar + \epsilon)}, \quad var = min(var_0, var_1)$$

L_0 is the first luminance value, L_1 is the offset value being blended, var is our variance, ϵ is a small value to avoid division by zero and offset minimal noise (depends on input noisiness and dynamic range, typically somewhere around 0.01 and 0.1) and exponent oL controls filtering strictness. Adjusting the filter's sensitivity to noise is possible by changing the ϵ: the filter blurs more when ϵ is bigger. var is determined as a minimum of source var_0 and destination var_1 because we only want to mix together two high variances: adding a high-variance pixel to a low-variance one could result in bleeding artifacts.

4.1.4.5 Combining the weights

Now we have all three weights w_N, w_D, w_L and we can combine them by simply multiplicating them:

$$w = w_N w_D w_L$$

Resulting w will be used as a weight for our bilateral filter. We can see the results when using all three of our weights in the figure 4.8.

4.1.4.6 Edge-Avoiding À-Trous Wavelets

Accumulated buffer we're filtering using our bilateral filter is still very noisy: we need a very wide filter to take care of the noise. Bilateral filter is in its brute-force version can be very slow: we are iterating over every kernel pixel for every image pixel. We will help ourselves instead with a hierarchical approach based on Edge-Avoiding À-Trous Wavelets introduced to noisy image data filtering by Dammertz et al. ([DSHL10]).

Another, albeit much less cool or French name for À-Trous Wavelets Transform is Stationary Wavelet transform (SWT). SWT is a slightly modified version of Discrete Wavelet Transform (DWT) with traslation invariance: it just leaves out a downsampling/upsampling steps and interleaves the downsampling/upsampling kernel instead. SWT processes data in this way:

1. Begin at level $i = 0$ with input signal c_0.

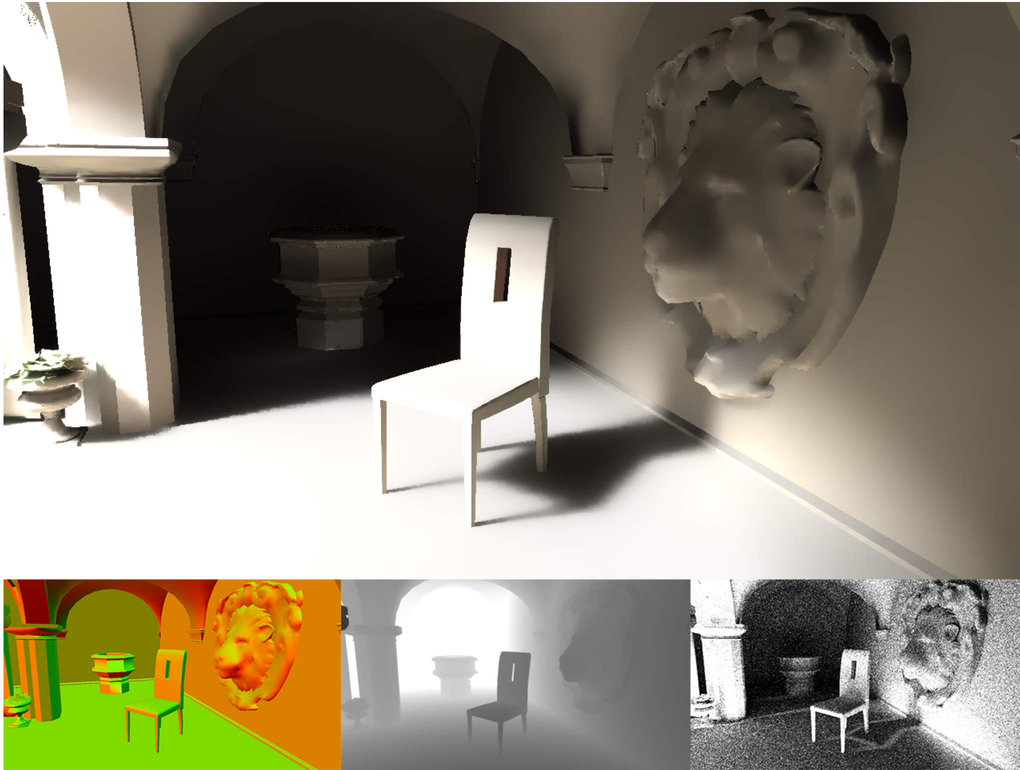

Figure 4.8: The result after applying our bilateral filter with weights calculated using normal, depth and variance buffers (pictured below the image). Our bilateral filter blurs more values of high variance together (such as the shadow borders) resulting in sharp shadows.

2. $c_{i+1} = c_i * k_i$ where $*$ is a convolution and k_i is a kernel with 2_i zeros interleaved between each pixel.
3. $d_i = c_{i+1} - c_i$ are detail coefficients of level i.
4. Increment i and go to step 2 until the required number of iterations is done.
5. $d_{0...N-1}$ and c_N are the wavelet transform of c_0.

We're not interested in detail coefficients, we can leave out step 3. A simple 3×3 kernel is used as a k. A 3×3 kernel convolution applied a few times in a row is a good Gaussian filter approximation: 6 levels of these convolutions are approximately equivalent to a 77×77 Gaussian blur kernel, but much faster to calculate on a GPU. The original paper used 5 levels of 5×5 kernels: I found out using 6 levels of 3×3 kernels instead executed much faster on lower-level GPUs and were of a better quality (larger kernel). The number of levels is arbitrary; one can use any number of levels depending on the amount of noise (image features begin to be a limiting factor for large kernels). The multi-scale convolution is illustrated in figure 4.9.

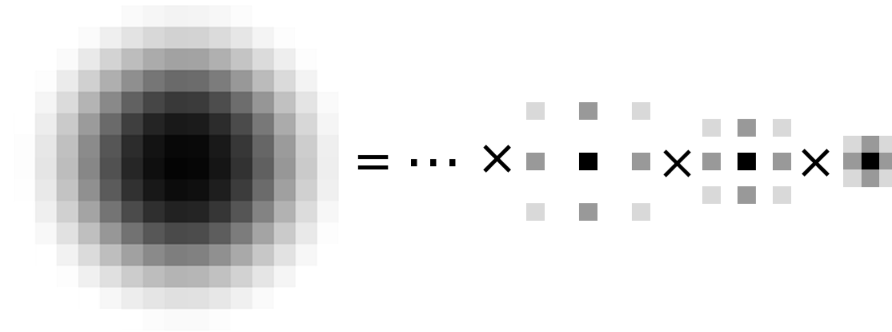

Figure 4.9: Illustration of multi-scale convolution used for accelerating the bilateral filtering: a wide convolution kernel is approximated by multiple smaller, faster convolutions in exponentially decreasing skip size ($2^2 - 1, 2^1 - 1, 2^0 - 1 = 3, 1, 0$ pictured).

We can introduce the blending weights w into À-Trous Wavelets by applying them at each scale. The results of later levels with large kernels can introduce artifacts by skipping large portions of the image (where they have zeros in their kernel). Luckily, the convolution is commutative, so we can do the scale operations in any order (like Hanika et al. have shown in [HDL11]): by doing the large scales first, the later low-scales stages clean up any high-frequency artifacts introduced by an earlier high-scale stage. Instead of doing convolution at scales $i = 0, 1, 2, 3 \ldots$, we do them at $i = 5, 4, 3 \ldots$ This detail is very important to the quality of the result but it's somehow missing from the original paper. The artifacts in question can be seen in figure 4.11. An example of filtering stages at multiple scales can be seen in firuge 4.10.

Remember the reprojected last frame in our accumulation phase? It's beneficial to reproject the output of the first, highest-frequency wavelet first instead of the raw last accumulated frame: convergence speed is increased at the cost of a small bias introduced to our light, as we can see in figure 4.12 (ϵ of the luminance weight w_L is more impactful to the bias in practice since the bias is largely eliminated by variance weighting). The first wavelet lookup might make hard shadows a bit softer over time, but speeds up convergence. It depends on our preference: we can even turn it off after enough samples were gathered.

Our first wavelet transforms now, however, has the largest step-size of $2^5 = 32$ and would introduce some artifacts to our accumulation buffer. We can rearrange the highest-frequency wavelet to be the first now instead. This might introduce some artifacts by not having it done last, but that doesn't happen in practice. Our wavelet filtering stages order now looks like this: $i = 0, 5, 4, 3, 2, 1$. The variance in accumulation phase still needs to be

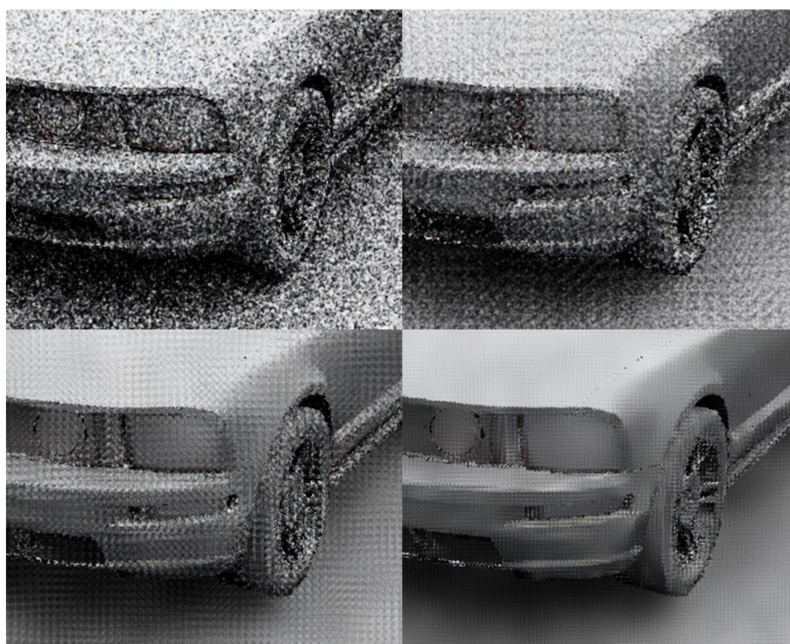

Figure 4.10: A few stages from bilateral filtering of a car model. Top left image is unfiltered, next three in reading order are three stages from our 6-stage bilateral filter. The scales are applied in a decreasing order to subsequently filter high-frequency artifacts introduced in a past low-frequency stage. These artifacts can be seen as “boxy” patterns in the noise. It can happen the pixel is filtered only by earlier stages and the artifact patern appears in the output: this is, however, noticeable only at very low sample counts and right after disocclusions, when one sample distributes too far.

calculated from raw accumulated data, without the first wavelet filtration: there must be two separate buffer inputs to the accumulation phase for variance calculation and further processing.

There is one more motivation of using a hierarchical implementation of a bilateral filter: we can filter our variance between stages. Variance is one order noisier compared to the diffuse lighting (it's calculated using a squared value) and calculating w_L based on a single, unfiltered value from variance buffer during large-kernel stages is too inaccurate, we would like to filter over a much larger area. Instead, we can filter our variance buffer in the same way as our diffuse image, just using a w^2 (image is blended with a weight w, so variance needs to be blended with the same weight squared):

$$var(L+1) = var(L)w^2$$

where $var(L)$ is a variance used for a w_L computation at a level L. We can see an example of this bilateral-adjusted variance in figure 4.13. Note that this filtered variance only propagates trough the wavelet filtering, we

Figure 4.11: The hierarchical wavelet filter was applied in increasing order of scales (high-frequency first, low frequency later). The resulting artifacts are clearly visible as repeated borders along edges.

still have to use the raw 2-component variance in the accumulation phase. This scale-depedent variance filtering formally transforms our purely bilateral filter into a bit more clever one that adjusts its weight at larger scales.

4.1.5 Final bledning

We have much less noise in our image now, thanks to the previous accumulation and spatial blending stages. There is, however, still a big problem with our output: all operations were done on a one-pixel level till now, and our image might manifest an aliased jagged edges following the same crude information in our normal and depth buffers. Merely introducing supersampling into these buffers doesn't solve our problem: our blending weights used in spatial filtering don't behave appropriately when the input is being interpolated.

Instead, we will introduce a technique well-known in real-time deferred rendering engines: temporal antialiasing (see [Kar14]). First, current filtered diffuse irradiance is modulated by albedo and tone mapping is applied (we want to filter the final appearance now). The previous frame is reprojected and blended with current one using the same formula as in accumulation phase:

$$c = cr + p(1 - r)$$

where ratio r is a constant this time (somewhere around 0.1 tends to work nicely). Now we need a new mechanism to handle disocclusions: the

depth-based approach from our accumulation phase won't work here because it doesn't operate on a sub-pixel level as we need and smearing would occur without any care for disocclusions.

The industry standard to this problem is blending with a previous frame's value clipped to the 3×3 neighborhood of the current pixel:

$$c = rc + (r - 1)clamp(p, c_{min}, c_{max})$$

where c is a current pixel's value, p is a previous reprojected pixel's value, r is the two frame's blending ratio and c_min, c_max are minimum and maximal value form c's 3×3 neighborhood. This formula implements clamping, clipping is very similar, but depends on p's direction and converges faster.

Some form of sub-pixel jittering is applied to the camera matrix prior to the path tracing itself to introduce subpixel information into our buffers temporally even when the camera is stationary: Halton sequences work very well for that. There might not be a well-reprojectable pixel in a current pixel neighborhood when the camera is moving: a 3×3 block can be searched for a sample closest to p instead: this reduces aliasing under motion.

This form of final temporal AA reduces aliasing around edges and minimizes all noise a bit further. A bit of wiggle room in clamping might need to be added to account for substantial amounts of noise. It also greately reduces remaining noise in isolated areas that didn't have enough similar areas around them in a spatial filtering stage (such as far away or thin objects). Additional FXAA, SMAA or similar screen-space approaches can be used on top of temporal AA to get rid of any other aliasing issues.

We can see how jagged lines get cleaned up in figure 4.14 and the result of our final blending stage in figure 4.16.

4.1.6 Tuning the algorithm

) We can balance the detail, the noise level and reprojection error using three parameters:

- a constant *age* in accumulation stage: setting this parameter higher prevents reprojection error from accumulating, but we lose some detail in shadows (as we can see on figure 4.15), it can always be se to 0 for a static camera
- *epsilon* in w_L calculation': higher values smooths shadows more, lower values speed up convergence at the cost of some noise

- we can gradually turn off the spatial filtering stage 4.1.4 for a noise/detail compromise: in practice, a low aoount of noise doesn't hurt the visual quality, but increases detail

The suggested values for parameters oN, oD, oL work without any issues in every scene to my knowledge. The steepness parameter from 4.1.4.3 depends on the depth range of the scene (the wrong setting doesn't break the algorithm, only adds noise at horizon).

4.2 Fewer samples per pixel

The previously described algorithm works well, converges very quickly (visually around 4 frames) and handles reprojection well. It, however, always expects one sample per pixel. Our path tracer might not be that fast every time (a low-end GPU, a CPU-based path tracer) but we would like to have real-time framerates anyway. The algorithm can be modified to tackle this problem. There are some issues we need to overcome, aside from apparent skipping of empty values (we can mark them by -1 or NaN).

4.2.1 Missing normal and depth information

This applies if we're using a path-traced normal and depth buffers: there might be holes present when a sample wasn't rendered during the last frame. That is a problem because spatial filtering stage expects a full 1 sample per pixel normal, depth and albedo buffers to constrain its filtering: it would get restricted around the holes instead, which is undesirable. We need to implement some changes, as we can see in diagram 4.17.

We need to somehow account for the missing information. One possible way is raytracing the normals, depth, and albedo buffer first (before path tracing towards the light), but our raytracer probably isn't fast to do that anyway. Data in diffuse irradiance buffer is very sparse at this stage as well (only one sample per pixel, where normals and depth aren't missing), so the results end up very blurry anyway, and a linear approximation is enough.

We implemented a top-down pyramid downsampling of the normal, depth and albedo buffers (inspired by Ritschel et al. [SA12]). The buffers are downsampled to a third of their size multiple times until no holes are visible and missing information in high-resolution buffers is then looked up from the lower-resolution levels using linear interpolation to get smooth gradients. That's a crude and blurry approximation, but good enough in practice (normal, depth and albedo buffers fill up to more meaningful values quickly afterward anyway).

This hole-filling stage can be useful for other parts of this algorithm as well (such as spatial variance approximation discussed later). We'd like more space to be available in there buffers: let's save some space. Using three floats for a normalized normal is wasteful, we can do the same thing with two of them if we choose a representation valid under interpolation.

We can use a spheremap transform, first appearing in CryEngine: source code of both conversions is in figure 4.18. *encodeN* converts three-component normalized normal into a compact two-component representation, decodeN converts them in the other way around. Both conversions are very speedy.

4.2.2 Overblurring from first wavelet level accumulation

Our original algorithm feedbacks the first wavelet level from spatial filtering to the new accumulation every time: this is fine for a sample per pixel every time, but overblurs the image if there is too little new information every frame. One possible solution that works for us in practice is feeding back the wavelet-filtered first level only after one full sample per pixel was accumulated: the jump only occurs in the noise which is filtered away during the spatial phase, so it isn't visible at all. Blending partially between previous raw accumulation and the smoothed wavelet wasn't successful for us.

4.2.3 Overblurring caused by too high ϵ in w_L

Using very low amount of samples like this results in a very high noise in the image and forces us to set ϵ in luminance weight w_L very high, essentially blurring our image more. But leaving the ϵ constant for the whole time, even when there is enough samples accumulated, actually overblurs now correct data in our accumulation buffer: we need to adjust ϵ according to the number of samples at a given place in an image. This would probably need some other pass over the image to be robust enough and wasn't implemented in our filter yet: ϵ lowers with more samples per pixel, but an additional slider is present to fine-tune the constant. It appears ϵ can be left constant when there are more than 3-4 samples per pixel already.

4.2.4 Reprojecting a fraction of samples per pixel

The extension described till now works fine for static inputs, but sad things can happen when reprojecting. It is, again, an issue only if you have ray-traced normal, depth and albedo buffers. It's not, unfortunately, possible to do the last frame reprojection the classic way: we can only reproject our new samples, but we have no information about the pixels in between. We need to forward reproject the old frame to the new one in some way. That's a

scatter operation, but we can use atomic operations of compute shaders in current GPU's to implement them effectively.

One possibility is projecting every pixel from the previous frame to world space and scattering it to the new frame from it. This solution introduces a bit of an error by rounding up pixel positions in screen-space: some pixels can even stay in the same space for multiple frames if their offset is less than one pixel. That is visually very appealing, but not what we want.

We saved the pixel's world-space position to a buffer instead of using depth to reconstruct it. The error introduced by rounding screen-space pixel positions is gone, but there is still a problem: there are no reprojected pixels in some parts of the screen. This missing information can quickly build up during camera movement: we need to approximate them somehow. We opted for a 3×3 neighborhood average in UV-space (UV used to look up the previous texture), but that still introduces a few-pixel jitter to the sharp shadow's edges and isn't perfect. A more robust solution such as a raymarching one or a few of the previous depth-buffers could result in a much better reprojection, but we didn't have time to implement something like that yet.

We need to take care to output only the topmost surface when more of them get scattered to the same position: we use atomic operations in compute shader to accomplish that.

4.2.5 Handling occlusions with holes

There is one more caveat when reprojecting images with holes (again, only for ray-traced normals, depth and albedo buffers): surfaces further in the distance might be visible through holes in geometry in front of it and mess up the hierarchical hole-filling. It clears up after some time but can be very distracting, as we can see in figure 4.19.

We look for the closest and farthest surface in our depth buffer in the 27×27 neighborhood (by using three 3×3 kernel passes) and discard any reprojected sample that is sufficiently far away (closer to the far side than near side). The kernel size can be bigger or smaller, depending on sample count. That solves our problem but introduces an unfortunate half-kernel-sized border of visibly blurred pixels in the image (all samples there were discarded), as we can see illustrated in figure 4.20. These discarded parts could be recognized by some edge-detecting filter and returned, but we didn't have time to apply that.

4.2.6 Temporal AA

Final blending stage of our filter expects one sample per pixel input to introduce a subpixel resolution to the resulting image. Nothing like that is,

unfortunately, achievable with such a low sample count. We didn't implement any approach solving this problem yet: lowering the blending ratio between the two frames is a quick workaround which somehow works. Subpixel jittering isn't recommended, it would introduce even more error in our reprojection. The right solution would be implementing an additional buffer for blending: only new samples would get updated there.

4.3 More samples per pixel

But what if we have the exact opposite problem: more samples at a single pixel? That shouldn't be a problem at first glance, but we would be throwing away a lot of information about variance by applying the basic algorithm right away: the variance between pixels is valid, but there is an additional variance inside a single pixel between multiple samples being blended. We don't want to throw this variance away.

The solution is very straightforward: the path tracer computes the variance for us while blending the samples in a single pixel. Later, we will mix this variance with previous frame's one according to the sample count: all this time using the two-component representation. We can compute the final variance only after this blending.

Reprojection is no issue in this scenario, we can use the usual approach. The normal, depth and albedo buffers are still a single sample per pixel: we might be throwing a bit of information away here, but temporal antialiasing can take care of most of it for us. The correct solution would probably be computing the normal/depth/albedo variance and blending the neighboring pixels accordingly.

4.4 Spatial variance estimate

The described algorithm works very well when there is at least some variance estimate: in other words when there are two or more samples. That, however, isn't always the case: we have invalid variance information even at the first frame with a one sample per pixel input.

We can approximate the variance spatially in this case and hope to avoid blurring at least the contrasty direct light samples. We can easily look at the current pixel's neighborhood and compute a weighted variance around a current pixel using a Gaussian-window (possibly using a hierarchical search approach from a previous section 4.2.1). We use only as narrow windows as possible to avoid overblurring the variance approximation and fall back to the default variance calculation when two or more samples are available.

In short, filtering during first few frames depends on our preference: we can have detailed, but noisy first frame or blurry, but noiseless one. It's even possible to split the filtering stages to sharp and soft direct and indirect diffuse lighting to filter them in a controlled, separate way (thisn approach was used in the original paper [SKW+17]).

Figure 4.12: Default accumulation buffer is shown on the top and the accumulation buffer with first-wavelet previous frame lookup on the bottom. The bottom one certainly has much less noise and speeds up convergence this way. It, however, increases bias as we can see in form of a slightly blurred shadow edges in the magnified area around the chair's legs. The bright speckles are new, yet un-spatially-filtered samples: technique using less samples per pixel was used (more about this version of the algorithm is described in section 4.2).

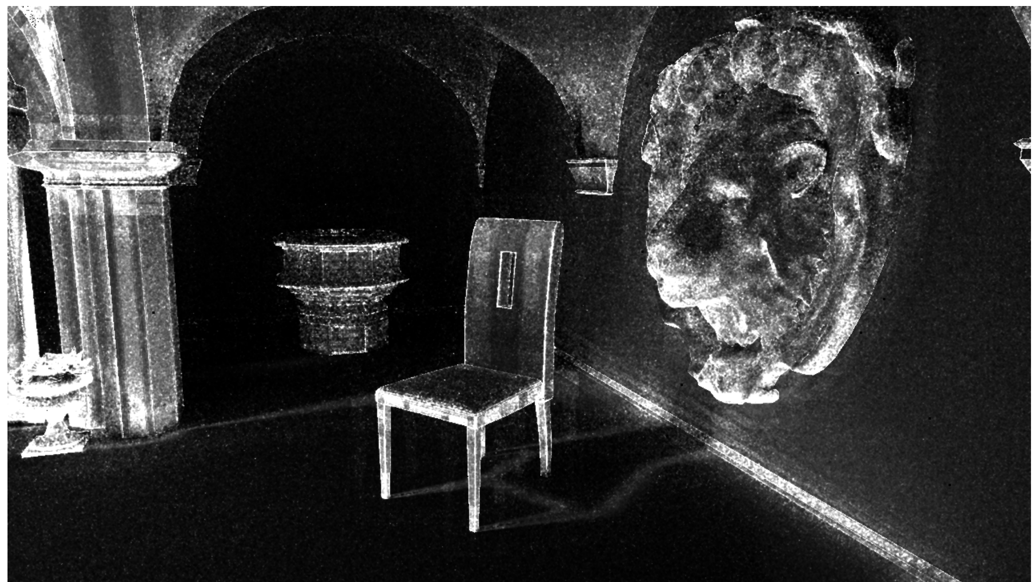

Figure 4.13: Variance buffer after being filtered by bilateral filter's weights two times. Only prominent areas are weighted significantly at this stage: only pixels with high variance are allowed to be filtered in later scales of the bilateral filtering to prevent overblurring (the variance weighting is scale-awave).

Figure 4.14: Temporal antialiasing smooths both jagged lines (easily visible with complex geometry) and noisy low-sampled areas (like the han rail here). Top image is without temporal antialiasing, bottom one with it.

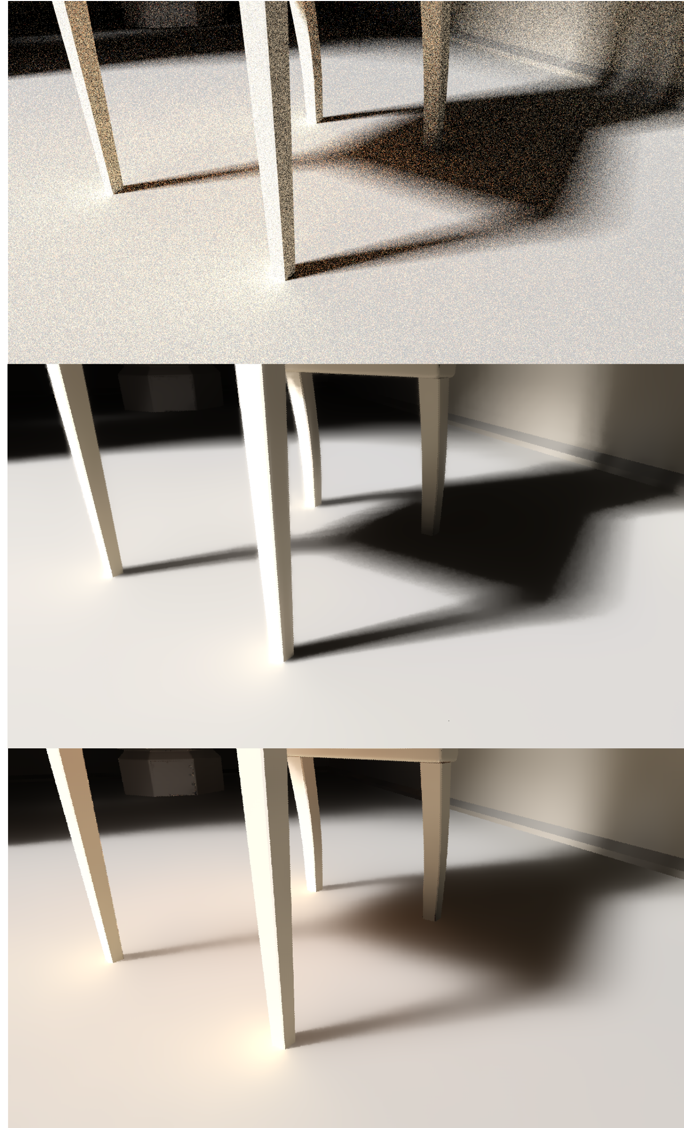

Figure 4.15: Accumulation ratio r is important for detail preservation. Top image is rendered using 16 samples/pixel with our filter off, second one using unbounded r and the third one with $age = 5$. The shadow's shape is correct even in the third row, but penumbra detail has been lost due to too aggresive blending. The missing slight brown tint in the middle picture is caused by changing material properties.

Figure 4.16: The result after the final blending phase: albedo is multiplied back in and themporal antialiasing is applied by blending with a previous frame.

Figure 4.17: This diagram illustrates the changes (highlighted in red) we need to make to our algorithm to support even less than one sample per pixel as input. All modifications are discussed in section 4.2.

```
vec4 encodeN(vec3 n)
{
    float p = sqrt(n.z*8+8);
    return vec4(n.xy/p + 0.5,0,0);
}
vec3 decodeN(vec2 enc)
{
    vec2 fenc = enc*4-2;
    float f = dot(fenc,fenc);
    float g = sqrt(1-f/4);
    vec3 n;
    n.xy = fenc*g;
    n.z = 1-f/2;
    return n;
}
```

Figure 4.18: GLSL code implementing encoding and decoding normal between its normalized 3-component and packed 2-component form. Code source:

Figure 4.19: It can occur previous frame gets reprojected under new sparse samples that are actually closer to the camera and should cover the old ones, resulting in holes in objects cleaning up only after many new frames. Our solution is using a min-filter-based rejection described in section 4.2.5.

Figure 4.20: Illustration of a eroded depth buffer: all holes with incorrect surfaces (much further compared to the eroded depth buffer) get rejected.

Chapter 5

Implementation and results

5.1 Performance

We implemented the algorithm into an existing CPU path tracer VRUT as a post-processing filter implemented in OpenGL. The progressive path tracer running on a single 6-core CPU can accomplish one sampler per pixel in 720p in about one second, so it's a good testing environment for our "less than one sample per pixel" version.

The benchmarks were done on these PC configurations (notebook, desktop and notebook):

- Intel HD 530 (2015, integrated GPU), 4-core Intel i7-6700HQ
- NVidia GeForce 720 (2013, mid-tier GPU), 4-core Intel Xeon E3-1240v3
- NVidia GeForce 1060 (2016, mid-tier GPU)

We experimentally found out that the performance wasn't sensitive to scene structure or amount of reprojection: inter-frame variation hid all differences. This is not true for the original algorithm: the authors of [SKW^{+}17] use a more complex one-pass bilateral filter only for disoccluded pixels, so their performance depends on the number of new pixels. We tested the algorithms using a single scene (Sponza with a chair) in 1280×720 and measured the timings using OpenGL queries GL_TIME_ELAPSED. The performance scales approximately linearly with the number of pixels on the screen. All render times are in milliseconds: they are mean values from 100 measurements with maximal deviation at 5th or 95th percentile after $\pm$.

stage	Intel 530 [ms]	NV 760 [ms]	NV 1060 [ms]
>=1spp	24.6 ± 1.01	5.92 ± 0.24	3.7394 ± 0.1016
<1spp	48.52 ± 1.97	8.09 ± 0.29	N/A
<1spp moving	52.83 ± 2.81	9.0299 ± 0.31	N/A
pre-reproj	4.32 ± 0.23	0.93 ± 0.032	N/A
accumulation	10.46 ± 0.57	1.13 ± 0.04	N/A
wavelet	13.66 ± 0.81	4.36 ± 0.19	N/A
+ CPU transfer	4.02 ± 0.14	4.13 ± 0.23	N/A

But what do the weird names in the first “stage” column mean?

- >=1spp: whole filtering stage when using one or more samples per pixel (with or without movement).
- <1spp: whole filtering stage when using less than one sample per pixel: hole-filling stage (see 4.2.1) is added.
- <1spp with movement: moving camera requires a hole-filling preprocessing stage using a scatter operation (see 4.2)
- pre-reproj: only a hole-filling preprocessing stage in <1spp (see 4.2)
- accumulation: only accumulation (possibly with reprojection)
- wavelet: only spatial filtering (see 4.1.4)
- + CPU transfer: additional time spent by copying data from CPU to GPU in our CPU-based VRUT. Seems to be the same for both configurations.

We can see the algorithm performs very well on NVidia 760: it’s faster than realtime (about 160 fps) at 720p and likely is at 1080p as well. <1spp variants would attack the 60 fps threshold at 1080p. Integrated 530 performs a bit worse: it accomplishes 30 fps at 720p.

It’s not realistic to do any fast path-tracing on integrated Intel 530, but it shows it’s usable for CPU path tracers or less real-time applications, where 30 fps isn!t necessary. NVidia 760 or any newer GPU can realistically both path-trace the scene and filter it afterward: the ratio gets only better with newer GPUs. As we can see (at least by a single measured >=1spp), the algorithm is faster on notebook 1060, but not much: there is probably a memory transfer limit.

An interesting thing to note is a strangely long execution time of accumulation stage on Intel GPU: it’s nearly as slow as a wavelet filtering stage, which is a 6-pass bilateral filter. This has probably something to do with a substantial shader length (Reproject.fsh has nearly 300 lines). The shader code is very messy and experimental and probably could be cleaned up significantly.

There wasn't any time to compare the performance of the filter with others, but our version runs at about the same speed on NVidia 760 as the original paper's version (see [SKW+17]) on Titan Pascal. This has to do mainly with a smaller bilateral kernel size (3×3 instead of 5×5) and partially with using a simpler spatial filter (instead of their large 1-pass bilateral filter).

5.2 Quality

We tested the quality of our results by accumulating 200 samples first and then testing structural similarity (SSIM) of the first few frames of the accumulatin with and without our filter against the high-quality 200-sample image. Our filter resulted in more similar results, by far.

SSIM without filter: 0.330, 0.334, 0.810 (variance) , 0.820, 0.824 0.826...
SSIM with filter: maximum 0.230

Chapter 6

Conclusion

The algorithm works, indeed! It denoises the low-sampled path tracing input rapidly by getting clues from scene geometry and sample variance. It resolves most of the lighting detail in images after just a few frames for one sample and pixel and an adequate number of frames for less or more samples. The filter takes a few milliseconds to filter the image, leaving enough time for GPU to do the path tracing itself. The current version of the algorithm supports opaque scenes with static diffuse lighting and a moving camera; all of these limitations can, however, be solved to some degree by possible extensions of the algorithm.

6.1 Limitation: no scene or light movement

Only camera movement is supported at the moment: any scene movement (with correct velocity buffer) or light movement results in smeared shadows (this smearing effect is illustrated in figure 6.1) or very gradual brightness changes. The lazy, highly-approximate solution to this problem would be increasing current-frame ratio of the reprojection during accumulation very high: the trailing invalid light data would be shorter and less noticeable, possibly blurring only a soft shadow or hidden by a motion blur in the scene. The correct solution to this problem is straightforward: determining which parts of the image's lighting changed, can't be used for reprojection anymore and should behave the same as disoccluded pixels.

We can use variance buffer to help us determine how much of an outlier our new value is: values laying well inside our variance range would be blended with the previous frame a lot and pixels outside just a little or not blended at all. Sharp shadows created by direct lighting would reject old pixels with wildly different values quickly and soft shadows would move to new values more gradually: this agrees with how humans perceive light.

A usable spatial variance approximation would be needed for fast-moving

Figure 6.1: This is what would the accumulation buffer look like without any reprojection applied (the camera is moving to the left). A similar smearing effect would be visible on shadows in a scenario a light or an object would move: uncorrelated pixels would get blended unless rejected using some other mechanism.

shadows (so they don't get over-blurred). This modification would be very beneficial: scenes with moving geometry, moving lights and more complex effects such as caustics would be possible to filter. More complicated methods could somehow attempt to reproject shadows from previous frames; that is, however, a very complex problem.

6.2 Limitation: diffuse component only

The algorithm in its current state can handle only a diffuse type of light. Specular component of light can be path-traced separately and blended with diffuse lighting without any filtering: filtering it the same way as the diffuse component would result in overblurring the reflections over the surface normals and reprojection would be invalid.

We would need to introduce a second, separate filtering pipeline for treating specular lighting output. Let us consider a sharp reflection first: the reprojection would need to be carried out in a "reflection space" (the offset vector would indicate the reflection movement from previous to current frame). The spatial filtering could be implemented in two ways, depending on how important reflection detail is for us:

1. Track reflection variance over time and filter them spatially using their variance.

Figure 6.2: Implementing the algorithm and ignoring reflections yields incorrect resuls. Top image shows a correctly pathtraced car with a reflection, the bottom one the same scene when our filter is applied: reflection is overblurred and it has albedo's color instead of a reflected one. The path to the correct solution is described in section 6.2.

2. Do the entire diffuse pipeline for the specular reflection as well: gather specular lighting and render albedo, normal and diffuse buffers inside the reflection. Accumulate the light buffer and filter it using both normal and depth buffers from diffuse and specular components. Modulate with albedo afterward.

The second approach would reproject the first specular bounce perfectly but would fail for second, third and all more recursive bounces. These could be handled easily for mirror-like surfaces, where the reflection is perfect, and only the innermost reflection is filtered. The other cases would need to fallback to the first variance-only filtering approach after a certain number of bounces or reproject using the brightest (most visible at a given pixel) reflection while using a more gradual frame blending, similar to the soft variance-based rejection idea from the previous section 6.1.

Glossy reflections (with blurred features) represent another problem: their normal and depth aren't clearly defined and can't be filtered or reprojected easily. Good approach to solving this problem could be using only variance-based filtering: interpolated normal and depth weight is not compatible with our w_N, w_D and doesn't introduce more information than variance anyway. Later, they would be reprojected by filtered (softened by blurring) UV coordinates and blended, again, by a softer blending function similar to the variance-based rejection discussed previously in section 6.1.

In short: an approach to handling specular reflection depends on how crucial specular lighting/reflections are in the current scene. A pure variance spatial filtering + reflection-space reprojection could be enough.

6.3 Limitation: no transparency

Our algorithm in its current form supports only fully opaque surfaces: it uses a single-layered GBuffer as a single source of information about the scene. The obvious next step is filtering the light buffer in multiple layers.

Let's suppose our path tracer doesn't blend all the surfaces for us and outputs the irradiance and normal, depth and albedo information for every layer separately, but possibly out of order. We need to use some order-independent transparency algorithm: a lot of research has been done in this area already.

The state-of-the-art method producing an accurate solution is probably Order Independent Transparency with Per-Pixel Linked Lists by Barta et al. (see [BKSSK11]) which stored the overlapping fragments and sorts them at each pixel. That, however, requires an unbounded memory when there are too many surfaces required to be drawn. Another popular, this time approximative approach Weighted, Blended Order-Independent Transparency by Morgan McGuire and Louis Bavoil (see [MB13]) uses, on the other hand, only a fixed amount of memory: it approximates the visibility of incoming surfaces by a heuristic calculated from depth and coverage information and works very well after a bit of tuning.

Both of these approaches can be combined: using per-pixel linked lists to a certain memory limit and just approximately weighting them together afterward. The weighted-blended surfaces cannot be exactly reprojected or filtered anymore: a soft variance-based rejection would come into play again.

6.4 Discussion

This method overall seems to be a very intriguing approach to filtering the noisy data acquired as an output of a path tracer. Not using any filtering similar to this one for real-time ray tracing/path tracing would be a waste: it can relatively easily enhance the visual quality of the output and reduce very evident noise for a much more subtle bias (especially in a textured environment). The filter can be used in its current form in a constrained setting (filtering only raytraced shadow etc.) or for all the lighting later (when specular filtering is usable as well).

One obvious alternative is a machine-learning approach: it probably has the advantage of no warm-up time (2-3 frames for 1 sample per pixel), but is probably more GPU-heavy and leaves less computing power or the path tracing itself.

One thing is for sure: real-time path tracing is slowly arriving and upcoming technologies enabling it are going to be very interesting.

Appendix A

Bibliography

[BKSSK11] Pál Barta, Balazs Kovacs, László Szécsi, and László Szirmay-Kalos, *Order independent transparency with per-pixel linked lists*, 2011.

[CKS+17] 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** (2017), no. 4, 98:1–98:12.

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann, *Interactive indirect illumination using voxel cone tracing: A preview*, Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '11, ACM, 2011, pp. 207–207.

[DSHL10] Holger Dammertz, Daniel Sewtz, Johannes Hanika, and Hendrik P. A. Lensch, *Edge-avoiding À-trous wavelet transform for fast global illumination filtering*, Proceedings of the Conference on High Performance Graphics (Aire-la-Ville, Switzerland, Switzerland), HPG '10, Eurographics Association, 2010, pp. 67–75.

[GTGB84] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile, *Modeling the interaction of light between diffuse surfaces*, SIGGRAPH Comput. Graph. **18** (1984), no. 3, 213–222.

[HDL11] Johannes Hanika, Holger Dammertz, and Hendrik Lensch, *Edge-optimized a-trous wavelets for local contrast enhancement with robust denoising*, 1879–1886.

[HDMS03] Vlastimil Havran, Cyrille Damez, Karol Myszkowski, and Hans-Peter Seidel, *An efficient spatio-temporal architecture for animation rendering*, ACM SIGGRAPH 2003 Sketches &Amp;

Applications (New York, NY, USA), SIGGRAPH '03, ACM, 2003, pp. 1–1.

[Jen96] Henrik Wann Jensen, *Global illumination using photon maps*, Proceedings of the Eurographics Workshop on Rendering Techniques '96 (London, UK, UK), Springer-Verlag, 1996, pp. 21–30.

[Kaj86] James T. Kajiya, *The rendering equation*, Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '86, ACM, 1986, pp. 143–150.

[Kar14] Brian Karis, *High-quality temporal supersampling*, SIGGRAPH Courses: Advances in Real-time Rendering in Games, SIGGRAPH, 2014.

[Kel97] Alexander Keller, *Instant radiosity*, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '97, ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56.

[LTH+13] Christian Luksch, Robert F. Tobler, Ralf Habel, Michael Schwarzler, and Michael Wimmer, *Fast light-map computation with virtual polygon lights*, Proceedings of ACM Symposium on Interactive 3D Graphics and Games 2013, ACM, March 2013, pp. 87–94.

[LW93] Eric P. Lafortune and Yves D. Willems, *Bi-directional path tracing*, PROCEEDINGS OF THIRD INTERNATIONAL CONFERENCE ON COMPUTATIONAL GRAPHICS AND VISUALIZATION TECHNIQUES (COMPUGRAPHICS 93, 1993, pp. 145–153.

[MB13] Morgan McGuire and Louis Bavoil, *Weighted blended order-independent transparency*, Journal of Computer Graphics Techniques (JCGT) **2** (2013), no. 2, 122–141.

[MMBJ17] Michael Mara, Morgan McGuire, Benedikt Bitterli, and Wojciech Jarosz, *An efficient denoising algorithm for global illumination*, Proceedings of High Performance Graphics (New York, NY, USA), ACM, July 2017.

[MMNL16] M. Mara, M. McGuire, D. Nowrouzezahrai, and D. Luebke, *Deep g-buffers for stable global illumination approximation*, Proceedings of High Performance Graphics (Aire-la-Ville, Switzerland, Switzerland), HPG '16, Eurographics Association, 2016, pp. 87–98.

[MMNL17] Morgan McGuire, Mike Mara, Derek Nowrouzezahrai, and David Luebke, *Real-time global illumination using precomputed light*

field probes, Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (New York, NY, USA), I3D '17, ACM, 2017, pp. 2:1–2:11.

[Nic65] Fred E. Nicodemus, *Directional reflectance and emissivity of an opaque surface*, Appl. Opt. **4** (1965), no. 7, 767–775.

[PRDD15] Stefan Popov, Ravi Ramamoorthi, Fredo Durand, and George Drettakis, *Probabilistic connections for bidirectional path tracing*, Proceedings of the 26th Eurographics Symposium on Rendering (Aire-la-Ville, Switzerland, Switzerland), EGSR '15, Eurographics Association, 2015, pp. 75–86.

[RDGK12] Tobias Ritschel, Carsten Dachsbacher, Thorsten Grosch, and Jan Kautz, *The state of the art in interactive global illumination*, Comput. Graph. Forum **31** (2012), no. 1, 160–188.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel, *Approximating Dynamic Global Illumination in Screen Space*, Proceedings ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2009.

[SA12] M. Solh and G. AlRegib, *Hierarchical hole-filling for depth-based view synthesis in ftv and 3d video*, IEEE Journal of Selected Topics in Signal Processing **6** (2012), no. 5, 495–504.

[SKW+17] Christoph Schied, Anton Kaplanyan, Chris Wyman, Anjul Patney, Chakravarty R. Alla Chaitanya, John Burgess, Shiqiu Liu, Carsten Dachsbacher, Aaron Lefohn, and Marco Salvi, *Spatiotemporal variance-guided filtering: Real-time reconstruction for path-traced global illumination*, Proceedings of High Performance Graphics (New York, NY, USA), HPG '17, ACM, 2017, pp. 2:1–2:12.

[SL17] Ari Silvennoinen and Jaakko Lehtinen, *Real-time global illumination by precomputed local reconstruction from sparse radiance probes*, ACM Transactions on Graphics (Proceedings of SIGGRAPH Asia) **36** (2017), no. 6, 230:1–230:13.

[ST90] Takafumi Saito and Tokiichiro Takahashi, *Comprehensible rendering of 3-d shapes*, Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques (New York, NY, USA), SIGGRAPH '90, ACM, 1990, pp. 197–206.

[TJ97] Rasmus Tamstorf and Henrik Wann Jensen, *Adaptive sampling and bias estimation in path tracing*, Rendering Techniques '97 (Vienna) (Julie Dorsey and Philipp Slusallek, eds.), Springer Vienna, 1997, pp. 285–295.

[TM98] C. Tomasi and R. Manduchi, *Bilateral filtering for gray and color images*, Proceedings of the Sixth International Conference on Computer Vision (Washington, DC, USA), ICCV '98, IEEE Computer Society, 1998, pp. 839–.

[Vea98] Eric Veach, *Robust monte carlo methods for light transport simulation*, Ph.D. thesis, Stanford, CA, USA, 1998, AAI9837162.