TÉCNICO LISBOA

Real Time Hybrid Rendering Techniques

Pedro Alegria Granja

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: João António Madeiras Pereira

Examination Committee

Chairperson: Prof. Daniel Jorge Viegas Gonçalves
Supervisor: João António Madeiras Pereira
Member of the Committee: Prof. Fernando Pedro Reino da Silva Birra

July 2021

Acknowledgments

I would like to thank my parents and family for all the love and support they have provided throughout the years and especially during this project. Without your encouragement and patience this project would not be possible.

I would also like to acknowledge my dissertation supervisor João António Madeiras Pereira for showing interest in this project and providing insight, support and sharing of knowledge making this Thesis possible.

Last and not least, I would also like to thank all my friends and colleagues that shared my troubles and where present at all times, with a special thanks to Diogo Almiro, Diogo Tito and Susana Gamito. Thank you so much for being my friends.

To each and every one of you – Thank you.

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Abstract

Ray tracing has long been the holy grail of real time rendering. This technique, commonly used for photo realism, simulates the physical behavior of light, at the cost of being computationally heavy. With the introduction of Nvidia RTX graphic card family, which provides hardware support for ray tracing, this technique started to look like a reality for real time. However, the same problems that afflicted the usage of this technique remain, and even with specialized hardware it is still extremely expensive. To account for these drawbacks, researchers and developers pair this technique with rasterization and denoising. This results in a hybrid system that tries to join the best of both worlds, having both photo realistic quality and real time performance. In this work we intend on further exploring hybrid render systems, offering a **review of the state of the art with a special focus on real time ray tracing** and our own **hybrid implementation with photo realistic quality and real time performance** (>30 fps), implemented using the Vulkan API. In this project, we highlight the detailed analysis of the impacts of History Rectification (Variance Color Clamping) on the temporal filter component of the denoising system and how to overcome the introduced artifacts. Additionally, we also highlight the analysis of the introduction of a separable blur on the spatial filter and the introduction of Reinhard Tone Mapping prior to denoising, consequently improving this procedure.

Keywords

hybrid rendering , real-time, ray-tracing pipeline, denoising

Resumo

Ray Tracing foi ao longo dos anos o grande objetivo de renderização em tempo real. Esta técnica, usada normalmente para a criação de imagens fotorealistas, simula o comportamento da luz, com a contrapartida de ser extremamente dispendioso. Com a introdução das placas gráficas da familia RTX da Nvidia, que dão suporte em hardware para ray tracing, esta técnica tornou-se possível para aplicações de tempo real. No entanto, mesmo com este suporte, esta técnica continua a ser extremamente dispendiosa. Para combater estas desvantagens, os investigadores e programadores, emparelharam esta técnica com a da rasterização e denoising. Isto resultou num sistema hibrido que tenta juntar o melhor de dois mundos, tendo qualidade foto realista e mantendo desempenho de tempo real. Neste trabalho pretendemos explorar estes sistemas de renderização hibridos, **oferecendo uma revisão do estado da arte com especial focus em ray tracing em tempo real** e a nossa propria implementação de um **sistema de renderização hibrido com qualidade foto realista e desempenho de tempo real** (>30 fps), implementado usando a API Vulkan. As maiores contribuições obtidas através deste projeto passam pela análise dos impactos da técnica de History Rectification (Variance Color Clamping) na componente do filtro temporal do sistema de denosing e como enfrentar os artifactos introduzidos pela mesma. Adicionalmente, é também importante de realçar a análise da introdução de um Separable Blur no filtro espacial e da introdução de Reinhard Tone Mapping antes da aplicação de denoising, consequentemente melhorando este procedimento.

Palavras Chave

renderização hibrida, tempo real, pipeline de ray tracing, denoising

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Introduction

Real time ray tracing hardware disrupted the real time industry generating a lot of research on how to best use this technology. Initially, games adopted this technique scarcely, being used only for specific effects like ray traced shadows[1] or ray traced reflections[2]. With time, the adoption has been steadily increasing and RTX is slowly becoming its own rendering pipeline and used to compute part or full global illumination. This research culminated in games like Quake 2 with complete path traced global illumination.

With the advent of multiple implementations, a question remains... What is the best way to use this technique? And how much should we use it in our real time pipeline for quality while maintaining performance?

One popular solution, commonly used by the real time industry, is the combination of ray tracing, rasterization and denoising into a hybrid system. For this reason, in this work, we further explore this approach and offer our own implementation.

1.2 Contributions

During the development of this project a major part of the time resources went into the denoising component of the system and as such, represent a great part of our contributions. We adapted a current state of the art technique (Spatiotemporal Variance-Guided Filtering or SVGF) in order to use history rectification, improving temporal response and reducing ghosting artifacts. This technique, however, reduced the efficiency of the temporal component of our system which we counterbalanced by adapting the spatial filtering component without increasing the computational cost. We also introduce a separable blur in the spatial filter component, greatly increasing the performance of this technique. Last, but not least, we show how applying Reinhard Tone Mapping before denoising, reduces flickering and contributes to a more homogenous final image.

1.3 Document Structure

This document follows a bottom up approach, starting with the basic concepts and restrictions (what we choose to not explore at least on an initial approach), all the way to the current state of the art, evaluation of our system and results. First, in the *Background* (section 1.3) , general concepts like the

[1] Shadow of the Tomb Raider
[2] Battlefield V

Figure 1.1: Scenes rendered in Octane Renderer by Enrico Cerica [1] and visualization of the rendering equation with w_0 being the direction towards the eye [2]

rendering equation and rendering strategies are introduced. In this section we also introduce restrictions into our work and approach. In the *Related Work* (section 2) the state of the art of hybrid rendering is presented, offering a detailed view on how to develop such a system. In the *Implementation* (section 3) we describe how our system was developed, followed by the *Evaluation Methodology* (section 4) where we proceed to describe how our system was evaluated. The *Results* of this evaluation are then presented on section 5 and finally we present a conclusion and future work on section 6. Background

In this section we aim to explore the introductory theory and the necessary restrictions for the entirety of the remaining document. We start by introducing the rendering equation (section 1.4), the goal of most realistic rendering algorithms and what we aim to approximate with our system. In the following section 1.5 we present the scattering function, a shading component of the rendering equation and what describes the way light interacts with an object's material. The project will tackle just the reflectance component of the chosen shading model. The transmittance component is not considered. The concept of the reflectance function will be described in section 1.6. We introduce the Cook-Torrance shading model on section 1.7, a specific model for the BRDF. The rendering equation is then decomposed and concepts like direct and indirect lighting (necessary for the remaining of the document) are presented in section 1.8. We finish the preliminaries by introducing the fundamental rendering strategies in section 1.9. These strategies are the tools used to approximate the rendering equation or part of it.

1.4 Rendering equation

The rendering equation introduced by both David Immel et al. [19] and James Kajiya [20] describes the way light interacts with a surface point under realistic circumstances. Solving this equation or a derivation produces realistic results (see Fig. 1.1.) and thus is the main goal of most realistic rendering algorithms.

The result of this equation is called global illumination and is given by the following (see table 1.1 for details about the equation components):

$$L_0(x,\omega_0) = L_e(x,\omega_0) + \int_\Omega f_r(x,\omega_i,\omega_0)\, L_i(x,\omega_i)\,(\omega_i \cdot n)\, d\omega_i \tag{1.1}$$

Table 1.1: Mathematical notation

x	surface point
ω_0	outgoing direction
ω_i	incoming direction
n	normal of surface point x
Ω	unit hemisphere centered around n
$L_i(x, \omega_0)$	incoming radiance/light in the incoming direction (ω_i)
$f_r(x, \omega_i, \omega_0)$	BSTF function of point x. Proportion of incoming Radiance/-Light reflected (ω_i) or refracted towards ω_0
$L_e(x, \omega_0)$	radiance/light emitted in the outgoing direction (ω_0) by surface x
$L_0(x, \omega_0)$	radiance/light in the outgoing direction(ω_0) by surface x

1.5 BSTF

The bidirectional scattering distribution function (BSTF) [21], represented as f_r in the rendering equation (Eq. 1.1), describes the way light is scattered by a surface. This is usually the combination of the BRDF function (bidirectional reflectance distribution function) and the BTDF function (bidirectional transmittance distribution function). In simple terms, these equations describe the way light is reflected (BRDF) and refracted (BTDF) after hitting a surface. **For simplicity and due to time constraints, we choose only to consider the BRDF function**. Throughout this document we thus refer to f_r as the BRDF.

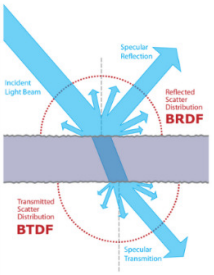


Figure 1.2: $BSTF = BRDF + BTDF$ [3]

1.6 BRDF

The bidirectional reflectance distribution function (BRDF), represented as f_r in the rendering equation (Eq. 1.1), attempts to represent the way an incoming light ray interacts with an object and is reflected outwards. BRDFs might be measured directly with special cameras, but more analytic models are used. These models split complex phenomena in more simplistic components like a diffuse and specular component (see Fig. 1.3). These components are then adjusted to create a wide variety of different materials with different surface properties. Many different BRDF models exist but only some attempt to be faithful to reality (physically based BRDF). **We explain and focus on the Cook Torrance model due to being physically based and thus producing realistic results**.

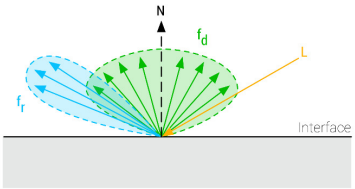


Figure 1.3: f_d represents the diffuse component and f_r is the specular component, together they provide a solid description on the way light interacts with surfaces [4]

1.7 Cook-Torrance

Cook-Torrance is a physically based shading model based on the theory of microfacets for the specular reflection. It is used by Disney [22] and was adopted by Epic Games [23] and many other popular engines

for real time rendering. A microfacet surface model assumes that any surface can be described by tiny, perfectly reflective mirrors called microfacets. Depending on the roughness of the surface, the microfacets alignment changes and consequently their reaction to light (see Fig. 1.4).

Figure 1.4: Specular reflection: rough vs smooth surface. Reprinted image from learn opengl [5].

The Cook-Torrance shading model contemplates a diffuse and specular component (Fig. 1.3):

$$f_r\left(x, \omega_i, \omega_0\right) = k_d \cdot f_{Lambert} + f_{cook-torrance}\left(x, w_i, w_0\right) \tag{1.2}$$

$$f_r\left(x, \omega_i, \omega_0\right) = k_d \cdot \frac{c}{\pi} + \frac{DFG}{4\left(w_o \cdot n\right)\left(w_i \cdot n\right)} \tag{1.3}$$

- k_d is the ratio of incoming light that gets refracted/diffused
- $f_{Lambert}$ is the diffuse component, known as Lambertian diffuse denoted as $f_{Lambert} = \frac{c}{\pi}$. π is the normalization factor. c is the surface color (albedo).
- $f_{cook-torrance}$ is the specular component:

$$f_{cook-torrance}\left(x, w_i, w_0\right) \;=\; \frac{DFG}{4\left(w_o \cdot n\right)\left(w_i \cdot n\right)} \tag{1.4}$$

$f_{cook-torrance}$ is represented by 3 functions, the normal **D**istribution function, the **F**resnel equation and the **G**eometry function.

1.7.1 Normal Distribution function

Approximates the amount of microfacets aligned to the halfway vector (h) according to surface roughness. Epic Games uses GGX Trowbridge-Reitz:

$$D\left(n, h, r\right) = \frac{r^2}{\pi\left(\left(n \cdot h\right)^2\left(r^2 - 1\right) + 1\right)^2} \tag{1.5}$$

- r is the surface roughness (values in range [0-1]).

1.7.2 Geometry function

Controls self-shadowing, due to microfacets overshadowing one another when the roughness is high, reducing reflected light. The Unreal paper [23] uses the Schlick Smith approximation [24]. However this technique was found to not be physically based [25] and a more accurate solution has been proposed, the Smith's height-correlated function [25] defined as follows (We present an equivalent version used by the Falcor framework [26] due to notation):

$$G(w_o, w_i, n) = \frac{1}{1 + \Lambda(w_0, n) + \Lambda(w_i, n)}, (w_i \cdot n) > 0, (w_o \cdot n) > 0 \tag{1.6}$$

with $\Lambda(w, n)$ being an auxiliary function which measures the self-shadowed microfacet area (masked) per visible microfacet area [7].

$$\Lambda(w,n) = \frac{-1 + \sqrt{(}1 + r^2 \cdot \frac{1-(w\cdot n)^2}{(w\cdot n)^2})}{2} \tag{1.7}$$

1.7.3 Fresnel function

The Fresnel effect states that depending on the view angle (w_0), reflections become stronger or weaker. The Fresnel-Schlick approximation, presented as follows:

$$F_{Schlick}(h, w_0, F_0) = F_0 + (1.0 - F_0) * (1.0 - w_0 \cdot h)^5 \tag{1.8}$$

- F_0 represents the base reflectivity (Amount of reflectivity at the zero incidence angle).
- h represents the halfway vector between $\omega_0 and \omega_i$.

The equation in this form holds only for dieletric materials (non-metallic surfaces). To account for metals a parameter called metalness is introduced. This parameter controls how close F_0 is to the surface color (albedo). In GLSL pseudo code:

$$\begin{aligned} vec3\ F_0 &= vec3\,(0.04) \\ F_0 &= mix\,(F_0, surfaceColor, metalness) \end{aligned} \tag{1.9}$$

To note, there are many different options for the functions presented, **we choose to cite the ones we intend on using due to providing good enough results for our objectives**.

1.8 Rendering equation decomposition

Instead of trying to solve the entire rendering equation, it is usually better to try to solve a more limited part of the equation. As such we decompose the integral into the necessary relevant parts for the remaining of the document.

The first thing we note is that the rendering equation can assume an area form defined over all the surface points of the scene ($all\ x'$). This formulation is useful for calculating direct lighting from area light sources:

$$\int_{all\ x'} f_r\,(x, \omega_i, \omega_0)\, L_i\,(x', -\omega_i)\, G\,(x, x', w_0, w_i)\, V\,(x, x')\, dA' \tag{1.10}$$

- V(x, x') is the binary function that indicates if surface x' is visible from x. This function is also responsible for shadows.
- $G\,(x, x')$ is called the geometric term and is given by the following:

$$G\,(x, x', w_0, w_i) = \frac{(w_0 \cdot n)\,(w_i \cdot n')}{||x - x'||^2} \tag{1.11}$$

L_i can be separated into direct and indirect lighting. Direct lighting is the result of direct lighting contributions from light sources. Indirect lighting is the contributions from the light reflected from other objects.

$$L_i = L_{direct} + L_{indirect} \tag{1.12}$$

Using the new area formulation, and the separation of L_i we can write:

$$\int_{\Omega} f_r \cdot L_{direct} \cdot (\omega_i \cdot n)\, d\omega_i + \int_{\Omega'} f_r \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i \tag{1.13}$$

$$\int_{all\ x'\ in\ light\ sources} f_r \cdot L_{direct} \cdot G\ \cdot V \cdot dA' + \int_{\Omega'} f_r \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i \tag{1.14}$$

Where the first integral is responsible for direct illumination and the second by the indirect illumination. Ω is replaced by Ω' to indicate that the domain is now the entire hemisphere excluding the direct light contributions (They are calculated separately in the first integral). **For simplicity, on this project we will only consider point lights and directional lights**. For point lights, we can thus replace the integral for a sum (No need to integrate over an area):

$$\sum_{all\ point\ light\ sources} f_r \cdot L_{direct} \cdot G\ \cdot V \cdot dA' + \int_{\Omega'} f_r \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i \tag{1.15}$$

Another important decomposition is the separation of the specular term from the diffuse term in the indirect lighting integral (second integral of Eq. 1.14):

$$\int_{\Omega'} \left(k_d \cdot \frac{c}{\pi}\right) \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i + \int_{\Omega'} \left(\frac{DFG}{4\,(w_0 \cdot n)\,(w_i \cdot n)}\right) \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i \tag{1.16}$$

$$\int_{\Omega'} \left(k_d \cdot \frac{c}{\pi}\right) \cdot L_{indirect} \cdot G \cdot V \cdot dA' \ + \int_{\Omega'} \left(\frac{DFG}{4\,(w_0 \cdot n)\,(w_i \cdot n)}\right) \cdot L_{indirect} \cdot (\omega_i \cdot n)\, d\omega_i \tag{1.17}$$

The first integral is responsible for the indirect diffuse illumination and effects like color bleeding. The second integral is responsible for the indirect specular illumination and for effects like reflections (see Fig. 1.5 for visual representations of these effects).

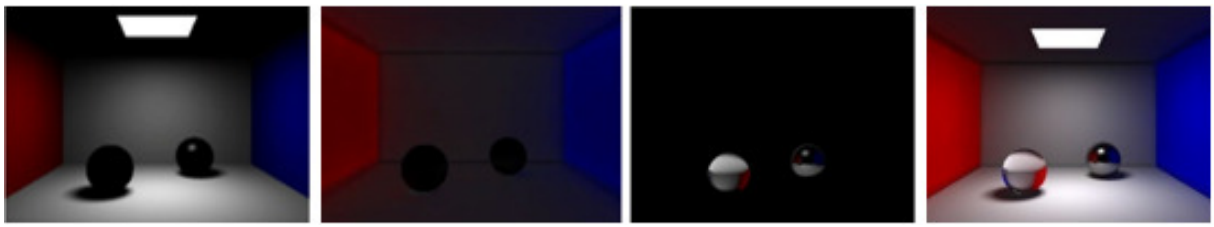


Figure 1.5: Direct illumination, Diffuse Indirect illumination, Specular Indirect Illumination and Global Illumination (composition of previous results), ImagesRelatedWork authored by lu, Kofan and Chang, Chun-Fa [6]

1.9 Rendering strategies

Two fundamental rendering strategies can be used to try to solve the rendering equation, rasterization and ray tracing. In the following sections we introduce the ray tracing techniques and more specifically Path Tracing, a popular offline ray tracing technique that solves the full rendering equation (Eq. 1.1). We choose to introduce this technique as it is commonly used for real time ray tracing implementations. After, rasterization is introduced. Rasterization is unable to solve the full rendering equation, having access to almost only local information (what is seen by the camera) and as such, it is composed of a set of "tricks" that approximate and "fake" scene lighting. The advantage of rasterization is its speed when compared to naive Path tracing which is not suitable to real time. We finally conclude this section with a note on hybrid rendering.

1.9.1 Ray tracing

Ray tracing is the umbrella term that describes the set of algorithms that use a Monte Carlo approach in order to get an approximate solution to the rendering equation or a part of it (ex: indirect specular). These methods are characterized by tracing rays to collect information about the scene, thus the name ray tracing.

1.9.2 Monte Carlo Method

The rendering equation produces integrals that are not analytically solvable (no closed form). Fortunately, these integrals can be approximated as a sum. This operation is called Monte Carlo Method:

$$E\left[\frac{1}{k}\sum_{i=1}^{k} f\left(X_i\right)\right] \approx \int f\left(x\right) dx \tag{1.18}$$

This implies an integral can be approximated by using the mean of a set of uniformly distributed samples X_i over the domain of the integral. In the context of computer graphics, one can estimate the mentioned integrals (like Eq. 1.1) by shooting rays and given their interactions with the scene (simulate light) estimate the value of an integral. Due to being an approximation, methods that are based on the Monte Carlo algorithm produce noise. This noise is a consequence of an insufficient number of samples (rays) being taken. In the particular case of an image, noise has the appearance of grain, as can be seen in the right image of Fig. 1.6.

1.9.3 Path Tracing

Path tracing [20] is a Monte Carlo method for solving the rendering equation and calculating global illumination faithful to reality. This method solves the rendering equation by shooting rays from the camera (backwards tracing). Each ray travels through the scene and when an object is hit, a new direction on the hemisphere of the hit point is selected. The ray is then relaunched and the process repeated, until it reaches a light source or is terminated. The connections between the hit points for a specific ray form a path hence the name path tracing (see Fig. 1.6). The color of each pixel is estimated by calculating the shading at each hit point starting at the light source (In Fig. 1.6, we calculate the shading using the order $p3 \rightarrow \ldots \rightarrow p0$). The color of each hit point (to note, the light source hit point only contributes with L_e being purely emissive), excluding the pixel p_0, is given by:

$$L_e\left(x, \omega_0\right) \; + \; f_r\left(x, \omega_i, \omega_{0i}\right) L_i\left(x, \omega_i\right) \left(\omega_i \cdot n\right) \tag{1.19}$$

When the pixel of the image p_0 is reached, the color is given by the mean of all the rays shot through this point (according to the Monte Carlo method):

$$\frac{1}{k}\sum_{i=1}^{k} f_r\left(x, \omega_i, \omega_0\right) L_i\left(x, \omega_i\right) \left(\omega_i \cdot n\right) \tag{1.20}$$

Intuitively one might expect that shooting rays from the light source would make more sense (forward tracing), the problem with this approach, is that many of these rays don't reach the observer and as such don't contribute for the image being calculated. Fortunately, backwards tracing is almost equivalent to forwards tracing and as such is the preferred approach and produces equally realistic Images.

Other Monte Carlo methods for solving the rendering equation exist, including bidirectional path tracing, metropolis light transport, etc. We chose to explain path tracing as it is the most commonly used in hybrid rendering.

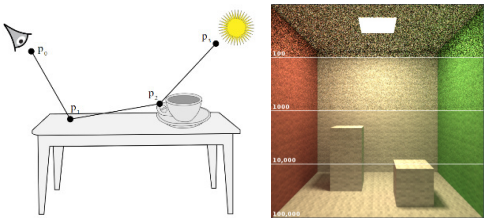


Figure 1.6: Left image - path made by a ray from the observer to the light source [7]. Right Image – Image quality and number of samples (rays) per pixel (numbers in white in the image [8].

1.9.4 Rasterization

Rasterization is a process in which 3D geometry is projected onto a 2D plane (screen). The geometry information present in the vertices is then bilinearly interpolated in order to form surfaces. These surfaces are then shaded according to their positions and the lights of the scene. This process does not map well to the way light interacts in real life and as such does not produce results with the quality of path tracing. Effectively this technique does not try to solve the rendering equation and is not a Monte Carlo method. It is, however, extremely fast when compared to Monte Carlo methods like ray tracing. Furthermore, this is the technique that has been used for most real time applications throughout the years and as such, a lot of tricks have been developed in order to get realistic results. **On this document we focus more on the ray tracing aspect of hybrid rendering** due to being the most challenging part to do in real time and what will end up representing a large proportion of our solution.

It is, however, important to introduce the idea of deferred shading [27]. Using deferred shading, on a first pass, the 3D geometry is projected onto the screen but instead of immediately shading the result, we store various attributes like the normals of the surfaces, their world positions, etc. on multiple screen sized textures. The set of these textures forms a buffer, called the G-Buffer. Textures (G-Buffer), provide the necessary information for the shading operation to be done on a secondary pass. With this technique, geometry operations are decoupled from shading operations resulting on increased performance with multiple lights on the scene. Another important aspect of this technique is the possibility of using the G-Buffer as a source of information for other processes like ray tracing and the reason why rasterization is used in combination with ray tracing in hybrid systems.

1.9.5 Hybrid Solution

Even with hardware support, full on path tracing is still difficult. However, its incredibly realistic results are desirable. As such, various hybrid solutions have been proposed by the community. This hybrid approaches use different combinations of rasterization and ray tracing, trying to balance quality vs performance.

An example of such technique is the calculation of primary rays (first rays launched from the eye) using rasterization [28] (primary rays reconstructed from G-Buffer information) and calculating the remaining using path tracing.

Another example is the calculation of almost full direct illumination using rasterization (shadows are still ray traced) and indirect illumination using ray tracing.

1.10 Vulkan Ray Tracing API

Vulkan was chosen for our implementation due to being cross platform and providing the ray tracing APIs required for the development of this project. Our implementation is done using the Vulkan API which provides a specific set of procedures and data structures in order to perform GPU accelerated ray tracing.

Ray tracing is a computationally expensive procedure and requires testing each ray against all scene geometries for intersection, and as such, requires a different pipeline and data structure to rasterization.

In Vulkan this structure comes in an abstracted form of a two-level data structure, the top level acceleration-structure (TLAS) and the bottom level acceleration structure (BLAS). Each BLAS contains the primitive geometry information while the TLAS contains a list of references to multiple BLAS with associated transform information. The programmer is responsible for creating and managing these data structures.

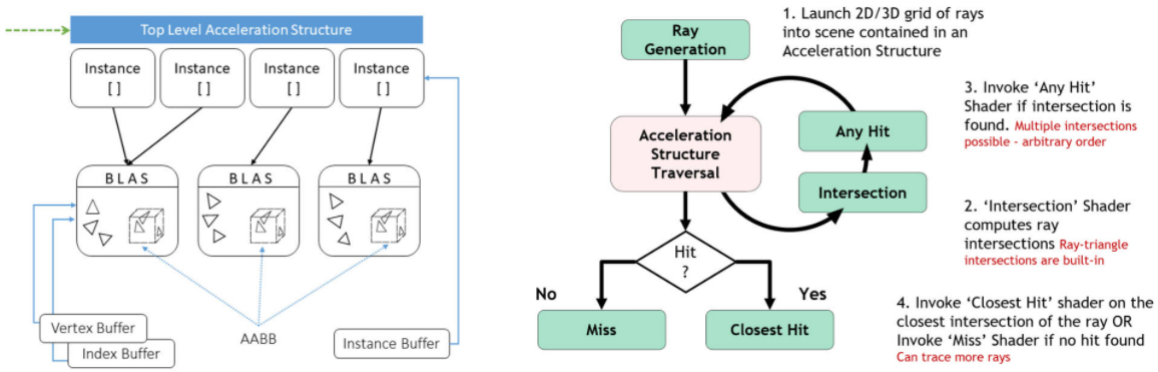


Figure 1.7: Vulkan acceleration structures [9] and ray tracing pipeline [10]

Once submitted to device, the acceleration structures are used on the ray tracing pipeline composed by the following programmable stages:

- **Ray Generation Shader:** Responsible for initiating the ray tracing process by generating primary rays.
- **Intersection Shader:** This shader provides flexibility to the system by allowing the programmer to specify custom ray-primitive intersection algorithms. Ttriangle-ray intersections are already supported as a built in function.
- **Any Hit Shader:** Invoked when a intersection is determined by the system, allowing operations to be done over the intersection point (ex. shading). This stage, is called for every intersection and doesn't guarantee any order.
- **Closest Hit Shader:** Similar to the *Any Hit Shader* with the guarantee that it is invoked only on the closest ray intersection. As such, this is the most usual location for shading operations.
- **Miss Shader:** Shader invokers when no intersections are found by the system. This is commonly used for cube maps and environment calculations.

Chapter 2

Related Work

In this section we start presenting the current state of the art of hybrid rendering.

The ray tracing component of a hybrid system requires special attention in order to get the desired frame rates. As such, in section *Optimizing Ray tracing* (section 2.1) we take a special look at this technique. After, in section *Global Illumination solutions* (section 2.2), we explain in greater detail concrete cases of hybrid rendering in the real time industry that try to solve full global illumination (Eq. 1.1). Afterwards, techniques that solve a more limited part of global illumination like indirect specular (second integral of Eq. 1.17) are presented in the *Per Effect solutions* section (section 2.3).

2.1 Optimizing Ray tracing

Due to the number of rays needed to resolve each effect, pure naive ray tracing cannot be used in real time, optimizations are required (see Fig. 1.6, a simple image requires 1000 rays per pixel to converge to a noise free result). These optimizations aim at making the overall image converge faster to a desirable quality, requiring, in general a reduced number of rays.

We start by introducing *Variance*, a statistical method for the analysis of the noise present in an image and what we try to minimize with the least number of rays possible. In real time each ray traced matters and as such we present better sampling strategies in the *Sampling* section. After, we introduce denoising techniques to remove the last remaining noise. These techniques are applied as a post processing over the image. We start by presenting *Temporal Denoising*, followed by *Spatial Denoising*.

2.1.1 Variance

Variance is a way for characterizing error in Monte Carlo methods and can be used as an estimation for the current quality of the image. As such, computing this metric is extremely important for the assessment of the techniques used. Furthermore, it can be used to guide other procedures like denoising. Variance is given as:

$$V[X] = E\left[(X - E[X])^2\right] = E\left[X^2\right] - E[X]^2 = \frac{\sum X^2 - \frac{(\sum X)^2}{N}}{N-1} \tag{2.1}$$

In practice X is the samples (rays) used for a particular pixel and N is the total number of samples used. $E\left[X^2\right]$ and $E[X]^2$ are known as the first and second raw moments in statistics.

In short, high variance means a pixel needs more information to converge (for a noise free image) and low variance the opposite.

Variance is effectively squared error and as such we need to reduce variance four times to cut noise in half. Another interesting property, as stated *On the Importance of Sampling* [29], is that for random samples, variance decreases linearly with the number of samples taken, meaning that if we double the samples, we can reduce variance in half.

2.1.2 Sampling

This section introduces better sampling techniques for the Monte Carlo Algorithms. These optimizations all aim at reducing variance and thus reduce the number of samples required for a noise free image.

2.1.2.A Quasi Random Monte Carlo

Monte Carlo assumes that samples are randomly selected over the domain. The problem with this approach is that random sequences may cause clusters or holes, resulting in parts of the domain to be under sampled (see Fig. 2.1 and notice how there are patches in the white noise). To solve this problem, quasi-random or low-discrepancy sequences can be used. These sequences generate low-discrepancy noise and cover the sample domain more evenly. A lot of options exist for these sequences such as Worley, Sobol, Halton or Hammersley (visualization of these sequences in [30]).

Blue noise in particular, has been a popular solution, being used by Unity [31] and Quake 2 [17] implementations, based on the original paper from the Arnold renderer [32]. This technique has the special property of having similar samples spread out far apart. This makes this noise specially uniform over the domain and results in better blurring and accumulation (Important for temporal and spatial denoising). To note, some of these sequences, blue noise included, can be fairly expensive to calculate and as such are usually precomputed.

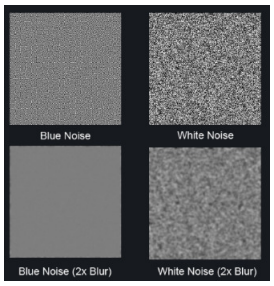


Figure 2.1: Comparison on blurring blue noise vs white noise. Blue noise is clearly more uniform vs normal random white noise. [11]

2.1.2.B Importance Sampling

The function being integrated using Monte Carlo may not weight all the domain uniformly. The rendering equation (Eq. 1.1) is one such function. The component $(\omega_i \cdot n)$ approaches zero if the incident vector ω_i is close to the horizon. This means that a sample in this direction would contribute very little to the result.

Importance sampling remedies this and allows the sampling of non-uniform distributions, biasing the

sampling towards samples that matter. The formulation is the following:

$$E\left[\frac{1}{k}\sum_{i=1}^{k}\frac{f\left(X_i\right)}{pdf\left(X_i\right)}\right] \approx \int f\left(x\right)dx \tag{2.2}$$

Using a non-uniform distribution biases the output of the Monte Carlo operation. To compensate, the weight function *pdf* is introduced ensuring the operation is unbiased. *pdf* is the probability density function of the sampling operation. In the case of the render equation a cosine weighted distribution can be used for sampling. In that case the *pdf* is given by $\frac{cos\theta_i}{\pi}$. The component $(\omega_i \cdot n)$ is not the only one that can be sampled, both the BRDF (f_r) and light sources can be used for importance sampling.

2.1.2.C Cook Torrance Importance Sampling

Lighting contributions are usually dominated by the BRDF and not the hemisphere. As such, it is usually better to importance sample based on the BRDF (and more specifically to the Cook Torrance BRDF, according to the D component due to also dominating the other functions (F and G)).

In Cook Torrance shading model, light is reflected closely or roughly around a reflection vector depending on the roughness of the surface (see Fig. 2.2).

Figure 2.2: Comparison between the specular lobe of a smooth surface vs that of a rough surface [5]

As such, importance sampling based on this specular lobe further reduces variance and is the sampling solution used by Unreal in IBL (Image Based Lighting, [23]) but can be integrated in ray tracing.

2.1.2.D Light Importance Sampling

One can also use importance sampling based on the light sources. However this is only useful if we consider area lights (We only consider non-area lights). Even so, this raises a problem, if we had to choose between importance sampling the light source and the BRDF what should we choose? The answer is both using a technique denominated multiple importance sampling.

2.1.2.E Next Event Estimation

Instead of depending on occasionally hitting the light source we can separate the direct lighting sampling contributions from the indirect lighting contributions using Eq. 1.14. By doing this separation, lights can be sampled directly by shooting "shadow rays" towards the light source. Direct lighting is one of the major contributors for the color of a point and as such this dramatically reduces variance and noise. At implementation level the only precaution that is needed is to ignore the contributions from a indirect ray randomly hitting the light sources, as lights are sampled separately.

2.1.3 Temporal Denoising

In path tracing, even with intelligent sampling, a large number of samples per pixel are still required. Temporal techniques try to accommodate this problem by reusing the information in previous frames,

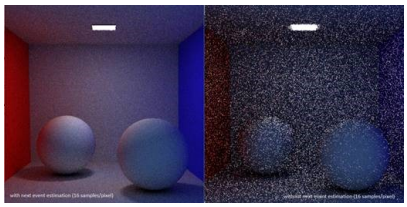


Figure 2.3: Comparison of 16 samples per pixel with next event estimation (first image) and without next event estimation (second image). [12]

reducing the number of samples per frame. This is extremely effective at denoising but introduces the problem of reprojecting the pixels from the previous frame into the new one. Furthermore, these techniques are prone to ghosting and temporal lag on moving lights and objects.

Despite the disadvantages this technique has been used in the game industry with Inside Temporal Antialiasing [33] and applied to game engines like Unreal [34].

As an overview, this technique starts by calculating a *Motion Buffer* between the previous and current frame, followed by either a *backwards* or *forwards reprojection*, where one frame is mapped onto another. Reprojection tests are used for each pixel in order to test the success of the reprojection operation. Following the reprojection tests, *history rectification* can be used, adapting the previous frame information in order to make it fit the current frame. As a final step the previous frame information is blended onto the current frame using an *exponetial moving average*.

2.1.3.A Motion Buffer

To reproject one frame into another, a Motion/Velocity buffer is required. This buffer contains the difference in position of each vertex between the previous and new frame. To calculate this buffer, it's necessary to store the MVP (Model View Projection) matrix of the previous frame and animation vertex velocity (If vertices are animated directly, the displacement between the previous frame and the current one is required).

2.1.3.B Forward Reprojection

Forward reprojection maps the previous frame into the current one. This technique is unusual due to requiring an indexing data structure in the opposite direction of what is usually presented in game engines.

2.1.3.C Backward Reprojection

Backward reprojection is the most common technique. It is the opposite of forward reprojection as it projects the current frame into the previous one. This can be done using the Motion Buffer:

$$motionVec = texture(motionBuffer, uv).xy$$
$$prevColor = texture(prevFrameColor, uv + motionVec).rgb$$

2.1.3.D Reprojection Tests

The reprojection of one pixel into another needs to be tested in order to account for disocclusion or occlusion events (see Fig. 2.4). These reprojection tests are done by comparing the features (normal, depth, color) of the pixels from the current frame and the previous frame and checking for similarity. Usually, in order to improve quality, a region of pixels (ex: 2x2) in the previous frame is tested instead of just one pixel. The pixels that pass the test, are then weighted accordingly to similarity to the current frame pixel and blended together for the next steps.

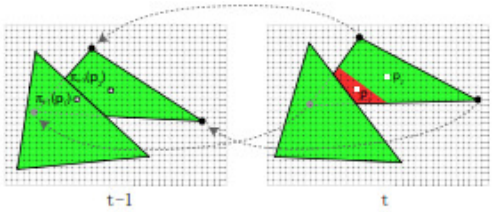


Figure 2.4: Example of a disocclusion event in backwards reprojection [13]

.

2.1.3.E History Rectification

The illumination of a point may change from one frame into another. With history rectification, the previous frame pixels are forced to fit the current frame pixel color distribution. There are 2 approaches, color clamping and clipping, with clipping being the most robust version. Color clamping [35] and clipping [34] start by calculating a color space bounding box based on a region surrounding the target pixel (ex: 3x3) on the current frame. This bounding box is then used to clip or clamp the previous frame pixel (first image of Fig 2.5.), forcing the previous colors to fit the new ones. Color clipping and clamping are susceptible to outliers and as such were extended by a more robust variance version. Variance Color Clipping introduced by Salvi [36] estimates variance from the region surrounding the target pixel (ex: 3x3) and uses this information to create the color bounding box.

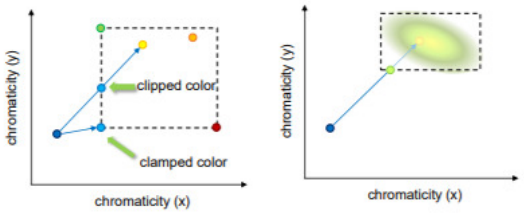


Figure 2.5: Example of color clipping and clamping on a 2D chromaticity space and variance color clipping. [13]

2.1.3.F Exponential Moving Average

To blend the information from the previous frame into the current frame an exponential moving average is used:

$$C_{final} = ((1 - \alpha) * c_{prev}) + (\alpha * c_{curr})$$

α is the parameter that defines how much of the current frame information is used. This can be a constant or selected based on a strategy, like the temporal gradient in A-SVGF [37].

2.1.4 Spatial Denoising

Temporal techniques and good sampling are still not enough to achieve a noise free image. As a final cleanup spatial filtering is usually employed. These techniques use information of adjacent pixels in order to denoise. Cross Bilateral Filters and Edge Avoiding À-Trous wavelet transform are some of the popular approaches for cleaning up the remaining noise while keeping high frequency details (small details).

2.1.4.A Gaussian Blur Filter

A Gaussian Blur filter is a low pass filter that reduces noise and smooths an image by replacing each pixel by a weighted average (based on distance) of the nearby pixels.

$$I_p = \sum_{q \in S} G_\sigma \left(||p - q|| \right) I_q \tag{2.3}$$

- S - Space domain containing all the possible pixel positions in an image
- p - Position of the pixel being denoised (central pixel)
- q - Position of pixel being visited
- G_σ - Two-dimensional Gaussian kernel
- I_q - Color of pixel q

2.1.4.B Cross/Joint Bilateral Filter

A Bilateral filter is a non-linear, low pass filter and essentially an extension to the Gaussian Blur, introducing the concept of weighting the difference between pixels in order to preserve edges. This weighting of the difference of pixels is usually done using a Gaussian of the difference of the intensities (colors) of the pixels. Furthermore, the weighting is not limited to just one weight, or to be based exclusively on color. More image attributes can be used and even information present in other Images (for example a depth image). If another image E is followed in order to filter an image I this is called a Cross/Joint bilateral filter [14] (see Fig. 2.6 for an example).

$$\begin{aligned} I_p &= \frac{1}{W_p} \sum_{q \in S} G_\sigma \left(||p - q|| \right) f \left(||E\left(p\right) - E\left(q\right)|| \right) I_q \\ W_p &= \sum_{q \in S} G_\sigma \left(||p - q|| \right) f \left(||E\left(p\right) - E\left(q\right)|| \right) \end{aligned} \tag{2.4}$$

- $E(p)$ - Color value of position p on image E
- $E(q)$ - Color value of position q on image E
- f - Function used for weighting the differences between $E(p)$ and $E(q)$

Figure 2.6: First image is the image to be denoised (image I), the second image is the one containing the edges we wish to preserve (Image E) and the last one is the result of the cross bilateral filter [14].

2.1.4.C Edge Avoiding À-Trous (with holes) Wavelet Transform (EAW)

The computational time of the usual isotropic (equal width and height) non-linear filter (like the bilateral filter) scales quadratically with the width of the kernel, making it prohibitively expensive for wide arrangements. These wide kernels are however useful in order to capture a wider range of information regarding the picture. The À-Trous wavelet transform allows the approximation of the behavior of a bilateral filter while providing a way to simulate wider kernels in real time. This filter was introduced by Dammertz [38] as a global illumination denoiser tool. The concept behind À-Trous is to instead of increasing the width and height of the filter, to introduce holes (holes represent almost 0 extra computational time) into the filter over multiple convolutions (see Fig. 2.7 for a visualization of the kernel over multiple convolutions).

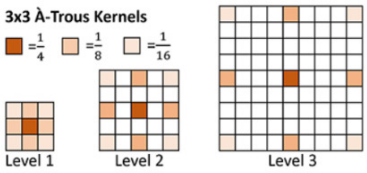


Figure 2.7: Representation of the differences in the kernel between 3 convolutions of the filter [15]

2.2 Global Illumination solutions

In this section we use the components introduced previously in order to look at concrete examples of real time renderers, their pipelines and how we can achieve real time photo realistic quality. All of the following algorithms work to solve the full global illumination.

2.2.1 Spatiotemporal Variance-Guided Filtering (SVGF)

SVGF [16] is a combination of all the techniques described and was presented as a denoising technique for 1spp (1 ray per pixel) global illumination in real time. Although more sophisticated techniques have been proposed, this technique is often the base and inspiration for most of them and is a robust way to compute global illumination. The path tracer used has the following properties:

- 1 spp color samples with next event estimation
- 1 bounce per path plus 2 shadow rays per hit point
- Rasterization for primary ray computation, making this a hybrid rendering system [28].
- Low discrepancy Halton sequence for sampling
- Path Space Regularization, essentially increase the roughness in secondary bounces in order to reduce fireflies [39]
- Separation of direct and indirect illumination for independent denoising

2.2.1.A Pipeline

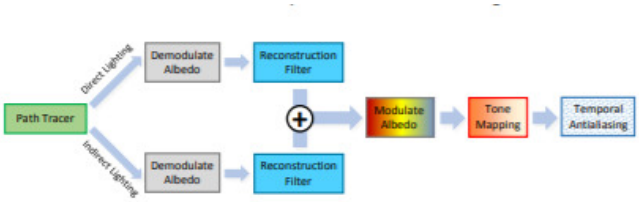


Figure 2.8: High level view of SVGF pipeline [16]

Following Fig.2.8, the process starts by path tracing the scene and splitting the results into direct and indirect illumination. Direct and indirect illumination is demodulated, removing the directly visible surfaces from sample colors (Essentially the texture data is removed from the direct and indirect lighting). This demodulation prevents the denoising pass from over blurring high frequency details from the texture information. The reconstruction filter (denoise) is then applied followed by a blending of the indirect and direct lighting and the remodulation of the texture information (step Modulate Albedo in Fig. 2.8). The next steps are post processing steps outside the context of this document.

2.2.1.B Reconstruction Filter (Denoising)

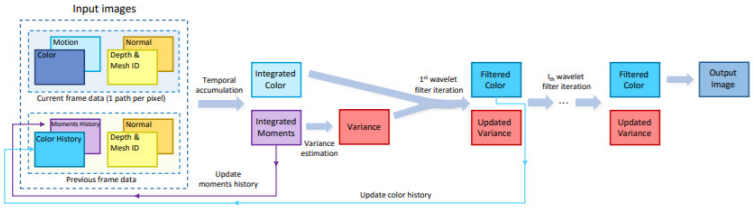


Figure 2.9: Reconstruction pipeline of SVGF used for denoising the indirect and direct illumination [16]

The reconstruction filter denoises the indirect and direct illumination and is composed by 3 steps, temporal filtering, variance estimation and spatial filtering through an Edge Avoiding À-trous wavelet transform.

2.2.1.C Temporal Filtering

SVGF uses Backward reprojection in order to access the previous frame colors and accumulate it with the current frame information using an exponential moving average with fixed $\alpha = 0.2$. To test reprojection depth, normal and mesh IDs are tested similar to the work of Krezner [40].

2.2.1.D Variance Estimation

Variance is estimated as it provides valuable information for spatial filtering, allowing a more aggressive filter where variance is high (noisy regions). SVGF computes variance temporally by calculating the first ($E[X]$) and second raw moments ($E[X^2]$) of color luminance, μ_1 and μ_2 respectively. After, if the current sample was successfully reprojected, this value is accumulated to the corresponding values of the previous frame, creating μ_1' and μ_2'. Variance is then given by $var(p) = \mu_1' - \mu_2'$. In cases where

no temporal information is available (failed reprojection), variance is estimated spatially using a cross bilateral filter with weights driven from normals and depths.

2.2.1.E Edge Avoiding À-Trous Wavelet transform

Similar to Dammertz [38], an Edge Avoiding À-Trous Wavelet transform is used, with the difference of the usage of depth, world space normal and luminance as weights. The luminance weight uses the variance estimated in order to adapt the contributions of the adjacent pixels.

After each iteration of the edge À-Trous, variance is reduced, resulting in progressively reduced blurring and thus preventing over blurring.

2.2.1.F Problems

Here we present some of the problems with this technique and motivation for the necessity of further improvements

- *Over blurred chrominance*: Due to solely tracking luminance variance.
- *Over blurred specular reflection*: Due to noise in indirect illumination, resulting in the denoising pass to over blur the reflections.
- *Detached Shadows in Motion*: Detached shadows due to temporal lag
- *Delayed color change*: Delays in color change due to temporal lag
- *Incompatible with Stochastic Primary Ray Effects*: The SVGF filter relies on calculating variance from the primary ray hits (G-Buffer). Due to this, primary rays must avoid noise, therefore making this approach incompatible with effects generated from stochastic primary ray effects like depth of field, motion blur, etc.

Even with these shortcomings the denoising procedure of this technique is the core for the current hybrid implementations and developers have extended it in order to account for these drawbacks.

2.2.2 A-SVGF

A-SVGF [37] introduces an extension to the SVFG filter in order to reduce the temporal lag and ghosting problems. This technique introduces a temporal gradient calculated by reprojecting forward surface samples from the previous frame (Each pixel has a collection of attributes stored in the G-Buffer. It's this collection of attributes that the author denominates as surface samples). The temporal gradient is calculated by taking the difference between the shading on the previous frame and the current frame of the forward projected samples. Thus, the temporal gradient contains an estimation of the rate of changes from the previous frame to the current. With this estimation, the author tunes the α parameter of the exponential moving average, increasing α where the rate of change is low and decreasing if opposite, reducing temporal lag and ghosting artifacts.

2.2.3 Machine Learning Denoising

Machine Learning denoisers using recurrent autoencoders (RAE) networks have been a promising solution for denoising (including global illumination) but are still slow for real time rendering. Although we could not find a direct comparison between SVGF and RAE, the original RAE paper [41] contains a comparison to the original work from Dammertz [38], Edge Avoiding À-Trous Wavelet Transform (EAW), being approximately 5 times slower. The À-trous wavelet transform component dominates the timings in SVGF and as such we can estimate that the performance differences between SVGF and EAW are not large which consequently makes SVGF a lot faster than RAE. In quality terms, the author of A-SVGF provides a good comparison, with A-SVGF generally outperforming or having similar results to A-RAE (extension to RAE with temporal gradient, outperforms RAE) with the SSIM and RMSE metric.

2.2.4 Quake 2 RTX

So far, we have only mentioned academic papers but RTX has been used with success in various games like Battlefield V, Control and most notably Quake 2. Quake 2 [17] has the particularity of being the first game to be fully Path Traced in real time, using A-SVGF as its main component (see Fig. 2.10). Although A-SVGF is the main algorithm used, developers introduce new ideas.

Quake 2 path tracer results are split into 3 channels: direct, indirect specular and indirect diffuse illumination, each denoised separately. The separation of illumination into different channels allows for better adjustment of the denoisers to each illumination phenomena.

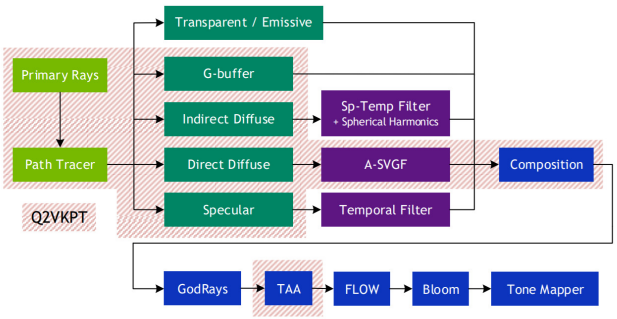


Figure 2.10: Quake 2 pipeline [17]

We focus on the Indirect Diffuse and Indirect Specular channels due to having substantial differences to what has been reviewed so far. Note, all of the temporal components of these channels use the estimated α component of the temporal gradient calculated using A-SVGF.

2.2.4.A Indirect Diffuse

The Indirect Diffuse channel is characterized by the sparsity of samples (bright dots, see Fig **??**). This makes variance almost impossible to estimate either temporally or spatially. As such a simple temporal and spatial filter (EAW) guided by depth and normals are employed by the author. In order to blur over a larger surface, the authors use geometric normals instead of normal maps. Normal maps contain high frequency details that would reduce the amount of light contributions due to the edge avoidance component of the spatial filter (ex. Samples on the lines of the floor would barely contribute to the flat regions). However, not using normal maps essentially means that the high frequency details of the illumination

is lost. To combat this the authors, use spherical harmonics (particularly the 2 first spherical harmonics). Spherical harmonics allows the encoding of illumination combined with its directional information (direction of the light for each sample) and to then be projected onto the normal maps, reconstructing an approximation of the illumination as if we had used the normal maps. In particular, illumination is converted to YCoCg color space with Y(luma) being decomposed into the first 2 spherical harmonics. Furthermore this channel is denoised at 1/3 of resolution due to performance.

2.2.4.B Indirect Specular

Indirect Specular is responsible for effects such as reflections often being referred by this name in the industry. Reflections are characterized by their high frequency being prone to over blurring (As seen by SVGF). Quake 2 simply uses a temporal filter for denoising this component. We believe this is due to the spatial denoising procedure blurring too aggressively and as such being removed.

2.3 Per Effect Solutions

In this section we explore in more detail the solutions to more specific components of global illumination, like the indirect specular component.

2.3.1 Indirect Specular

As seen by Quake 2, the indirect specular channel can be particularly problematic. Other approaches for this effect have been proposed by the community. One such example being PICA PICA reflections [42] (PICA PICA is a hybrid rendering system developed by Electronic Arts). We focus on the spatial reconstruction, temporal accumulation and the bilateral cleanup passes of their indirect specular approach.

2.3.1.A Spatial Reconstruction

Spatial Reconstruction was introduced in Stochastic Screen Space Reflections [18] (SSR). This technique assumes the neighboring pixels shoot useful rays and that the intersection results can be reused, resulting in a gain in performance (Visibility might change but assumed it does not).

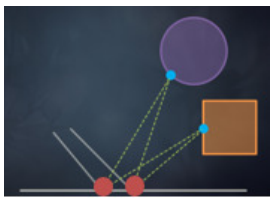


Figure 2.11: Two points (represented in orange) and shading computations using adjacent hit points (represented in blue [18])

Looking at the resolution of the rendering equation (Eq. 1.1) using Monte Carlo with importance sampling (Eq. 2.2), the color of the first orange point can be given by:

$$L_0 = \frac{1}{k}\sum_{i=1}^{k}\frac{f_s \cdot L_i \cdot (\omega_i \cdot n)}{pdf\,(w_i)} \tag{2.5}$$

To note, we use f_s instead of f_r to indicate we are using only the specular component of the BRDF. In order to reuse the hit point from the adjacent second orange point the authors of SSR use f_s of the local sample BRDF (BRDF of first orange point) and the pdf of the original hit point (pdf value of second orange point). However, this can result in spikes in the $\frac{f_s}{pdf}$ ratio resulting in visual quality drops. To combat this the author rewrites the integral in the following similar format with reduced spikes:

$$L_0 = \frac{\frac{1}{k}\sum_{i=1}^{k}\frac{f_s \cdot L_i \cdot (\omega_i \cdot n) \cdot FG}{pdf\,(w_i)}}{W_p} \qquad W_p = \frac{1}{k}\sum_{i=1}^{k}\frac{f_s \cdot L_i \cdot (\omega_i \cdot n)}{pdf\,(w_i)} \tag{2.6}$$

This technique decouples the shading calculations from the rays, allowing ray-tracing to be done on a different resolution to the shading (ex: ray trace at half resolution and calculate shading at full resolution)

2.3.1.B Temporal Accumulation

PICA PICA uses variance color clipping [36] in order to make sure that the colors of the previous frame match the current frame, avoiding temporal lag and ghosting.

2.3.1.C Bilateral Cleanup

To avoid over blurring when using a spatial filter, PICA PICA calculates the specular component variance and uses this information to change the size of the kernel used in the bilateral filter. A low variance indicates that the surface is close to a mirror and as such the kernel size is reduced, and the corresponding to the opposite case. This technique prevents over blurring the reflections result.

2.3.1.D Other Reflection Techniques

Battlefield V [43] is a well renowned case for raytraced reflections being adopted by Unity [31] for their indirect specular system. Their system is similar yet more complex than PICA PICA, including techniques like defragging, variable rate raytracing, screen space hybridization and ray binning, all in order to increase performance.

2.3.2 Shadows

Although it is possible to compute complete direct illumination using path tracing or ray tracing similar to Quake 2, the industry often opts for separating visibility calculations from shading. This separation allows for the rasterization of the shading operation and the raytracing of the visibility calculations (shadows). This allows for shadows to be denoised separately with increased quality, with the disadvantage of each shadow caster requiring its own visibility buffer and denoising pass. This would be extremely expensive for normal RGB data, but visibility can be represented with only one component. This implies 3 visibility buffers operations are roughly equivalent to the performance of a single RGB channel (ex: Indirect Specular). Furthermore, the rasterization of direct illumination produces good quality results and avoids temporal artifacts (ghosting and temporal lag).

To separate shading from visibility calculations it is possible to use (from first integral of Eq.1.14):

$$\int f_r \cdot L_{direct} \cdot G \cdot V \approx \int f_r \cdot L_{direct} \cdot G \cdot \int V \tag{2.7}$$

This formula is an approximation but works well for diffuse surfaces and is commonly used in shadow maps [44]. A more sophisticated technique for decoupling shading and shadows has been proposed in the paper *Combining Analytic Direct Illumination and Stochastic Shadows* [45], but for our purposes, the above approximation is sufficient thus why we choose to present it.

2.3.2.A PICA PICA Shadows

PICA PICA assumes 1 distant light (directional light), calculating visibility by shooting shadow rays in a cone towards the light. The rays shot detect if there is an object blocking the light and the results are calculated accordingly and stored in a visibility buffer. The softness of the shadow (size of the penumbra region) is defined by the angle of the cone. The visibility buffers are then denoised using a similar filter to SVGF, adapted for visibility (replace luminance calculations by visibility). Other improvement is the addition of Variance Color Clipping [36] on the temporal filter, to reduce temporal lag.

2.3.2.B Other Shadow Techniques

Ray Traced Shadows: Maintaining Real-Time Frame Rates [46] describes an interesting algorithm where adaptative sampling is used in order to launch more rays in zones with high temporal variance (Penumbra regions of shadows) and less in areas with low temporal variance (reaching even 0 spp), increasing performance.

Unity [31] separates visibility and shading using the technique proposed in the paper *Combining Analytic Direct Illumination and Stochastic Shadows* [45]. For denoising, a separable bilateral filter is used. Separable bilateral filters are not Math accurate (non-linear filter), but they where found to work in this instance, being a good optimization to consider as a separable filter is easy to implement and more performant. It is worthy to note that using the chosen visibility separation implies evaluating the BRDF for each visibility computation which is a big performance hit versus the simple approximated separation (Eq. 2.7). To avoid high computation times Unity uses a simpler BRDF model. Deferring ray generation to compute shaders has also shown to increase performance.

2.3.3 Indirect Diffuse

Not a lot of solutions exist for the calculation of the indirect diffuse component, apart from brute force path tracing.

PICA PICA indirect diffuse solution "splits" the scene by using surfels. Indirect lighting is calculated by path tracing with just one bounce between the surfels. Over time, this technique eventually converges to results comparable to using more than one bounce of indirect light. This happens due to surfels accumulating shading results over time. As surfels keep updating one another eventually the results produced resemble multiple light bounces similar to path tracing. Furthermore, to avoid temporal lag, new surfels have a larger number of paths allocated, for faster convergence. To compensate the lack of detail caused by the limited resolution provided by the surfels, a color variant of screen space ambient occlusion is used.

Other recent approach used by Unity [31] and proposed in *Real-Time Global Illumination Using Precomputed Light Field Probes* [47] uses lighting probes. This is the solution used by Unity for indirect diffuse and also specular. This technique follows a similar scheme to the PICA PICA solution. Rays are cast around the probes and due to these probes being used to calculate lighting for the surfaces of the scene, eventually the results accumulate and become similar to multi bounce lighting.

Chapter 3

Implementation

3.1 Rendering Pipeline

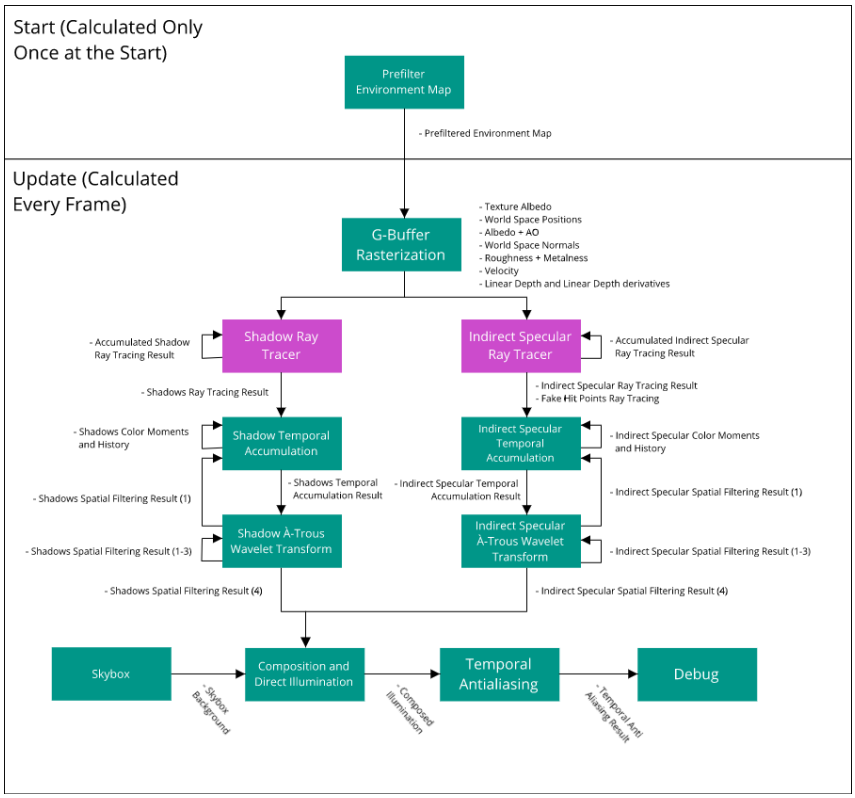


Figure 3.1: Application Architecture

The rendering pipeline can be seen in Figure 3.1. In general, both the indirect specular channel and shadows go through an Svgf filter with some alterations (History Rectification, Adapt Blur to Scene Features, Separable Blur and Reinhard Tone Mapping). Each pass is responsible for the following:

- **Pre Filter Environment Map:** Stage responsible for environment map pre filtering necessary for Image Based Lighting. Calculates a Cube Map with 16 bits per channel in SFLOAT format (Floating point format).
- **G-Buffer Rasterization:** Stage responsible for the creation of the G-Buffer containing the necessary information for the remaining stages. Calculates the following Framebuffers, each with 4 channels:

- **Texture Albedo:** FrameBuffer with 8 bits per channel in SRGB format.
- **World Space Positions:** Contains the World Positions of the currently visible (by the camera) scene objects. FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).
- **Albedo + Ao Contains the albedo:** information in the RGB channel and Ambient Occlusion information in the A channel. FrameBuffer with 8 bits per channel in SRGB format.
- **World Space Normals:** FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Roughness + Metalness:** Contains the Roughness and Metalness of the currently visible (by the camera) scene objects. FrameBuffer with 8 bits per channel in UNORM format (8 bit unsigned normalized).
- **Velocity:** Contains the displacement between the current and previous frame of the visible objects in normalized device coordinates. FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Linear Depth and Linear Depth derivatives:** Contains the Depth and Absolute Depth Derivatives (x and y derivatives). FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).

- **Shadows Ray Tracer:** Responsible for ray tracing shadow rays. Currently, only one shadowed light is supported by the prototype, but this can easily be extended to 4 shadowed lights due to the number of channels available on each Framebuffer. Calculates the following Framebuffers:
  - **Shadows Ray Tracing Result:** Shadow Ray Tracing result. FrameBuffer with 8 bits per channel in UNORM format (8 bit unsigned normalized).
  - **Accumulated Shadow Ray Tracing Result:** Accumulation over multiple frames of the Shadows result. This Framebuffer is used in order to calculate the ground truth results of the indirect specular channel. FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).

- **Shadows Temporal Accumulation:** Temporal Accumulation Filtering for the shadows ray tracing results. Calculates the following Framebuffers:
  - **Shadow Temporal Accumulation Result:** Result of the shadows temporal filtering. Alpha channel contains variance. FrameBuffer with 8 bits per channel in UNORM format (8 bit unsigned normalized).
  - **Shadow Color Moments and History:** The first and second color moment (In the RG channel respectively) of the shadows channel and number of frames successfully reprojected in each image pixel (History, stored on the B channel). FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).

- **Shadows À-Trous Wavelet Transform:** Spatial Filtering for the shadows temporal filtering results. 4 iterations of this pass are used, each one applied over the results of the previous pass. The results of the first iteration are used in the Shadows Temporal Accumulation pass. Calculates the following Framebuffers:
  - **Shadows Spatial Filter Result (1-4):** FrameBuffer with 8 bits per channel in UNORM format (8 bit unsigned normalized).

- **Indirect Specular Ray Tracer:** Responsible for ray tracing secondary light rays (resultant from the importance sampling procedure), calculating the material response to the specular component of the BRDF, and storing the demodulated result (Color result with textures removed) in a Framebuffer. Calculates the following Framebuffers:
  - **Indirect Specular Ray Tracing Result:** Indirect specular demodulated result. FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).
  - **Accumulated Indirect Specular Ray Tracing Result:** Accumulation over multiple frames of the Indirect specular demodulated result. This Framebuffer is used in order to calculate the ground truth results of the indirect specular channel. FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).
  - **Fake Hit Points Ray Tracing:** Contains the hit points of the secondary rays if they were shot through the reflective surface. FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Indirect Specular Temporal Accumulation:** Temporal Accumulation Filtering for the indirect specular ray tracing results. Calculates the following Framebuffers:
  - **Indirect Specular Temporal Accumulation Result:** Result of the indirect specular temporal filtering. Alpha channel contains variance. FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
  - **Indirect Specular Color Moments and History:** The first and second color moment of the indirect specular channel and number of frames successfully reprojected in each image pixel (History). FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Indirect Specular À-Trous Wavelet Transform:** Spatial Filtering for the indirect specular temporal filtering results. 4 iterations of this pass are used, each one applied over the results of the previous pass. The results of the first iteration are used in the Indirect Specular Temporal Accumulation pass. Calculates the following Framebuffers:
  - **Indirect Specular Spatial Filtering Result (1-4):** Indirect Specular Spatial Filtering Result. FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Skybox:** Calculates the background skybox. Computes the following Framebuffers:
  - **Skybox result:** FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).
- **Composition and Direct Illumination:** Calculates the results of the direct illumination using rasterization with a deferred rendering technique and blends the denoised specular and denoised shadows into a final result. Computes the following Framebuffers:
  - **Composed Illumination:** The composed lighting result, containing all the lighting information of the scene. FrameBuffer with 32 bits per channel in SFLOAT format (Floating point format).
- **Temporal Anti Aliasing:** Applies temporal antialiasing to the composed illumination results (We use an adaptation of Falcor temporal antialiasing [26]). Calculates the following Framebuffers:
  - **Temporal Anti Aliasing Result:** Final Anti Aliased Image result. FrameBuffer with 16 bits per channel in SFLOAT format (Floating point format).

- **Debug:** Debug Pass responsible for presenting various frame buffers for debugging purposes.
  - **Final Result:** Final Image result to be presented to the user. FrameBuffer with 8 bits per channel in SRGB format.

3.2 G-Buffer

Most of our passes rely in information present in the G-Buffer. As seen previously, this buffer contains the information regarding the positions, normals and texture information of the objects currently visible by the camera:

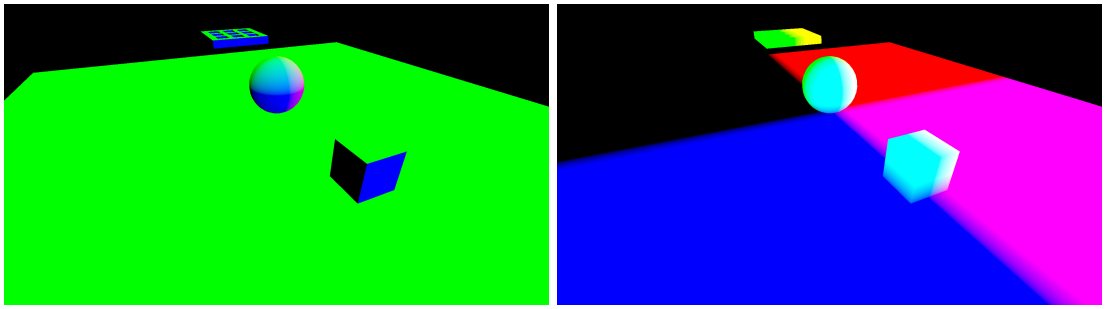


Figure 3.2: G-Buffer contents example (normals and positions)

In order for these buffers to be generated, we iterate through all the primitives present in the scene graph and issue a draw call for each. The geometric and material information is then accessed and stored in the respective frame buffer:

```
//example of the access and output of the albedo texture information
outTextureAlbedo = texture(texSampler,fragTexCoord);
```

The final result of this operation are multiple buffers containing the necessary information for the various procedures of our system (Direct illumination, ray tracing from the G-Buffer, velocity buffer for temporal accumulation).

3.2.1 Velocity Buffer

In order to perform temporal accumulation, a measure of the displacement between the previous and current frame is necessary. This is given by the Velocity Buffer, and is calculated by performing the difference between the current and previous fragment position in normalized device coordinates. The position of the previous fragment is known by storing the model, view and projection matrix from the previous frame and multiplying the vertices (In the vertex shader) by this matrix.

Vertex Shader:

```
//Calculation of the previous fragment position
//(This name is for consistency with the fragment shader,
//but it would be more accurate to call it prev vertex position)
  prevFragPosScreen = ubo.prevProj
                    * ubo.prevView
                    * ubo.prevModel
                    * vec4(inPosition.xyz, 1.0);
```

Fragment Shader:

```
//Velocity
vec3 fragPosNDC = fragPosScreen.xyz / fragPosScreen.w;
vec3 prevFragPosNDC = prevFragPosScreen.xyz / prevFragPosScreen.w;
fragPosNDC = fragPosNDC * 0.5f + 0.5f; // convert from [-1,1] to [0,1] range
prevFragPosNDC = prevFragPosNDC * 0.5f + 0.5f; // convert from [-1,1] to [0,1]

vec2 velocity = fragPosNDC.xy - prevFragPosNDC.xy;
//remove camera jitter introduced by temporal antialiasing
velocity -= vec2(ubo.jitter.x, ubo.jitter.y);
outVelocity = vec4(velocity, 0.0, 0.0);
```

The position of the previous fragment is known by storing the model, view and projection matrix from the previous frame and multiplying the vertices (In the vertex shader) by this matrix:

3.2.2 Linear Depth and Linear Depth Derivatives

In order to calculate the linear depth and linear depth derivatives the following code is used:

```
//Linear Camera Space Depth and Linear Camera Space Depth Derivatives
float linearDepth = fragPosScreen.z * fragPos.w;
float ddxDepth = abs(dFdx(linearDepth));
float ddyDepth = abs(dFdy(linearDepth));
outDepth = vec4(linearDepth, ddxDepth, ddyDepth, gl_FragCoord.z);
```

3.3 Ray Tracing

In this section we describe our implementation regarding the ray tracing component of our system. We start by explaining the necessary buffers containing the information necessary for the shading and the tracing of the scene, followed by an explanation on how our rays are generated and processed.

3.3.1 Ray Tracing Memory Layout

For the Ray tracing component of our system, our scene data (Geometry, materials, etc.) need to be available and placed in the right structures. As such, our program stores each imported primitive into its own Bottom Level Acceleration Structure (Each Bottom Level Acceleration Structure points to a region of memory on the vertex and index buffer). These structures are then instanced using the Top level Acceleration Structure and become ready to be ray traced. The Top Level Acceleration Structure is updated each frame. Vulkan's ray tracing structures contain only the geometric information of the scene. But to calculate the shading of a certain point, the material and texture information are required. This is up to the developer to provide to the ray tracing shaders through buffers and our implementation is defined in the following Figure:

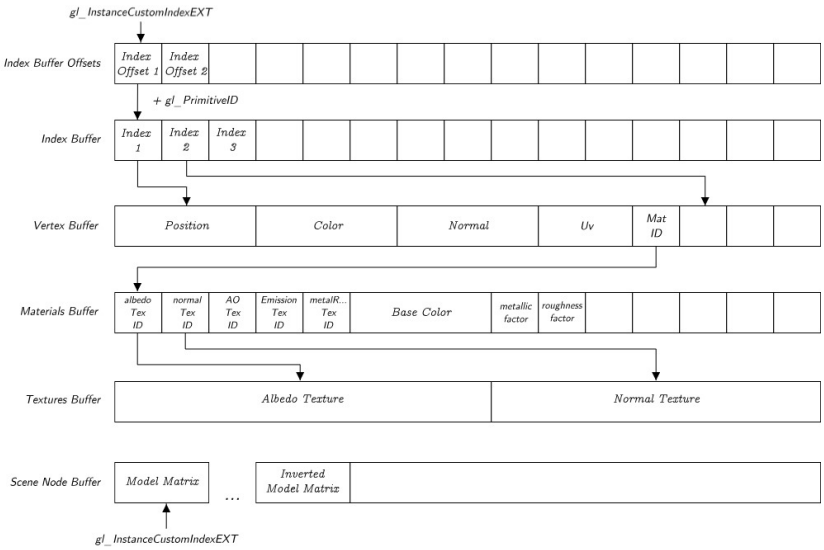


Figure 3.3: Memory layout used for accessing data required for shading

The special buffers, that are different from most rasterization implementations are the Index Buffer Offsets, the Materials Buffer, the Textures Buffer and finally the Scene Node Buffer. The access pattern used in order to access the scene information can be observed in Figure 3.3. Each Bottom Level Acceleration Structure instance has an associated id, accessed using the gl_InstanceCustomIndexEXT variable. This id is used to index a buffer containing the start location of the indices of the particular primitive (Index Buffer Offsets). The variable gl_PrimitiveID (each primitive triangle has a different gl_PrimitiveID) is then used in conjunction with the information from the index buffer offsets in order to access the indices of the primitives. These indices are then used in order to acquire the respective vertices and from there the material information (vertices contain a special ID, Mat ID, that identifies the associated materials), followed by the texture information. Other buffer is accessed using the gl_InstanceCustomIndexEXT variable, being the buffer that contains the information regarding the model and inverted model matrix (Scene Node Buffer). Using this pattern of accesses all the scene information is available in all the ray tracing shaders.

3.3.2 Ray Generation

The ray generation process can be divided in the following steps:

- Data loading - Load the necessary data (ex. normals)
- Pseudo Random Number Generation - Generate a random number used in sampling
- Sampling - Use the pseudo random number generator in order to generate an out going ray direction. (In reflections importance sampling is performed and in shadows uniform cone sampling is performed)
- Ray tracing - Trace the rays towards the out going ray directions
- Compute final result - Calculate and store the final result in a framebuffer

3.3.2.A Pseudo Random Number Generation

A – White Noise

White noise is generated using the TEA algorithm (Tiny Encryption Algorithm), a simple procedure for

random number generation (See code A.2). This procedure is then used for generating the white noise results as follows:

```
//random index for each pixel in screen
uint index = launchIndex.x * uint(1973) +
             launchIndex.y * uint(9277) +
             uint(ubo.frameIndex) * uint(146624);

//Divide by 4294967296.0f to convert from [0, MAXINT] to [0,1] in float format
xi = vec2(blockCipherTEA(index, offset)) / 4294967296.0f;
```

The first step calculates a unique index value for each pixel and frame. This number is then passed to the TEA function that calculates 2 random values used in the sampling operations. An offset parameter that guarantees that for the same index it's possible to generate multiple different randomized numbers.

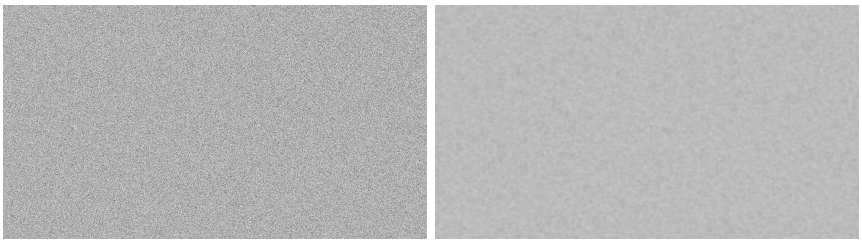


Figure 3.4: Frame containing the method for generating white noise (left) and corresponding blurred result (right)

B – Blue Noise

Blue noise is sampled from 64 textures with 64x64 dimensions (Textures where obtained from [48]). A Cranley Patterson rotation (A offset each frame) is used in order to decorrelate the samples between multiple frames. The random number generator thus becomes:

```
ivec2 uv = ivec2(int(inUV.x * width) % 64, int(inUV.y * height) % 64);

float r1 = fract(texelFetch(blueNoise[frameIndex % 64], uv, 0).x
           + wang_hash_float(frameIndex)); //cranley patterson rotation
                                           //(random offset each frame)

float r2 = fract(texelFetch(blueNoise[frameIndex % 64], uv, 0).y
           + wang_hash_float(frameIndex)); //cranley patterson rotation
                                           //(random offset each frame)

return vec2(r1, r2);
```

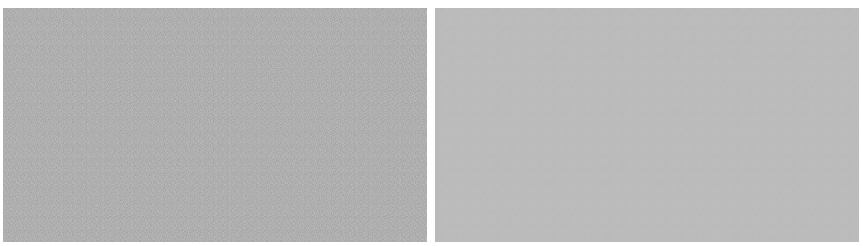


Figure 3.5: Frame containing the method for sampling blue noise (left) and corresponding blurred result (right) As can be seen, using blue noise results in better distributed samples and consequently, when the blur used in reflections is applied, it results in a more homogeneous image than white noise.

3.3.2.B Sampling

In order to generate the ray directions for the indirect specular component of lighting, the importance sampling of the GGX function is used (See Code A.3).

In order to ray trace shadows, directions are randomly sampled from a cone (See code A.4). The directions sampled from the uniformSampleCone function are all relative to the unitary vector (0.0f, 0.0f, 1.0f) and need to be rotated towards the light direction. For this, a coordinate frame is produced from the light direction vector (See code A.5).

3.3.2.C Ray Tracing

Given the ray directions, they can now be traced towards the scene:

```
//Trace shadow ray
traceRayEXT(topLevelAS,
            gl_RayFlagsTerminateOnFirstHitEXT |
            gl_RayFlagsOpaqueEXT |
            gl_RayFlagsSkipClosestHitShaderEXT,
            0xFF, 0, 0, 0, origin, TMIN, lightDir, TMAX, 0);

//Trace reflection ray
traceRayEXT(topLevelAS,
            gl_RayFlagsOpaqueEXT,
            0xff, 0, 0, 0, origin, TMIN, wi, TMAX, 0);
```

As can be observed, the parameters used when tracing the rays are slightly different. For shadows, simply detecting if an intersection occurred is sufficient, and as such, the gl_RayFlagsTerminateOnFirstHitEXT and gl_RayFlagsSkipClosestHitShaderEXT flags are used, as we can terminate our ray on the first surface hit and no calculations are required when the surface is hit. In terms of the indirect specular channel this procedure is more complicated, as we need to perform shading at the surface-ray intersection. This is done by using the memory structures and shading operations already established. Furthermore, shadow rays and other indirect specular rays can be traced from the surface-ray intersection, forming a recursive function that ends on a user defined depth limit (2 are used by default). If no surface is intersected by the ray, the environment map is sampled instead.

3.3.2.D Compute final result

Once the ray tracing is performed, and the results become available, the final color of each respective channel is calculated. In case of the shadows, if a surface has been hit, the final color result becomes 0 (we are in the shadow) and if no surface has been hit the final result becomes 1 (no shadows are present). In case of the indirect specular channel, once the ray tracing results are available, the specular component of the surface is calculated (Surface reaction to the out going ray). These results can then be accumulated over time (when the ground truth are being calculated). Reinhard Tone Mapping is applied over the final ray tracing results improving denoising (Pixels with high brightness cause flickering to be noticeable, similar approach described in [49]). This is done as follows:

```
res /= (1.0f + luma(res) * lumaMultiplier);
```

Reinhard Tone mapping is then ”removed” after denoising. This operation causes a divergence from the ground truth, but removes a lot of flickering (See Figure 3.6).

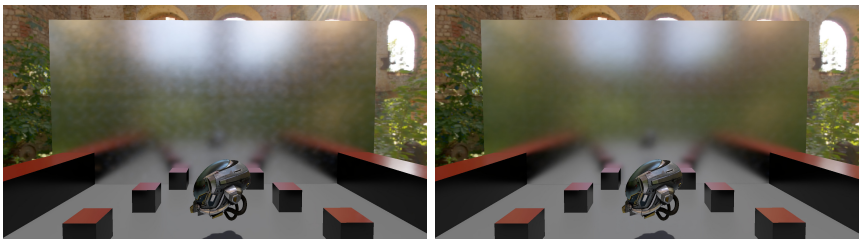


Figure 3.6: Distance Test Scene without Reinhard Tone Mapping and with reinhard Tone Mapping applied. In order to make artifacts more visible, these results where obtained with temporal accumulation deactivated.

Finally, demodulation is applied over the results of the indirect specular channel before being stored, removing texture information:

```
vec3 demodulate(vec3 c, vec3 albedo)
{
    return c / max(albedo, vec3(0.001, 0.001, 0.001));
}

//raygen demodulation
imageStore(image, ivec2(gl_LaunchIDEXT.xy),
                  vec4(demodulate(res, mix(textureAlbedo, vec3(0.0f), sd.metal) +
                                       mix(vec3(F0), textureAlbedo, sd.metal)), 1.0f));
```

This is done in order to avoid blurring high frequency information from the textures during denoising (The texture information is reintroduced after denoising, see section 3.6).

3.4 Temporal Accumulation

Temporal accumulation can be divided in the following steps:

- Calculate previous sample position - Use the velocity buffer in order to backwards reproject the current frame into the previous frame
- Data loading - Load the necessary data from the current and previous frame
- Test Reprojection - Test if the previous sample corresponds to the current frame sample
- History Rectification - Adapt the previous frame sample color to the current sample colors
- Blend previous frame information - Temporally accumulate valid samples.

The code we present is the one used for indirect specular, but in practice the only difference between the indirect specular channel and the shadows channel is that for the shadows the luminance doesn't need to be calculated (Luminance is used in order to reduce a color vector into a single float number, since shadows are represented only by one component of the channel, luminance isn't necessary).

3.4.1 Calculate previous sample position

In order to calculate the previous sample position, the velocity buffer is sampled and the result is subtracted from the current uv position.

```
vec2 prevUv = inUV;
(...)
    prevUv -= texture(velocityBuffer, inUV).xy;
```

This technique, however, doesn't work perfectly on the indirect specular channel. This happens because reflections have their own parallax that isn't properly captured by the common velocity estimation.

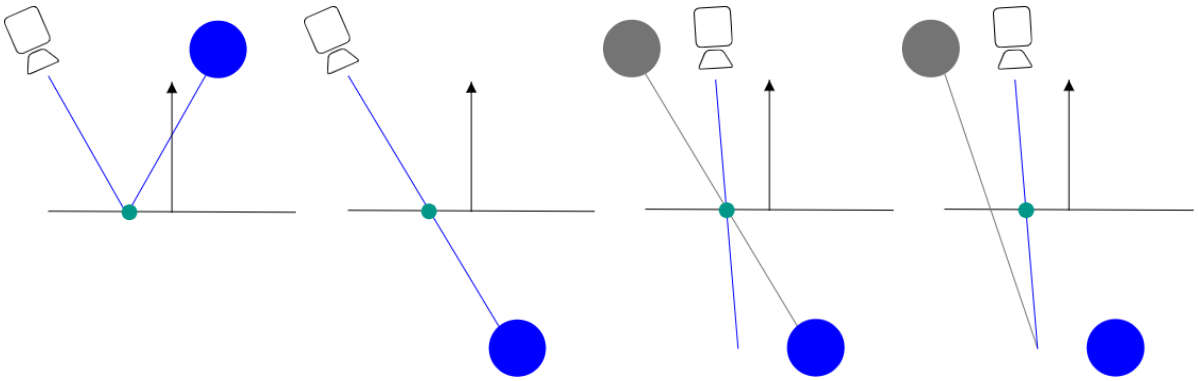


Figure 3.7: Example of the displacement problem caused by reflections. The first image shows the image being captured and the point being temporally accumulated (green point). The second image shows a "simplification" of the problem, where the reflection ray and the object being captured is flipped on the x axis (This scene is equivalent to the previous one). The third image shows the problem with using the velocity estimation of the reflector. These points are correctly matched on the reflector, but they don't correspond to the correct points (Notice how the new blue ray doesn't hit anything, while the previous grey one hits the blue sphere). The fourth image shows the correct solution where the reflection hit points are matched and not the reflector.

In order to estimate the displacement between the reflections hit points, a similar idea to the x axis flip present in Figure 3.7 is used, "fake" hit points being generated in the raygen shader:

```
//Pretend the ray was shot straight through the reflection surface (Used in reprojection)
fakeHitPoint = (rayPayload.dist + length(origin - ubo.camPos.xyz)) * w0;
```

The velocity procedure used on the G-Buffer was then applied to these "fake hit points", estimating the reflections hit point velocity (See code A.6). This, however, ended up not being used by default due to not performing well on curved surfaces and due to reprojections being well handled on reflective surfaces when history rectification is applied.

Figure 3.8: Comparison between temporal accumulation with only the reflectors velocity (normal velocity calculated on the G-Buffer pass), only the velocity calculated from hit points reprojection and only history rectification. Notice how even though the hit points reprojection solve the ghosting on the flat surfaces it introduces problems on curved objects (Helmet).

3.4.2 Test Reprojection

In order to test the validity of the temporal reprojection, depth, normals and mesh ids are tested (See Code A.7). The normals function checks for the similarity between the normal of the pixel being denoised and the neighboring sample. The ids function compares the mesh ids. And finally, the depth function

weights the difference between the current depth and the previous depth weighted by the largest depth derivative. The depth derivative weight guarantees that the inclination of the surface is taken into account (if the surface is inclined we don't want to reject it).

For better temporal accumulation in motion, multiple samples of the previous frame are tested. As stated in the Svgf implementation, the 4 corners of the pixel being evaluated are visited using bilinear taps:

```
//2x2 positions
ivec2 offsets[4] = {
        ivec2(0, 0), ivec2(1, 0), ivec2(0, 1), ivec2(1, 1)
};
```

Each valid color and moment sample (successfully passes the validity tests) is accumulated and weighted into the final result as follows:

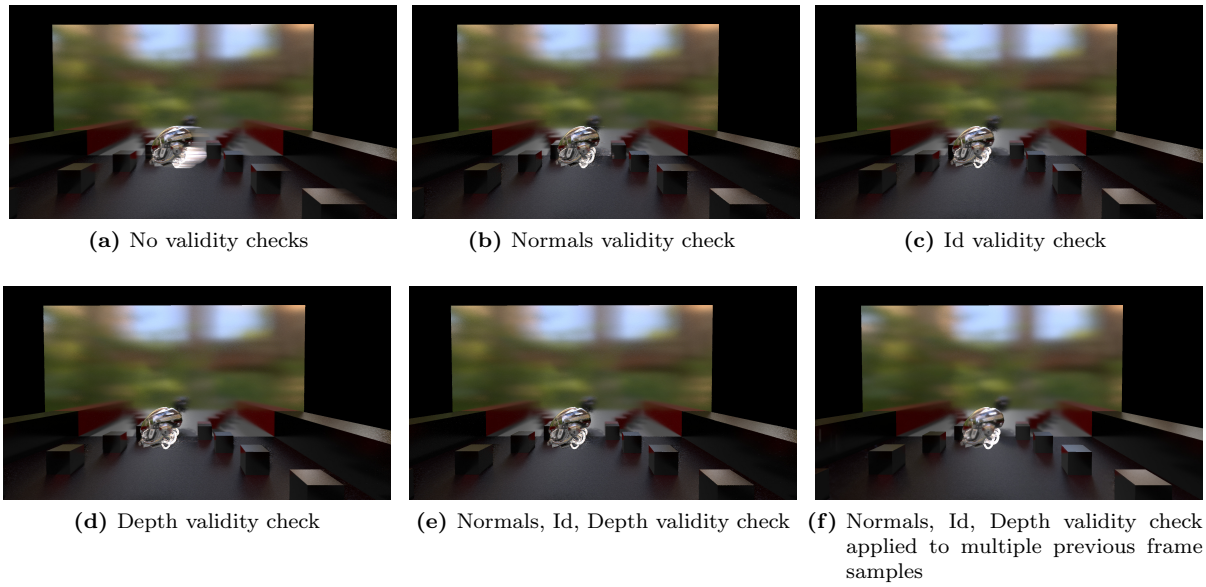
```
//bilinear weights
float x = fract(prevPos.x);
float y = fract(prevPos.y)
const float w[4] = { (1 - x) * (1 - y), x * (1 - y),
                     (1 - x) * y,       x * y };
(...)
for(int i = 0; i < 4; i++) {
        if (v[i]) //if this sample is valid...
        {
                res += w[i] * texelFetch(prevImage, uv, 0).rgb;
                colorMom.xy += w[i] * texelFetch(prevMoments, uv, 0).xy;
                sumw += w[i];
        }
}

if(sumw >= 0.01) {
        res = res / sumw;
        colorMom.xy = colorMom.xy / sumw;
        return true;
}
```

If this procedure fails, the neighbouring pixels are visited (validity tests being performed, see code A.8).

The result is then normalized (each sample has a uniform weight):

```
if(numValidSamples > 0) {
        res /= float(numValidSamples);
        colorMom.xy /= float(numValidSamples);
        return true;
}
```

Only if all samples are invalid, does a sample get rejected and no temporal accumulation is performed (alpha is set to 1.0). The result of this operation corresponds to the previous frame color and color moments.



(a) No validity checks **(b)** Normals validity check **(c)** Id validity check

(d) Depth validity check **(e)** Normals, Id, Depth validity check **(f)** Normals, Id, Depth validity check applied to multiple previous frame samples

Figure 3.9: Comparison between multiple reprojection validity checks. From left to right, the first image is temporal accumulation with no validity checks. As can be observed ghosting is present on the surfaces where disocclusions are present. The next three images contain individual validity tests for only one previous frame sample. The test order is normals, id (Mesh Id) and depth. The fifth image contains the results of all 3 reprojection tests and the final image contains the result of the validity checks applied to multiple previous frame samples. As can be observed, temporal accumulation improves on the regions where disocclusions are present.

3.4.2.A History Rectification

In order to make sure that the previous sample color is consistent with the current frame color, history rectification is used. Two approaches where tested, variance color clipping and variance color clamping. The first step of both approaches is the calculation of the color variance and mean on the current frame pixel neighborhood. These values are then used to create a "bounding box" (represented by colorMin and colorMax), which represents the values we expect the previous frame samples to conform too (See code A.9). Using these values, either clamping (variance color clamping) or clipping (variance color clipping) are used.

```
if(params.useClipping) //clipping
    return YCgCoToRGB(clipAABB(prevColor, ...));
else { //clamping
    return YCgCoToRGB(clamp(prevColor, colorMin, colorMax));
}
```

The clipping operation is based on the temporal antialiasing solution created by Play Dead Games ([50], see code A.10).

When comparing both approaches we ended up choosing variance color clamping for the final approach. This option is the faster and both results are extremely similar.

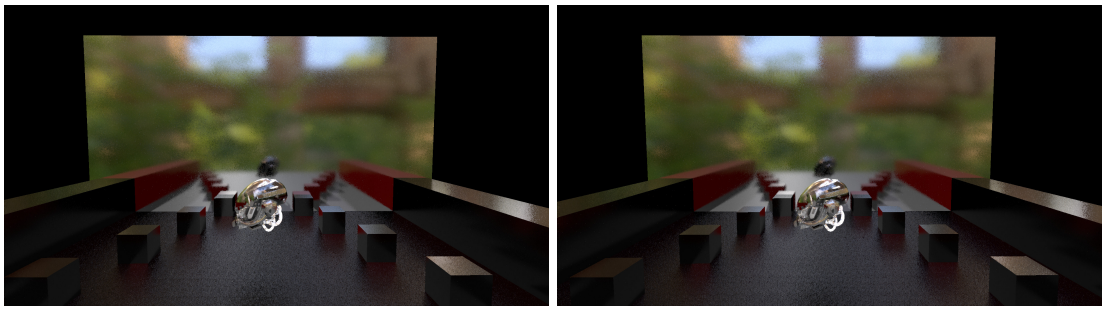


Figure 3.10: Comparison between color clipping and clamping

3.4.3 Blend previous frame information

Once the valid and invalid samples are determined, the color and moments can be temporally integrated using an exponential moving average:

```
//Temporally accumulate luminance moments
float currLum = luminance(currColor);
outMoments =  params.momentsAlpha * vec4(currLum, currLum * currLum, 0.0f, 0.0f)
               + (1.0 - params.momentsAlpha) * prevMom;
(...)
//Temporally accumulate color
vec3 result = alpha * currColor + (1.0 - alpha) * prevColor;
```

Different alphas are used depending if the camera is moving or not. The objective is to rely less on temporal information when the camera is moving and presumably the information is less reliable. The opposite is expected when the camera is still. The default values used for the alpha are 0.05 when the camera is static and 0.2 when the camera is moving.

3.5 Edge-avoiding À-trous Wavelet Transform

This shader is responsible for the spatial denoising procedure which is applied over the results of the temporal accumulation. This procedure is performed 4 times for both the shadows and reflections while vising progressively further pixels (controlled by the blurScale variable). It is defined by the following steps:

- Adapt Blur to Geometry Features
- Data Loading - Load the necessary information
- Blurring Procedure - Calculate the final pixel color from the neighbour pixels information. Edge detection functions are used in order to avoid accumulating information from pixels that are too distinct from the target

3.5.1 Adapt Blur to Scene Features

Denoising using SVGF becomes insuficient when history rectification is used. As such, and due to the fact that different surface roughnesses produce various degrees of noise (low surface roughnesses produce a reduced amount of noise and high surface roughnesses produce a high amount of noise), we opted for changing the blur scale depending on the surface roughness. This is similarly done for shadows, but with the shadow angle:

```
//Indirect Specular Channel
if(roughness < 0.2)
    // << operator is equivalent to doing pow(params.blurScale,2)
        blurScale =  1 << params.blurScale;
//larger blur scale for high roughness surfaces
else
        blurScale = 1 << min((params.blurScale + 1), 4);

//Shadow Channel
if(params.shadowAngle > 6.0f)
        blurScale =  1 << min((params.blurScale + 1), 3);
else
        blurScale =  1 << params.blurScale;
```

3.5.2 Blurring Procedure

In order to calculate the final pixel color and variance, the neighbouring pixel information (5x5 kernel) is used. In order to improve the performance of this procedure, the 5x5 kernel is separated into a vertical and horizontal pass (Separable Blur). This separation reduces the amount of pixels visited (The 5x5 kernel is separated into two 5x1 kernels, resulting in 10 visits per blur pass compared to the original 25 neighborhood visits) (See code A.11).

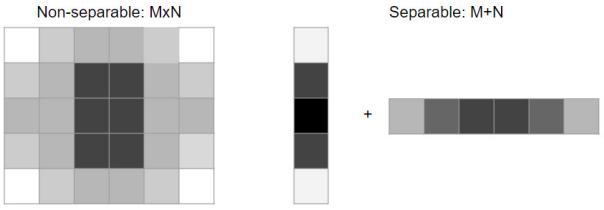


Figure 3.11: Comparison between a full kernel vs the equivalent, separated blur kernel

In terms of comparison to the ground truth, the results remain similar, although an increase in the amount of blur can be noticed, increasing some banding artifacts. This can also be a positive improvement in some regions with insufficient blur. The cause of the increase in blur is due to the variance only being updated in the second pass of the separable blur (vertical pass). Due to the positive improvement in regions with insufficient blurring we opted for continuing with this approach. After accumulating the results from the pixels neighborhood, the final color and variance is normalized:

```
sumColor /= weightSum;
varSum /= (weightSum * weightSum);
```

3.5.3 Edge Stopping Functions

The edge stopping functions used are defined by the following (See code A.12):

- **Normal weight function:** controls the impact of the normals into the blurring procedure. The params.reflectionNormalSigma controls the impact of this function. A value of 128.0 is used.
- **Luminance weight function:** controls the impact of the pixels luminance into the blurring procedure. The params.reflectionsLuminanceSigma controls the impact of this function. A value of 10.0 is used. Furthermore, variance is also introduced into this function in order to control the blurring procedure depending on the amount of variance (and expected noise) present. The larger the variance, the larger the value of this function and consequently, the more influence the neighbouring pixels have onto the pixel being denoised.
- **Depth weight function:** controls the impact of the pixels depth into the blurring procedure. The params.depthSigma controls the impact of this function. A value of 0.5 is used. This parameter takes into account the depth derivatives, and consequently the inclination of the surface into account (Samples on a steep, uniform incline need to be accumulated and not rejected).

3.6 Composition and Direct Illumination

Once both shadows and the indirect specular channel are denoised, the results can be composed and direct illumination can be calculated. This pass can be defined in the following steps:

- Data loading - Load the necessary data for shading calculations
- Direct Illumination - Calculate the direct illumination using the buffers provided by the G-Buffer and the shading procedures (See A.1). The only particularity is the usage of the denoised shadows buffer. Each pixel in this buffer is multiplied by the final result of the direct illumination calculations, effectively applying the shadows to the scene.
- Indirect Illumination - Calculate the indirect illumination
- Tone Mapping - The shading calculations are in the $[0,\infty[$ range, but they need to be passed to the [0,1] display range. This is done using a simple tone mapping method ([51])

3.6.1 Indirect Illumination

In this passage, the programs starts by calculating the indirect diffuse component, which on our program is represented by a constant illumination value provided by the user, effectively being similar to a ambient term (Our program does not implement the indirect diffuse component and has such, needs to be represented with a "fake" parameter):

```
vec3 diffuse = mix(albedo, vec3(0.0f), metal)
               * globalShaderInputs.indirectDiffuseWeight;
```

This is followed by the calculation of the indirect specular component, which is calculated by sampling the respective denoised channel and applying the inverse operations of demodulation and reinhard tone mapping:

```
//Sample the denoised indirect specular channel and reapply the textures
 specular = texture(reflectionsSampler, inUV).xyz
            * (mix(textureAlbedo, vec3(0.0f), metal)
               + mix(vec3(F0), textureAlbedo, metal));
(...)
//"remove" the reinhard tone mapping operation used while denoising
 specular = specular * (1.0f + luma(specular))
            * globalShaderInputs.reinhardToneMappingWeight;
```

3.6.2 Image Based Lighting

Low surface roughness, should produce very dense noise and as such require a reduced amount of blurring. This is, however, not the case, has the subsequent ray bounces have no guarantees in regards to noise (The objects being reflected may be rough and as such produce sparse noise that requires a lot of spatial filtering in order to be denoised). Due to this property, we experimented with using an image based lighting technique in order to calculate the shading for the second ray intersections (Using this technique [5]), guaranteeing that no noise is present. This then allows us to change the number of blur iterations depending on the surface roughness, reducing over blurring on low surface roughnesses:

Figure 3.12: Comparison between not using Image Based Lighting for secondary ray intersections and using Image Based Lighting. Notice how in the second image the reflections aren't as blurred.

```
//Adapt the number of iterations to the amount of noise we expect
if(roughness <= 0.0) {
        maxBlurIteration = 0;
}
else if(roughness <= 0.05)
        maxBlurIteration = 1;
else
        maxBlurIteration = 4;
```

However, this technique is deactivated for the remaining of the document due to deviating the results from the ground truth, nevertheless, it showcases how other rasterization techniques like lighting probes could be used in order to improve denoising, by guaranteeing a noise free second ray intersection. Another important point to mention is that this strategy also affects the denoising of shadows in reflections (high shadow angles visible through reflections produce a lot of noise) and as such would also need an alternative solution.

Chapter 4

Evaluation Methodology

This master thesis objective is the creation of a Hybrid Renderer capable of producing realistic images with a performance of at least 30 frames per second. As such, both quality and performance are evaluated.

4.1 Hardware

The tests where performed on a machine containing an Intel(R) Core(TM) i7-4770K with a base clock speed of 3.5 GHz, 24GB of RAM and with a GPU ASUS NVIDIA Geforce RTX 2080 graphics card. The IDE used was visual studio 2019, and the performance tests where collected using nvidea nsight 2020.3. The vulkan version used was 1.2.162.0.

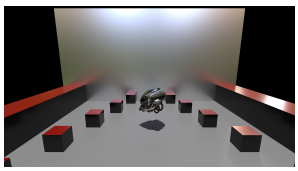
4.2 Image comparison metrics

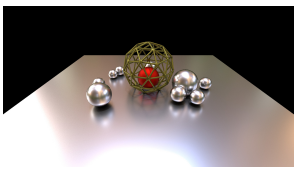
In order to compare image pairs Structural Similarity Index Measure (SSIM) was used.

4.2.1 Structural Similarity Index Measure (SSIM)

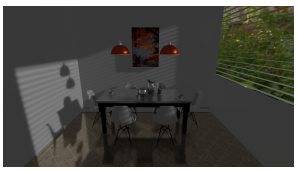
Structural Similarity Index Measure computes the perceived difference between 2 images. It does such, by evaluating not just the difference between each pixel pair, but by comparing perceptual metrics like luminance and contrast. Furthermore, the analysis is done not at a pixel level, but at a small window level. The results from this operation are given in a range of 0 to 1, where 1 means the images are equal and 0 the opposite.

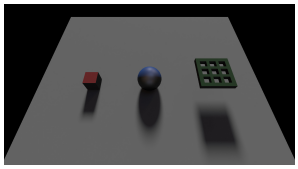
4.3 Scenes

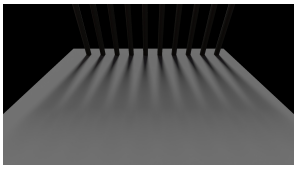

(a) Cubes distance scene

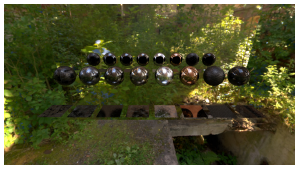

(b) Icosphere scene


(c) Glossy Sponza scene


(d) Breakfast Room scene


(e) Shadow Objects scene


(f) Pillars scene


(g) Metals scene

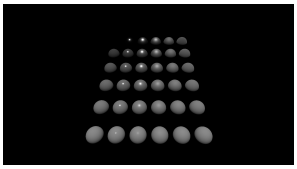

(h) Spheres Shading scene

Figure 4.1: Scenes used for testing.

In order to evaluate our system, we create multiple scenes with different geometry and materials.

4.3.1 Test Scenes

1. Cubes distance scene
   (a) **Description:** Large mirror scene, with multiple simple objects progressively further away from the mirror object
2. Icosphere scene
   (a) **Description:** Multiple spheres scene with a "wireframe" icosphere
3. Glossy Sponza scene
   (a) **Description:** Complex scene with multiple materials tuned to be smoother (reduced roughness). This modification was done in order to boost the indirect specular component.
4. Breakfast Room scene
   (a) **Description:** Interior scene with multiple materials tuned to be smoother
5. Shadow Objects Scene
   (a) **Description:** Multiple large and simple objects
6. Pillars Scene
   (a) **Description:** Multiple narrow pillars objects
7. Metals Scene
   (a) **Description:** Multiple textured metals
8. Spheres Shading Scene
   (a) **Description:** Multiple spheres arranged in order of roughness and metalness

Scenes	Number of Materials	Number of Vertices
Cubes distance scene	4	14664
Icosphere scene	5	80164
Glossy Sponza scene	26	655.238
Glossy Breakfast Room scene	16	1.222.490
Pillars Scene	2	1292
Shadow Objects Scene	4	626
Metals Scene	10	3892
Spheres shading scene	36	626

Table 4.1: Geometry and Materials of each scene

4.4 Ground Truth

Ground truth images calculated by our system are created by accumulating 10.000 samples in both reflections and shadows channel. Our reflections ray tracer has a depth of 2 and the images calculated have a resolution of 1920x1057. The indirect diffuse component is disabled.

4.5 Testing Structure

The tests are divided in three sections, Proof of Correctness, reflections scenes evaluation and reflections shadows evaluation:

4.5.1 Proof of Correctness

In order to prove that our system is correctly implemented we do the following comparisons:

- **Shading:** Comparison between our system ground truth and Falcor ground truth (Same conditions as the ones used by our path tracer). The tests performed are the following:
  - **Spheres Shading Scene Direct Illumination Test:** Analysis of how our system handles direct illumination with different roughnesses and metalness.
  - **Metals Scene Direct Illumination Test:** Analysis of how our system handles direct illumination with different textured metallic surfaces.
  - **Metals Scene Full Illumination Test:** Analysis of how our system handles global illumination with different textured metallic surfaces. The scene contains a large perfect mirror, in order to compare the shading operations performed on the second bounce of indirect specular light.
- **Reflections Importance Sampling:** Comparison between our system ground truth and Falcor ground truth (Same conditions as the ones used by our path tracer). The test is performed on the Metals scene, where the ground truth results are compared over multiple surface roughnesses for the large mirror in the scene. This allows the verification of the importance sampling procedure used.
- **Shadows:** Comparison between our system ground truth and Blender Cycles ground truth (Same conditions as the ones used by our path tracer). The test is performed against the Blender system due to Falcor not possessing a "sun" light type. The test is performed on the Shadows Object scene, where the ground truth results are compared over multiple shadow angles. This allows the verification of the shadows ray tracing procedure.

- **Svgf:** Comparison between the SSIM metrics of our Svgf implementation and the respective ground truth and Falcor Svgf results and the respective ground truth. This test is done over multiple scenes [Cubes Distance scene, Icosphere scene, Glossy Sponza scene]. The objective is the verification of our Svgf implementations.

4.5.2 Reflections scenes evaluation

Analysis on how our system works on diverse conditions over different scenes in both quality and performance. The following parameters are analysed:

- **Surface Roughness:** Surface roughness of one or multiple materials is altered. The roughnesses values used are the following [0.0, 0.1, 0.2, 0.3, 0.4, 0.6, 1.0]. Lower roughnesses present a more concentrated number of surface roughness samples, due to presenting more interesting phenomena (Over 0.6 roughness reflections stop being visible and the visual results become similar to a diffuse term).
- **Type of movement:** Our system is tested against multiple types of movement [Static (No movement), Moving camera, Moving Lights and Objects].
- **Denoising improvements:** Sequential improvements to our implementation of SVGF. The improvements showcased are the following [Our Svgf (baseline algorithm), History Rectification, Blur adapted to Surface Roughness, Separable Blur, Reinhard Tone Mapping]. These techniques are built on top of one another and the results are progressively analysed.

The scenes analysed and the performed tests are the following:

- **Cubes Distance Scene Test:** Test the sharpness of objects at different distances from the reflective surface (closer objects should be sharper). **Surface roughness**, **type of movement** and **denoising improvements** are analysed.
- **Icosphere Scene Test:** Analyse how reflections behave on curved objects and finer geometry. **Surface roughness**, **type of movement** and **denoising improvements** are analysed.
- **Glossy Sponza Scene Test:** Analyse how reflections behave on more complex environments with textures and high frequency normal maps ("rough" normal maps with a lot of detail). **Type of movement** (light and object movement aren't analysed due to the low impact derived from the presence of textures) and **denoising improvements** are analysed.

4.5.3 Shadows scenes evaluation

Similar to what is done in reflections, an analysis is performed over diverse conditions and scenes, in both quality and performance. The parameters and situations evaluated are [**Shadow Angle**, **Type of movement**, **Techniques Used**]. The type of movement is equal to that of reflections, with exception to the movement of the camera, due to presenting almost no differences to the static case. The rest of the test conditions are defined as follows:

- **Shadow Angle:** The shadow angle of the light source is altered, causing the size of the penumbra regions to vary. The shadow angle values used are the following [0.0, 2.0, 4.0, 6.0, 10.0, 20.0, 40.0, 90.0]. Furthermore, the artifacts caused by the larger shadow angles are similar between one another.

- **Denoising improvements:** Sequential improvements to our implementation of SVGF. The improvements showcased are the following [Our Svgf (baseline algorithm), History Rectification, Blur adapted to Shadow Angle, Separable Blur]. These techniques are built on top of one another and the results are progressively analysed.

The scenes analysed and the performed tests are the following:

- **Shadow Object Scene Test:** Test shadows with large and simple objects. **Shadow angle**, **type of movement** and **denoising improvements** where analysed.
- **Pillars Scene Test:** Test shadows produced by narrow objects. **Shadow angle**, **type of movement** and **denoising improvements** where analysed.
- **Glossy Breakfast Room Scene Test:** Test shadows in more complex environments. **Shadow angle**, **type of movement** and **denoising improvements** where analysed.

4.5.4 Evaluation Format

The parameters of both shadows and reflections were evaluated using a quality and performance table. In the quality table, each line represents a new technique (From the Denoising Improvements) and the columns are composed by an image of the scene, close up images and SSIM graphics where the x axis represents surface roughness or shadow angle and the y axis the respective quality value. These graphs contain the plot of the current table line, the previous table line and SVGF (baseline), making it simple to understand how each technique impacts quality.

The performance table follows a similar approach to the quality table, with each line representing a new technique. The columns contain 2 different graphics with the first column containing a graph where the x axis represents the surface roughness or shadow angle and y axis the time in ms each pass took (Ray tracing, Temporal Accumulation, Edge-Avoiding À-Trous Wavelet Transform). The second and final column contains a comparison between the current table line and the previous table line, offering a view on how each technique affects timings when compared to the previous iteration. In order to further complement the performance metrics we present a table containing the average, minimum and maximum frames per second on an animation sequence in each of the scene.

Due to Sponza receiving the roughness from textures and due to the fact that the Sponza scene contains lot of different scenarios, this scene is evaluated differently with the SSIM graphics and timing graphics being replaced by tables where each entry represents a new technique and the columns a different scene inside Sponza.

Chapter 5

Results

5.1 Proof Of Correctness

5.1.1 Shading

Test	Shading Scene Direct Illumination Test	Metals Scene Direct Illumination Test	Metals Scene Full Illumination Test
SSIM	0.999	0.993	0.812

Table 5.1: Shading Tests

As can be observed by metrics, our ground truth results are very similar to the ground truth results of Falcor. The only exception is on the Metals Scene Full Illumination Test when the environment map is used. This can be explained by the differences in sampling between our method of sampling the environment map and the one used by Falcor (We sample the enviroment map directly, while Falcor interpolates the results of the 4 texel corners). Furthermore, there is a small offset on the background environment maps further causing the divergence when comparing both systems. By computing the difference between the images, these artifacts become visible (See Figure 5.1).

(a) Falcor Result **(b)** Our Result **(c)** Shading Image Differences

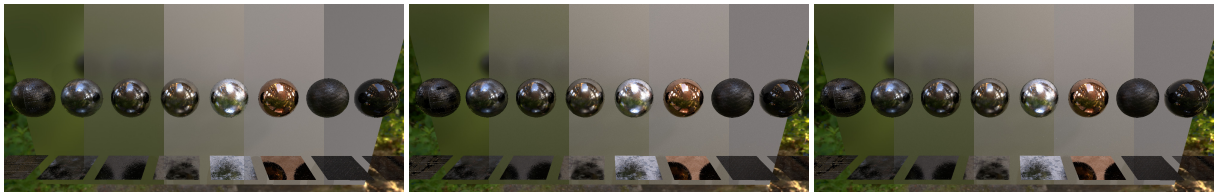
Figure 5.1: Differences between Falcor ground truth and our ground truth

5.1.2 Reflections Importance Sampling

To prove the correctness of our importance sampling method and random number generation we use the Metals Scene, and progressively change the roughness of the present "mirror" object.

Surface Roughness	0.0	0.2	0.4	0.6	0.8	1.0
White Noise	0.853	0.842	0.828	0.824	0.829	0.835
Blue Noise	0.853	0.842	0.827	0.826	0.828	0.835

Table 5.2: Result of the ground truth calculations on a scene with multiple roughness levels (SSIM)



(a) Falcor Result **(b)** White Noise Result **(c)** Blue Noise Result

Figure 5.2: Ground truth result comparison between Falcor, our white noise result and our blue noise result. Each "stripe" represents a level of roughness with the following values in order [0.0, 0.2, 0.4, 0.6, 0.8, 1.0]

As can observed, the metric results are very similar to the previous shading test (**Metals Scene Full Illumination Test**), and suffer from similar problems to those previously seen. If the results are compared according to the expected error it can be seen that they remain consistent. Furthermore, by direct observation we can attest that the results are similar, proving that our importance sampling, and consequently our reflections ray tracer is working as expected.

5.1.3 Shadows

In order to prove the correctness of our shadows, the ground truth results obtained from the shadow ray tracer are compared to the results obtained from Blender cycles. This evaluation is done by direct observation since our system shading model isn't exactly equal to that of blender. This test is performed using a simple test scene designed for shadows, and different shadow angles are analysed [4.0, 10.0, 20.0].

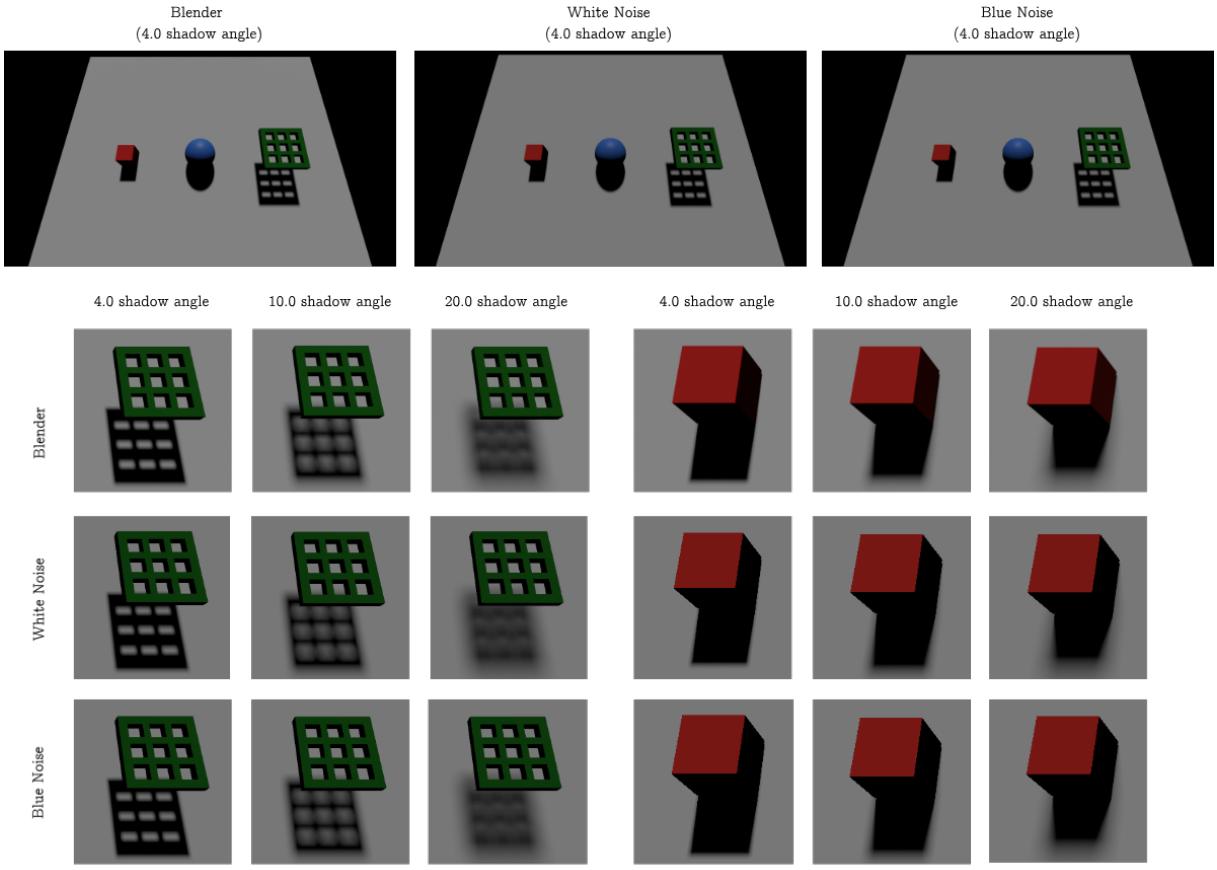
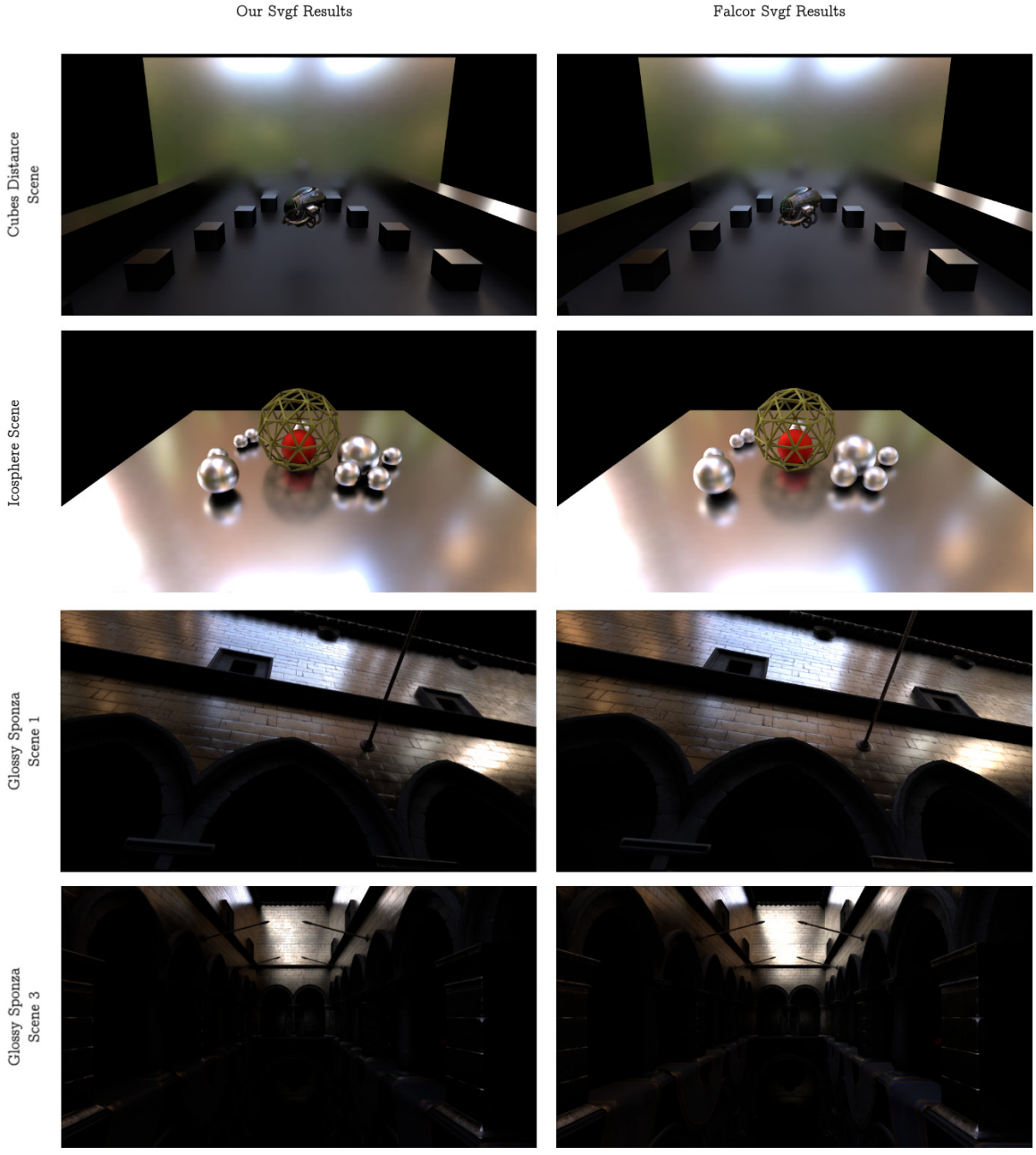


Figure 5.3: Shadow Test

As can be observed the results are very similar between our system and the results obtained from blender, letting us conclude that our shadows ground truth are correct and according to expectations.

5.1.4 Svgf

Our Svgf and Falcor Svgf implementations are compared. Only the indirect specular channel is analysed, as Falcor does not possess a "sun" light type. However, as both implementations are very similar it is possible to extrapolate the correctness of shadows from the indirect specular Svgf denoising procedure.



Surface Roughness	0.0	0.1	0.2	0.3	0.4	0.6	1.0
Cubes distance scene	0.9459	0.9782	0.9717	0.9679	0.9664	0.9665	0.9677
Icosphere scene	0.9022	0.9539	0.9592	0.9597	0.9585	0.9396	0.9374

Table 5.3: Our Implementation (SSIM)

Surface Roughness	0.0	0.1	0.2	0.3	0.4	0.6	1.0
Cubes distance scene	0.9367	0.9769	0.975	0.9734	0.9693	0.9695	0.9693
Icosphere scene	0.9337	0.9462	0.956	0.9571	0.9573	0.9452	0.935

Table 5.4: Falcor Implementation (SSIM)

Surface Roughness	Result
Glossy Sponza Scene 1	0.7956
Glossy Sponza Scene 2	0.8935
Glossy Sponza Scene 3	0.9007

(a) Our Implementation Sponza (SSIM)

Surface Roughness	Result
Glossy Sponza Scene 1	0.761
Glossy Sponza Scene 2	0.84922
Glossy Sponza Scene 3	0.86874

(b) Falcor Implementation Sponza (SSIM)

As can be observed the results are very similar between both implementations by both direct observation and metrics. The exceptions are the 0.0 surface roughness for the Distance Test, Icosphere and for the Sponza scene. The 0.0 surface roughness divergence is caused by the difference in sampling the environment map.

(a) Our 0.0 surface roughness Icosphere Ground Truth

(b) Falcor 0.0 surface roughness Icosphere Ground Truth

Figure 5.4: Comparison between our ground truth and Falcor's ground truth. Notice how the skybox reflection seems slightly blurrier in Falcor's version when compared to ours.

The Sponza divergence is caused due to our ray tracer producing darker ground truth images when compared to falcor. The exact reasoning for this divergence could not be found. The problem seems to only occur when textures are present, as can be seen by the remaining metrics with texture-less data (Icosphere scene and Cubes distance scene).

(a) Our Glossy Sponza Ground Truth

(b) Falcor Glossy Sponza Ground Truth

Figure 5.5: Comparison between our ground truth and Falcor's ground truth. As can be observed our result is slightly darker than the ones produced by Falcor

5.2 Quality Metrics

Static Camera

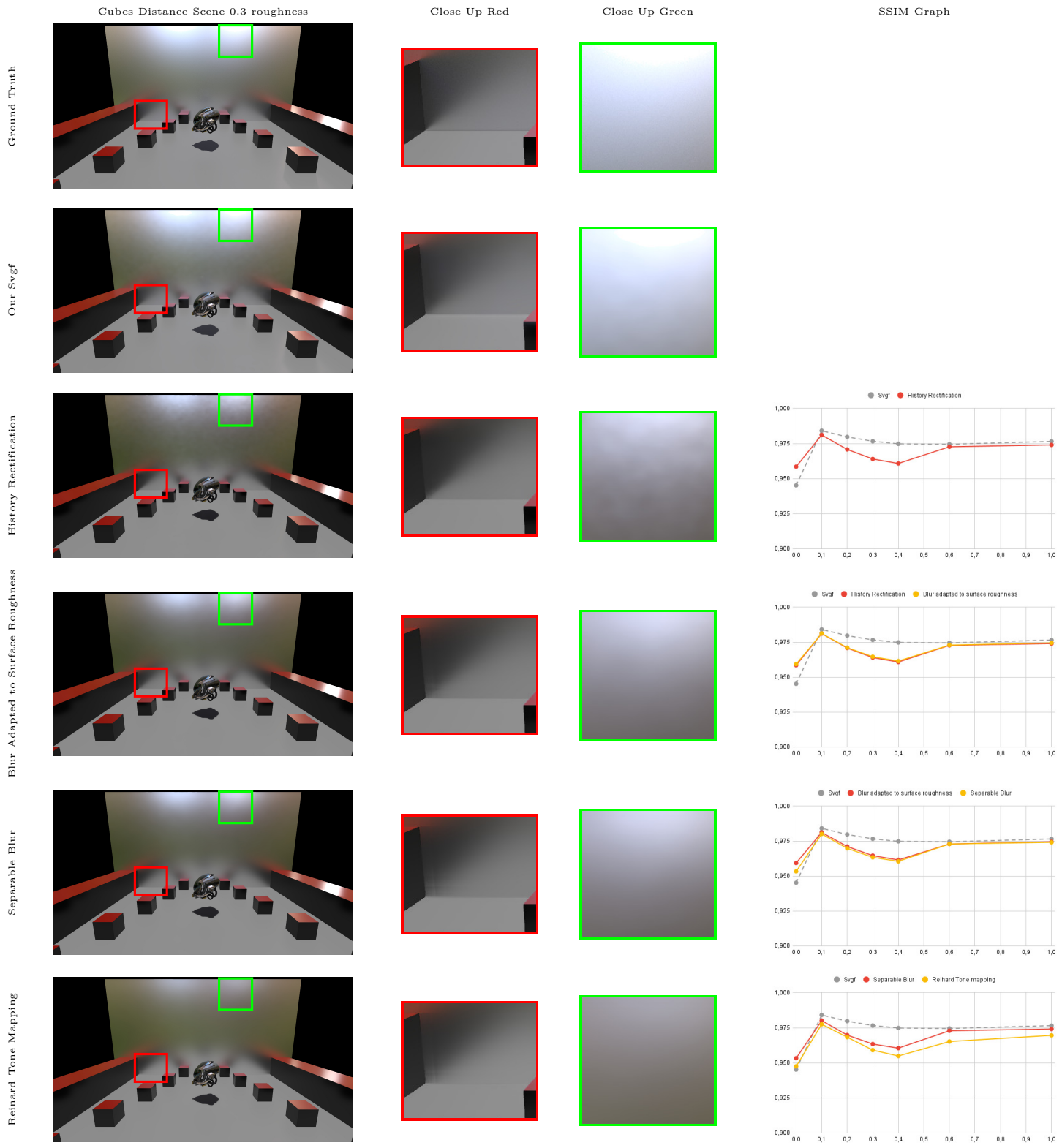


Table 5.5: Static camera, cubes distance quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.

Moving Camera

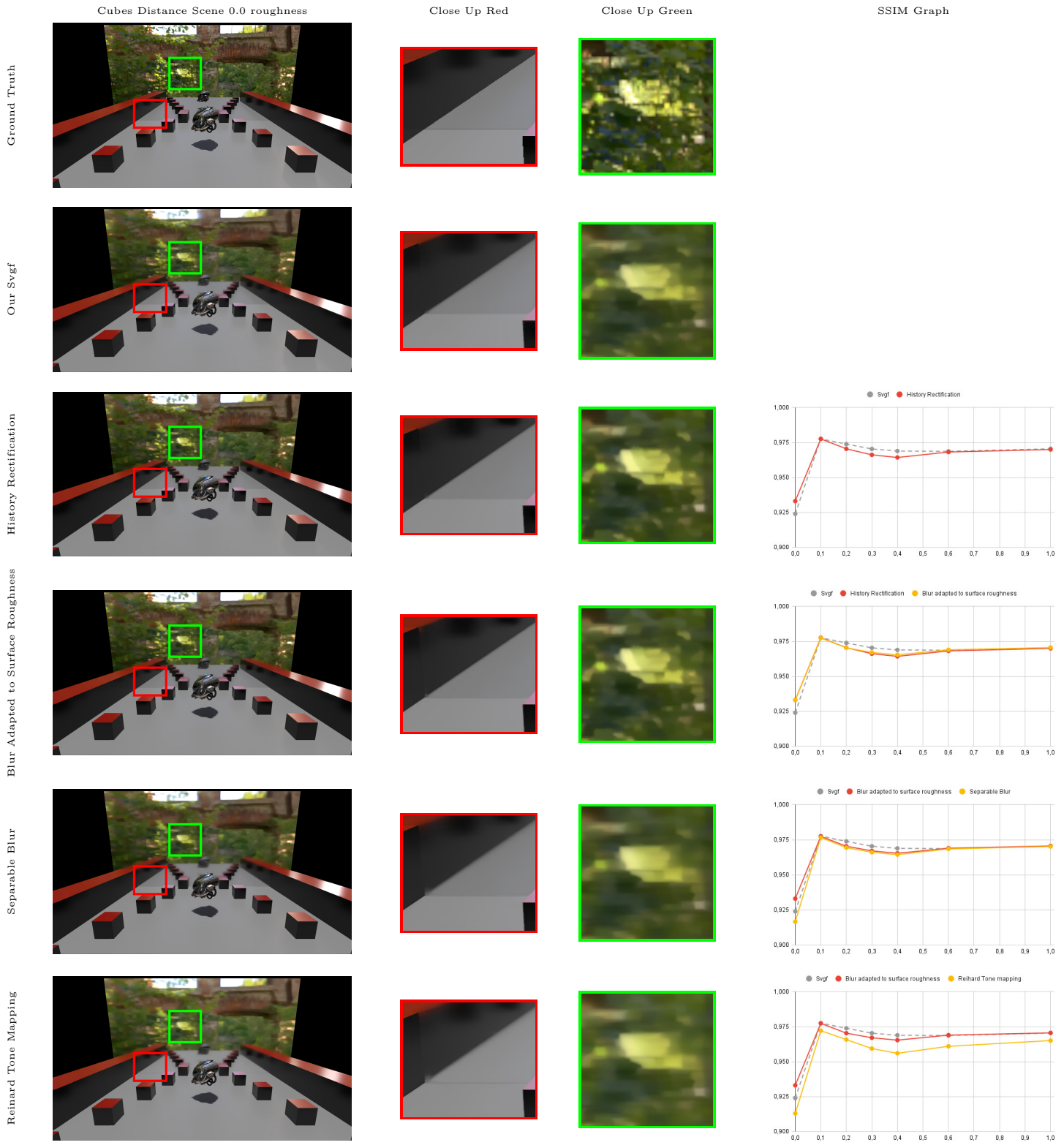


Table 5.6: Moving camera, cubes distance quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.

Moving Objects and Light

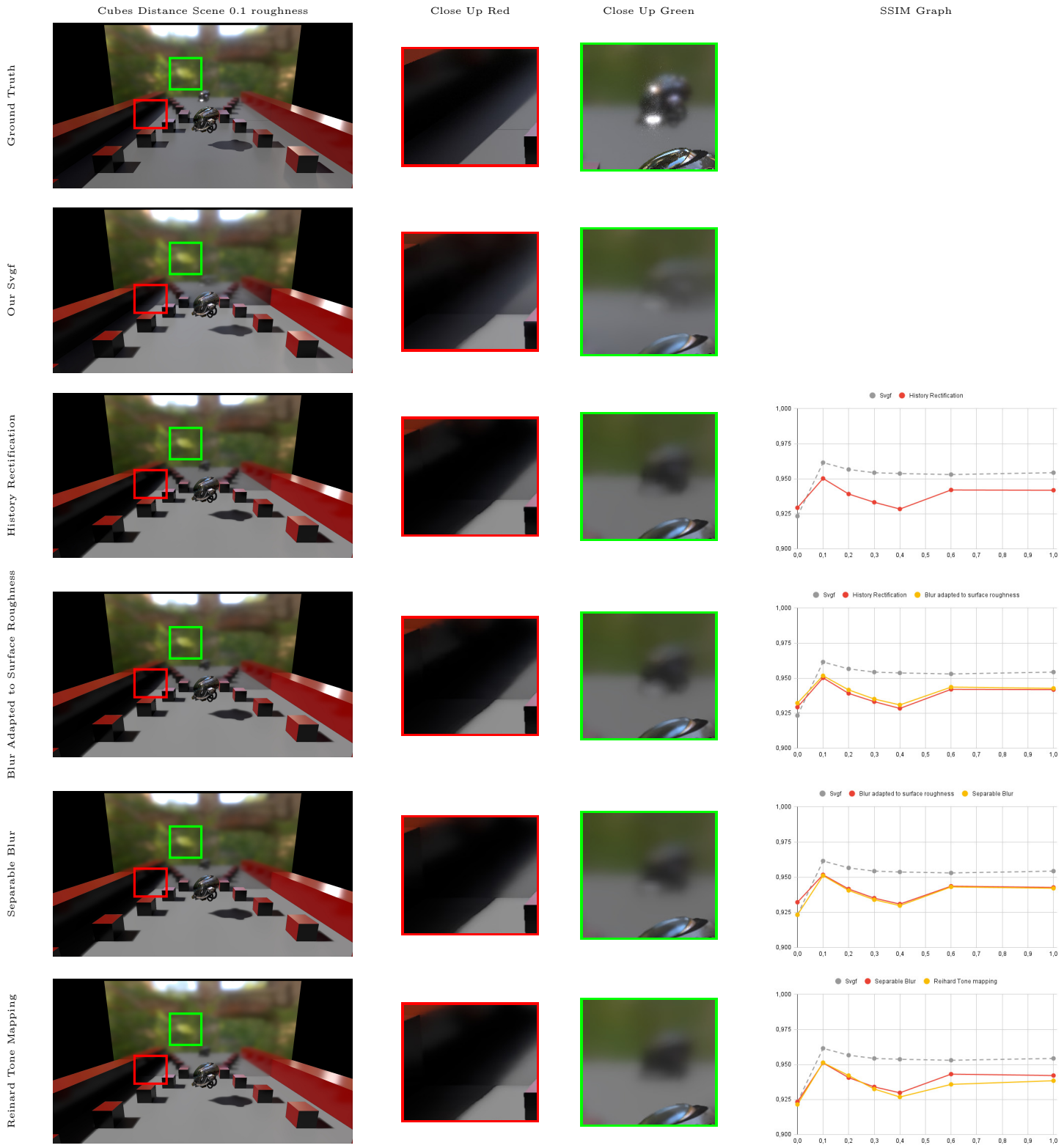


Table 5.7: Moving objects and light, cubes distance quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.

Static Camera

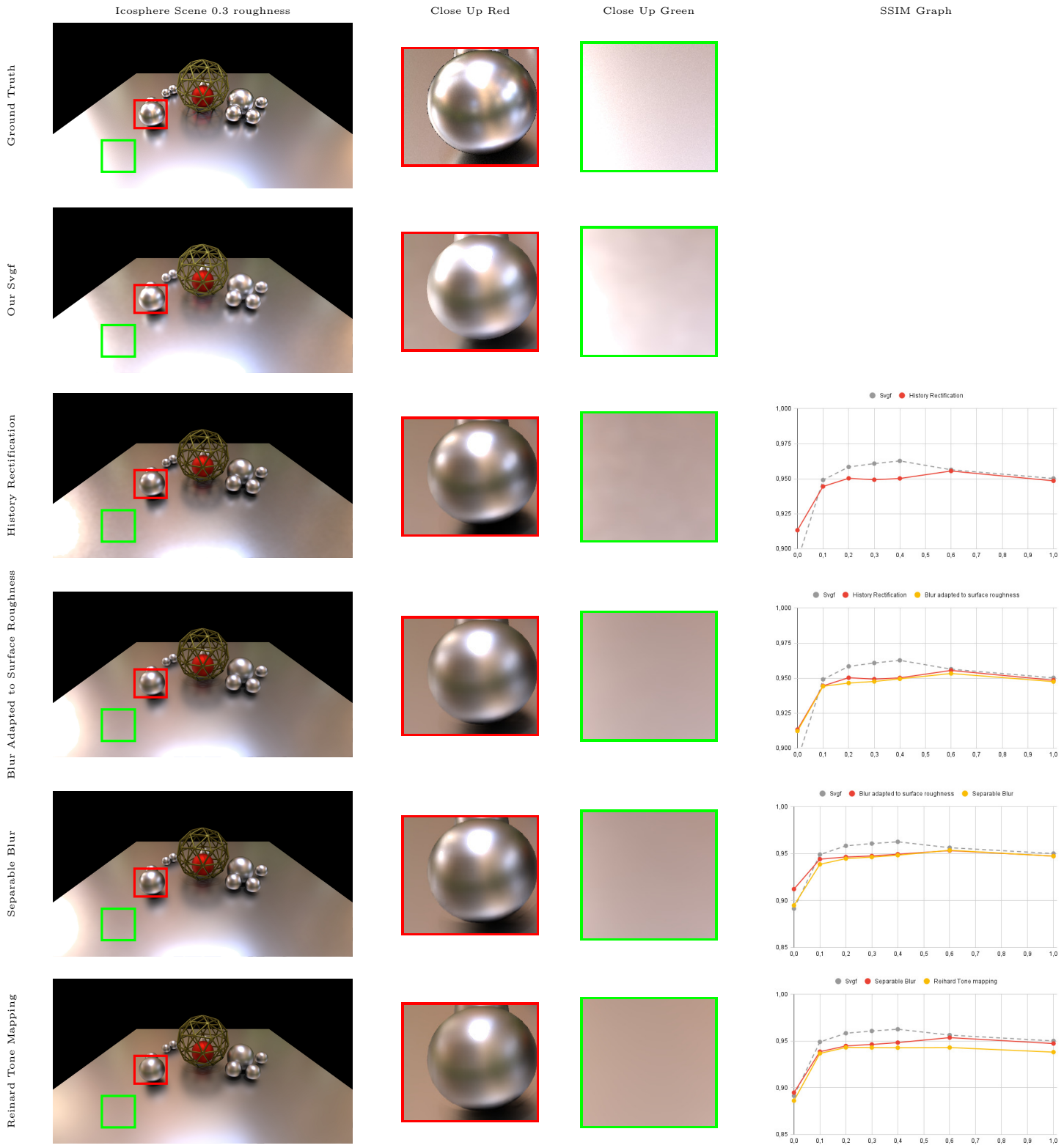


Table 5.8: Static camera, Icosphere quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.

Moving Camera

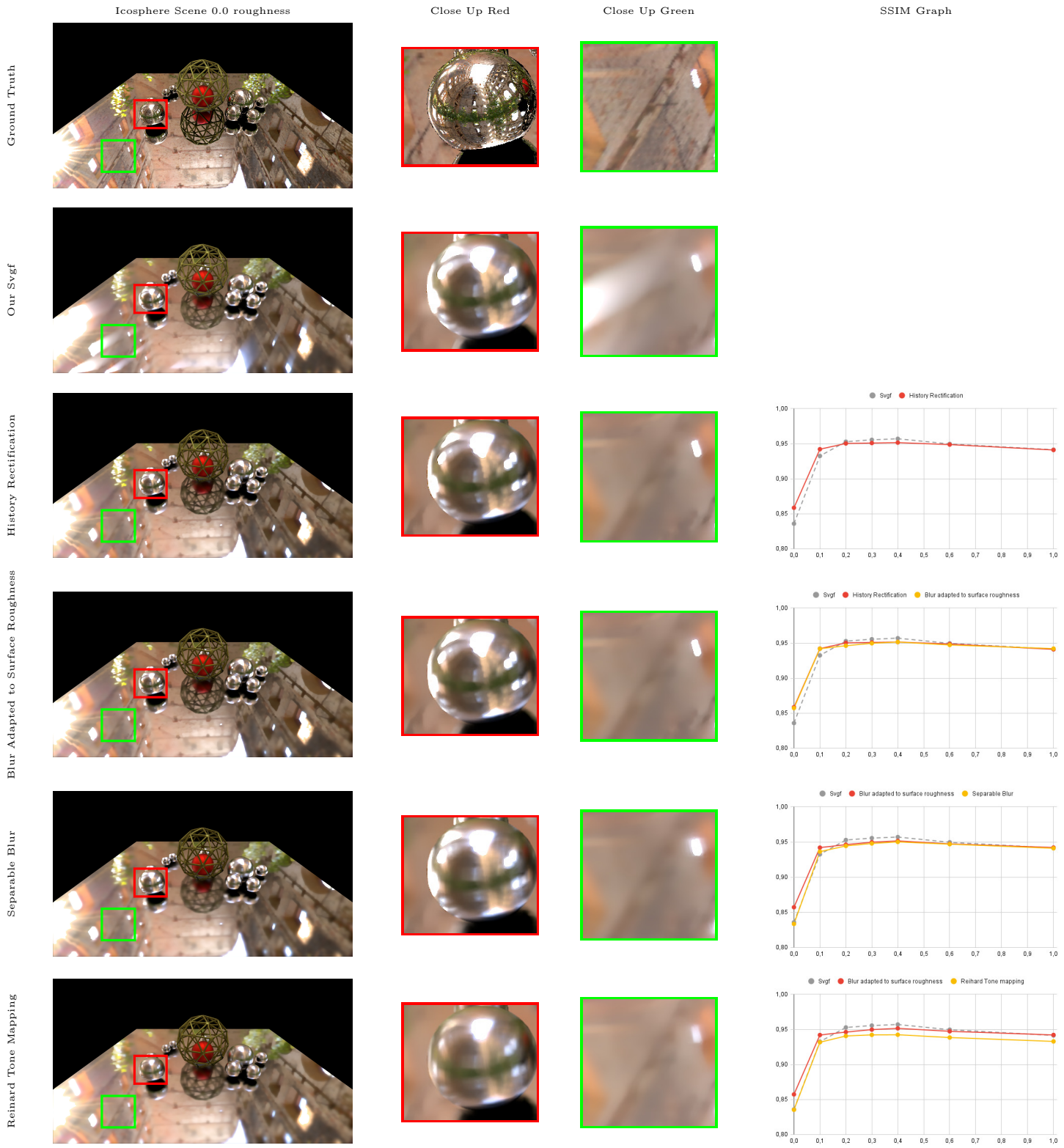


Table 5.9: Moving camera, icosphere quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.)

Moving Objects and Light

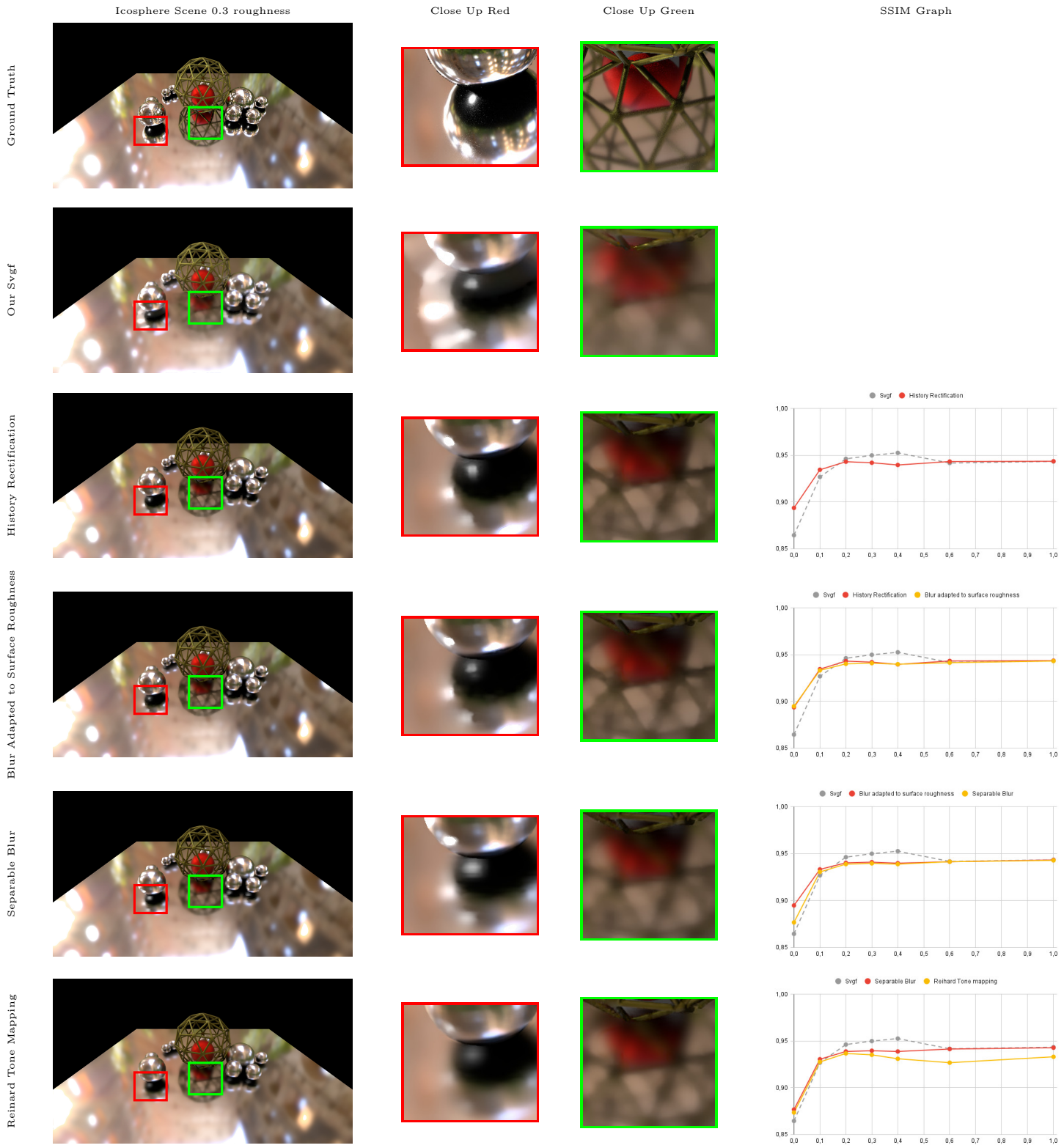


Table 5.10: Light and object movement, icosphere quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the surface roughness.

5.2.1 Cubes Distance and Icosphere analyses

History Rectification:

- **Static:** The static scenes present an increase in quality at the 0.0 surface roughness values, a decrease in the mid tier surface roughness values ([0.2, 0.3, 0.4]) and 0.1 roughness values and, finally, a similar quality in the high tier surface roughnesses ([0.6, 1.0]). These results are to be expected given the removal of the low probability paths caused by history rectification. Temporally unstable noise becomes more apparent in the image (See cubes distance scene, static camera, close up green).
- **Moving Camera:** The results follow a similar pattern to the static tests, with an increase in quality in the low surface roughness values ([0.0, 0.1]), a decrease in the mid tier surface roughness values ([0.2, 0.3, 0.4]) and, finally, a similar result for the high tier surface roughnesses ([0.6, 1.0]). These similar (follow the same pattern) but better results can be explained by the improvements towards ghosting (Compare the close up table entries to the previous table entry), which was not present in the static tests.
- **Moving Objects and Light:** Both scene results show the same tendencies to the previous camera states, with the exception that the Cubes Distance scene results have a large divergence compared with Svgf on the higher tier surface roughnesses. Looking at both scenes, a lot of improvements can be seen towards ghosting artifacts.

Blur Adapted To Surface Roughness: As can be seen, the quality results from the introduction of this technique remain very similar to the previous entry, however, a reduction in the amount of temporally unstable noise can be noted (notice how in the cubes distance scene, static camera, close up green, the results appear more homogenous). Banding artifacts start to become visible (cubes distance scene, static camera, close up red).

Separable Blur: Similarly to the previous entry, the quality results stay largely unchanged with only the low surface roughnesses being affected due to over blurring (see the increase in blur in the cubes distance scene, moving camera). Banding artifacts become more visible and can be easily noticeable (cubes distance scene, static camera, close up red).

Reinhard Tone Mapping: The introduction of this technique causes a slight divergence from the ground truth results, however, visually, the results stay largely the same.

Static Camera

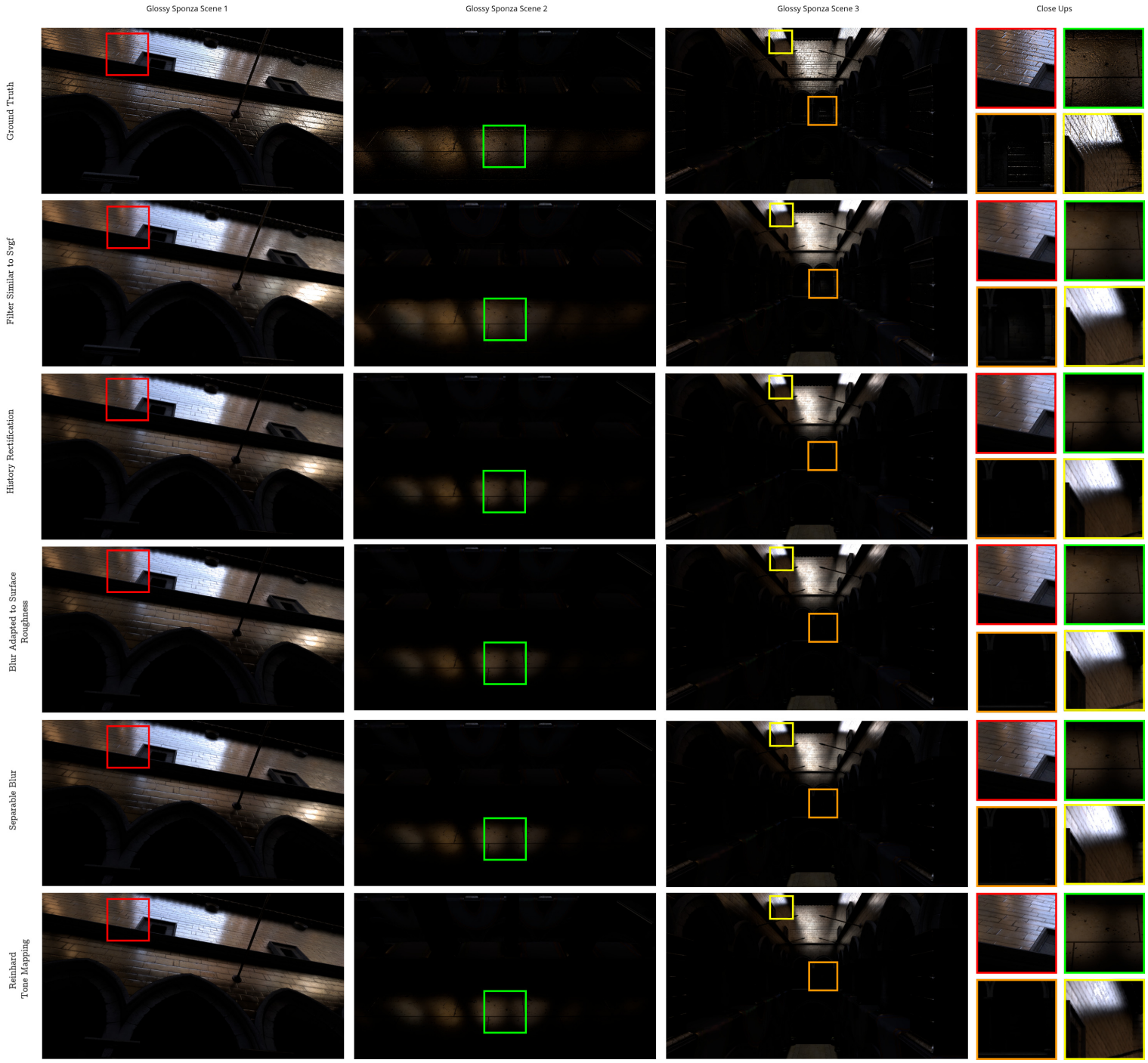


Figure 5.6: Static camera, sponza test images

Moving Camera

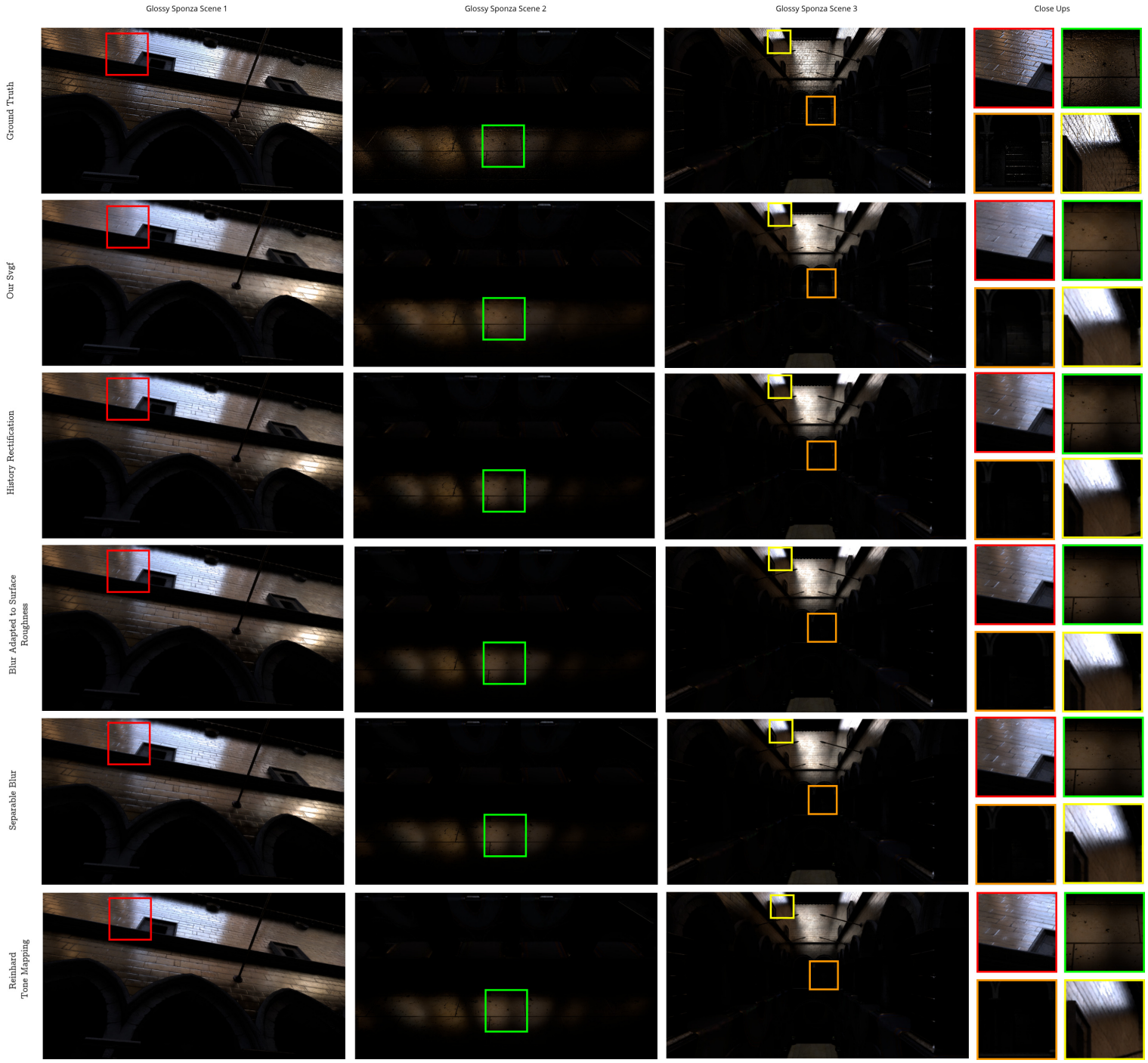


Figure 5.7: Moving camera, sponza test images

Scenes	Glossy Sponza Scene 1	Glossy Sponza Scene 2	Glossy Sponza Scene 3
Svgf	0.7914	0.8932	0.8989
History Rectification	0.7841	0.8296	0.8566
Blur Adapted To Surface Roughness	0.7816	0.8302	0.8533
Separable Blur	0.7822	0.8313	0.8525
Reinhard Tone Mapping	0.7853	0.8368	0.8561

Table 5.11: Static camera, sponza quality evaluation (SSIM)

Scenes	Glossy Sponza Scene 1	Glossy Sponza Scene 2	Glossy Sponza Scene 3
Svgf	0.7696	0.8789	0.8832
History Rectification	0.7771	0.8583	0.8584
Blur Adapted To Surface Roughness	0.7746	0.8583	0.8525
Separable Blur	0.7709	0.8565	0.851
Reinhard Tone Mapping	0.7755	0.8622	0.8566

Table 5.12: Moving camera, sponza quality evaluation (SSIM)

5.2.2 Sponza Analysis

History Rectification:

- **Static:** Results show a deterioration in the quality metrics when this technique is introduced. By direct observation, differences can also be perceived with regions of the scene becoming darker (See Glossy Sponza Scene 2).
- **Moving Camera:** Similarly to the previous test case, the results show an overall deterioration in the metrics with only the Glossy Sponza Scene 1 showing a positive improvement. However, by direct observation, a great reduction in ghosting can be noticed (See Moving Camera, Glossy Sponza Scene 1).

Blur Adapted To Surface Roughness and Separable Blur Similar to previous tests, metrics remain largely unchanged with improvements regarding the temporally unstable noise.

Reinhard Tone Mapping: Surprisingly, the introduction of this technique improved the quality metrics on every scene. This is probably caused by a potential brightening effect, counteracting the impacts of History Rectification.

Static Camera

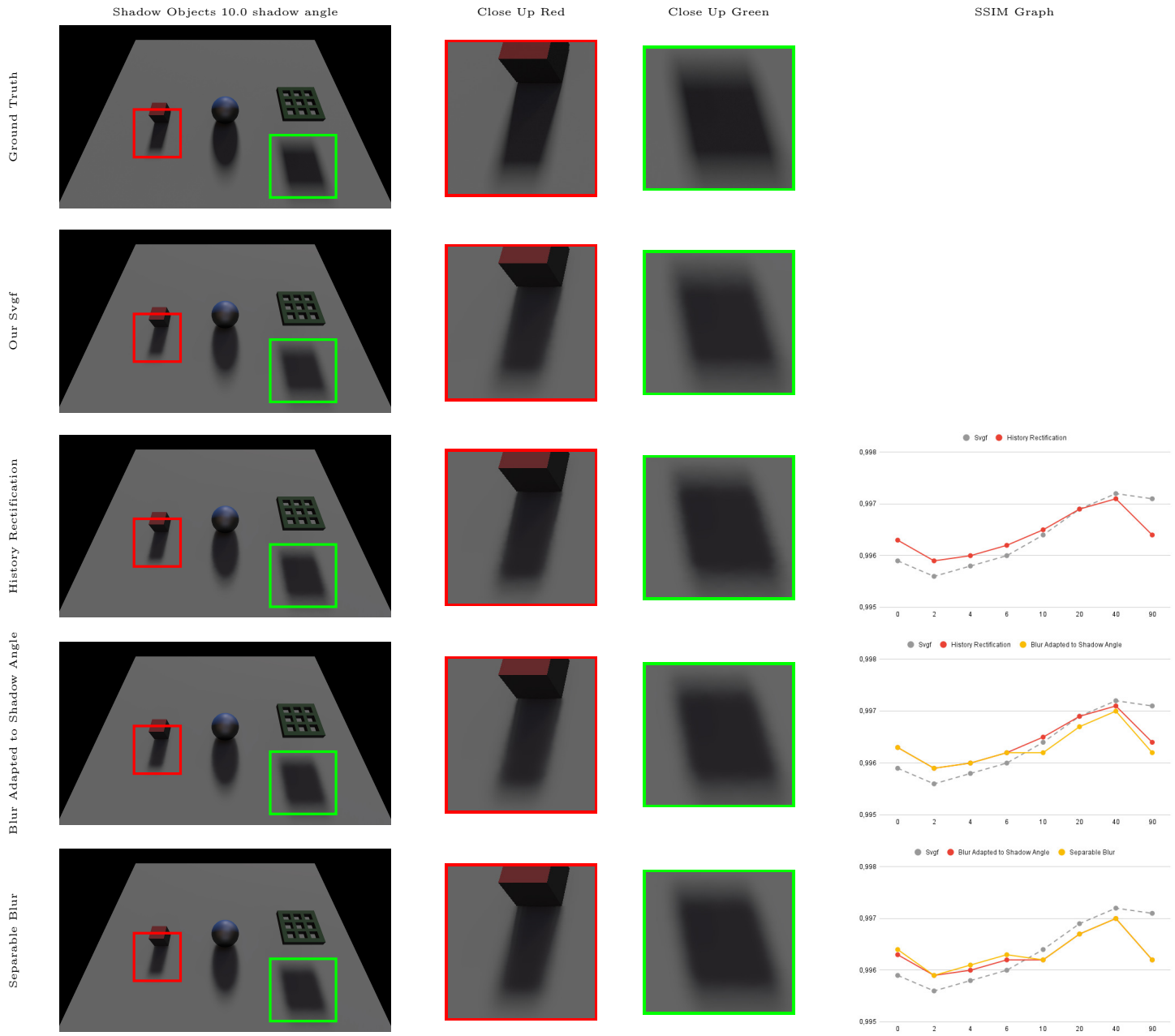


Table 5.13: Static camera, shadow objects quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

Light and Object Movement

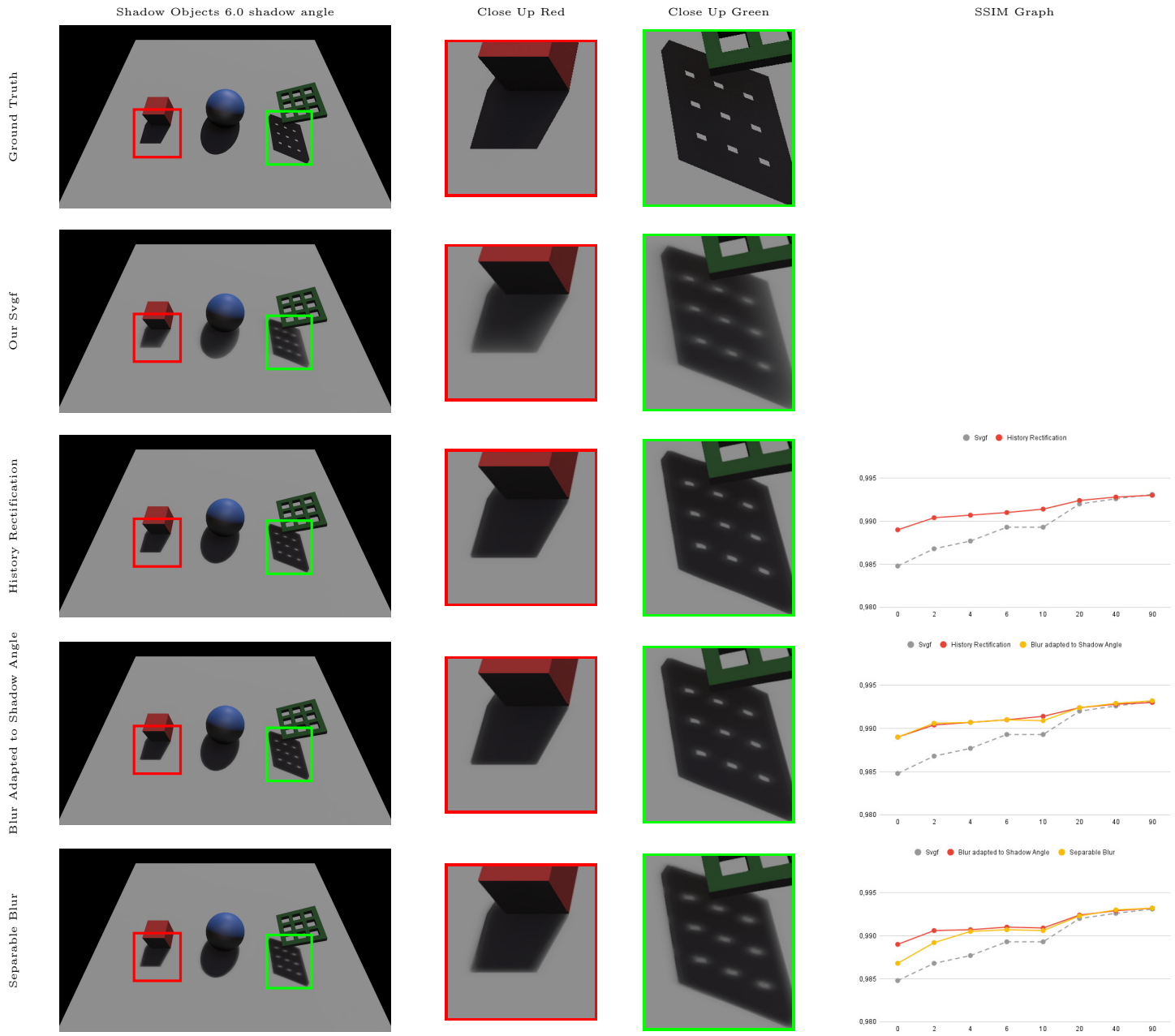


Table 5.14: Light and object movement, shadow objects quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

Static Camera

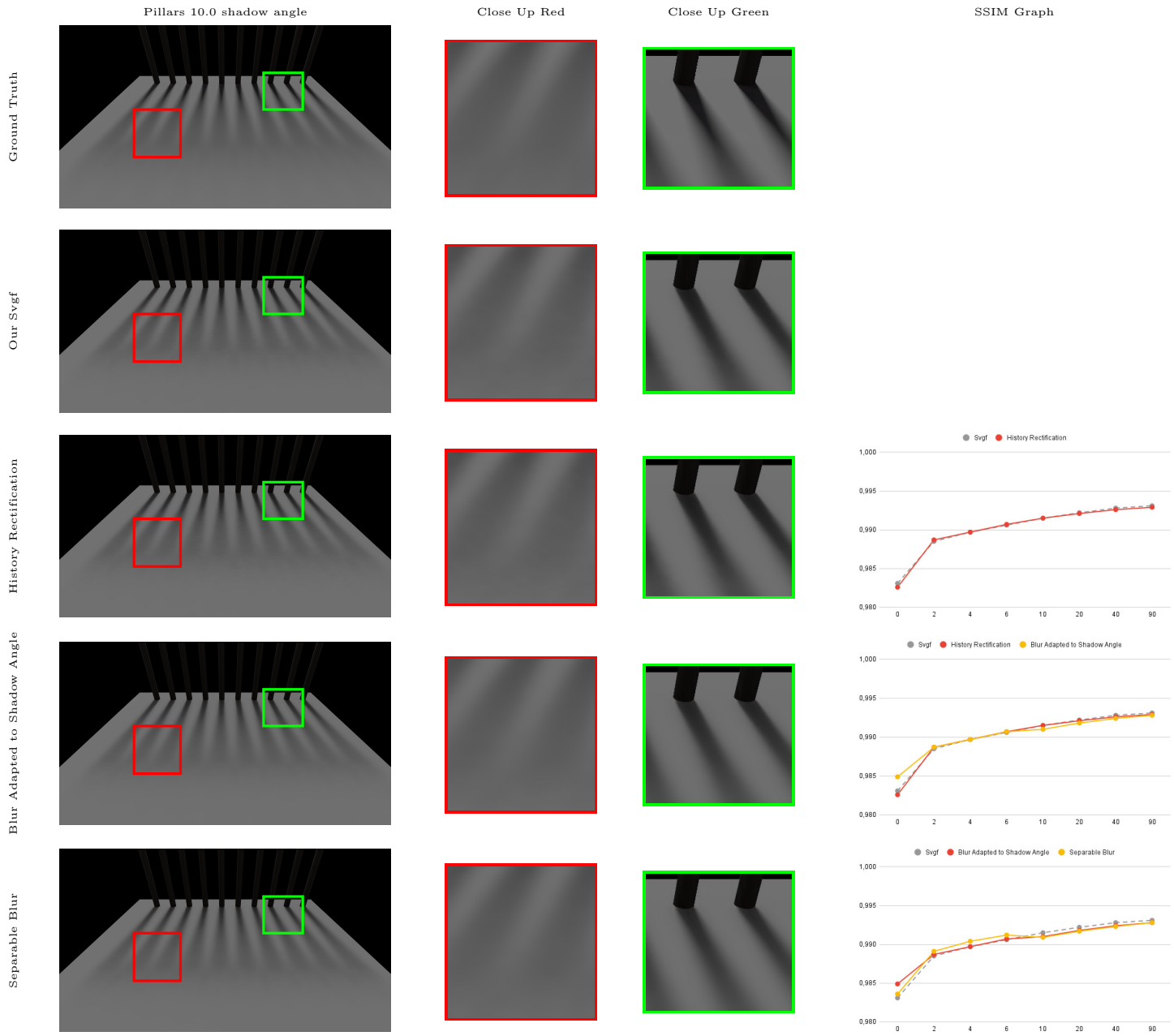


Table 5.15: Static camera, pillars quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

Moving Objects and Light

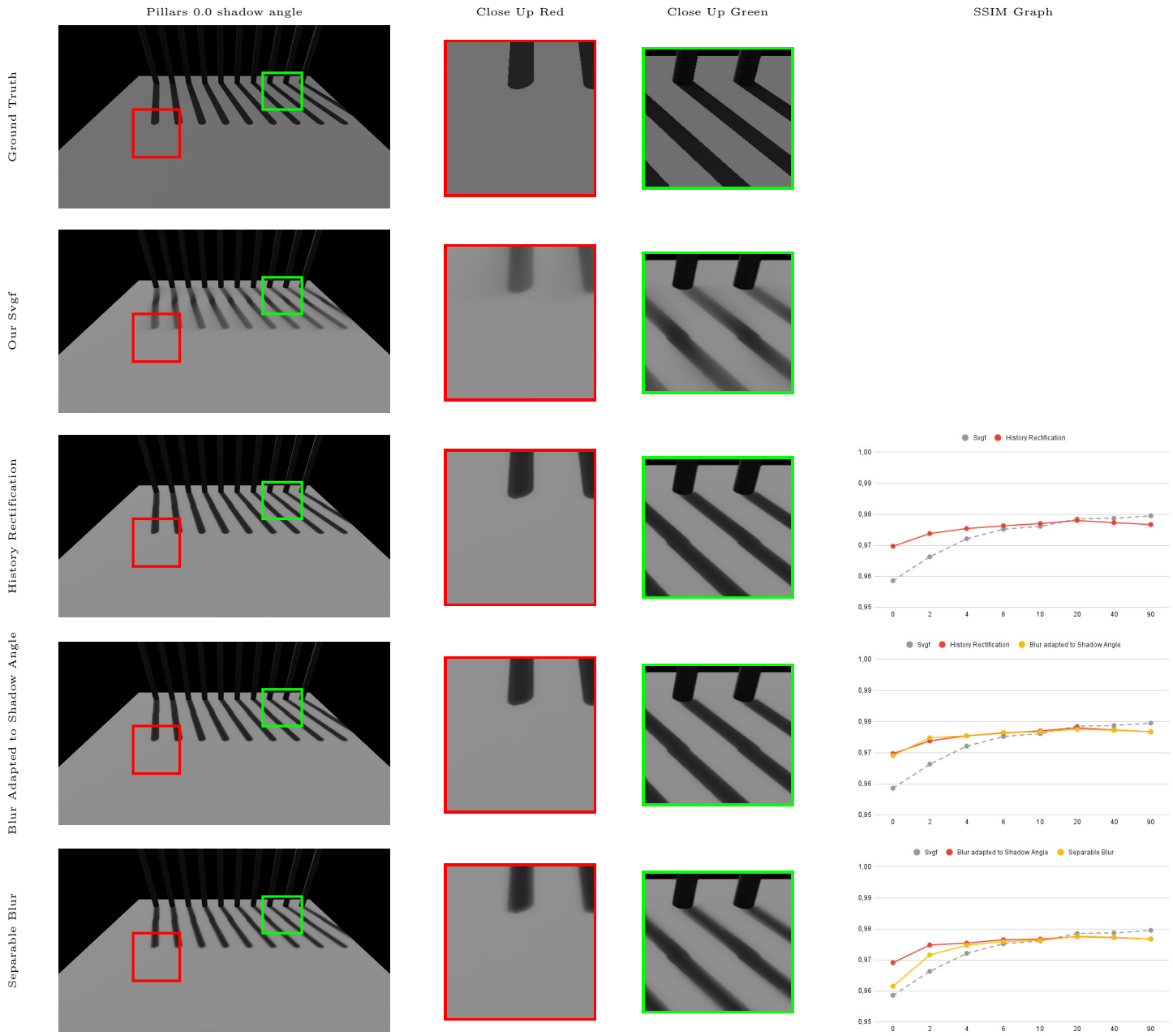


Table 5.16: Light and object movement, pillars quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

Static Camera

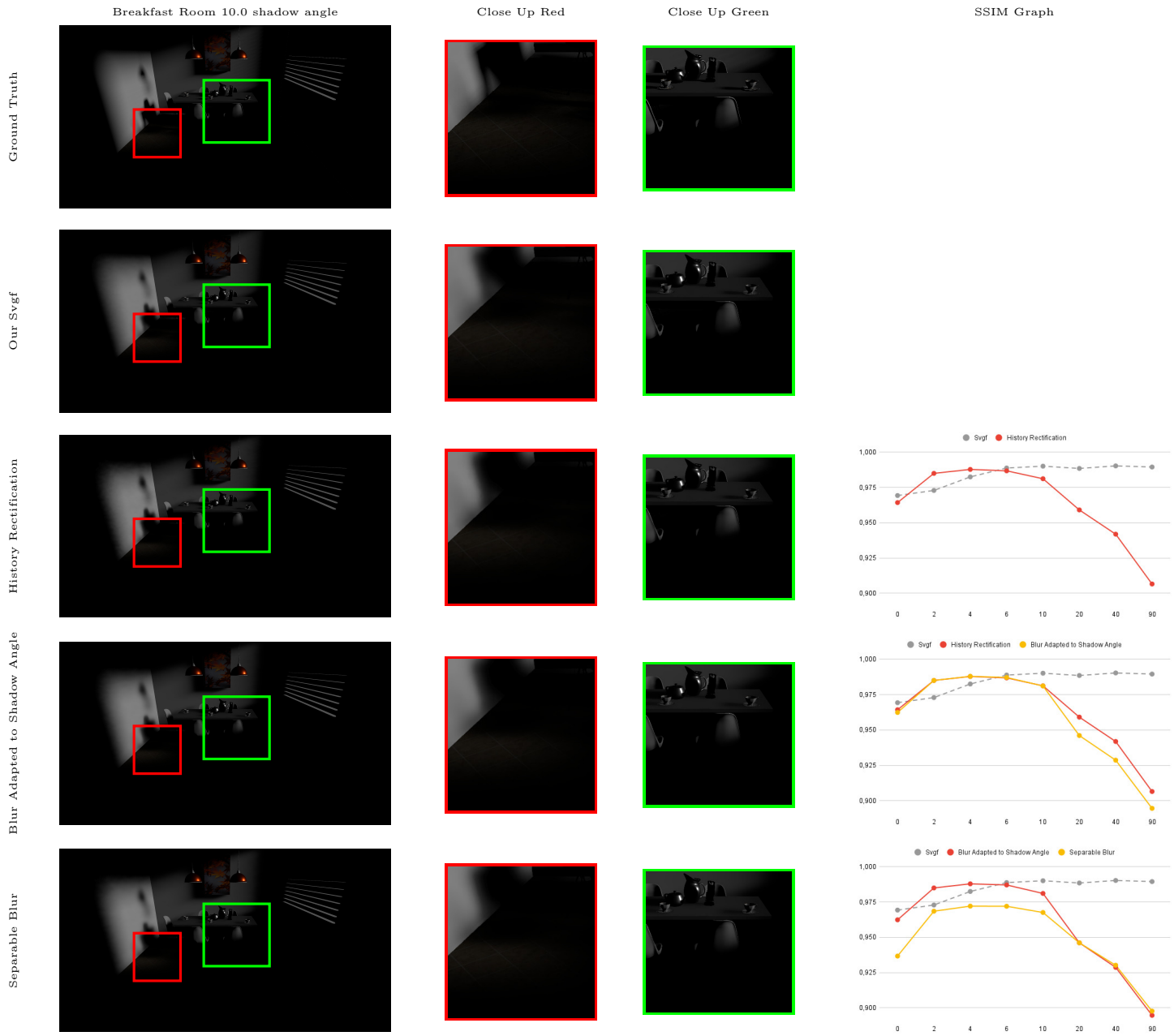


Table 5.17: Static camera, breakfast room quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

Moving Objects and Light

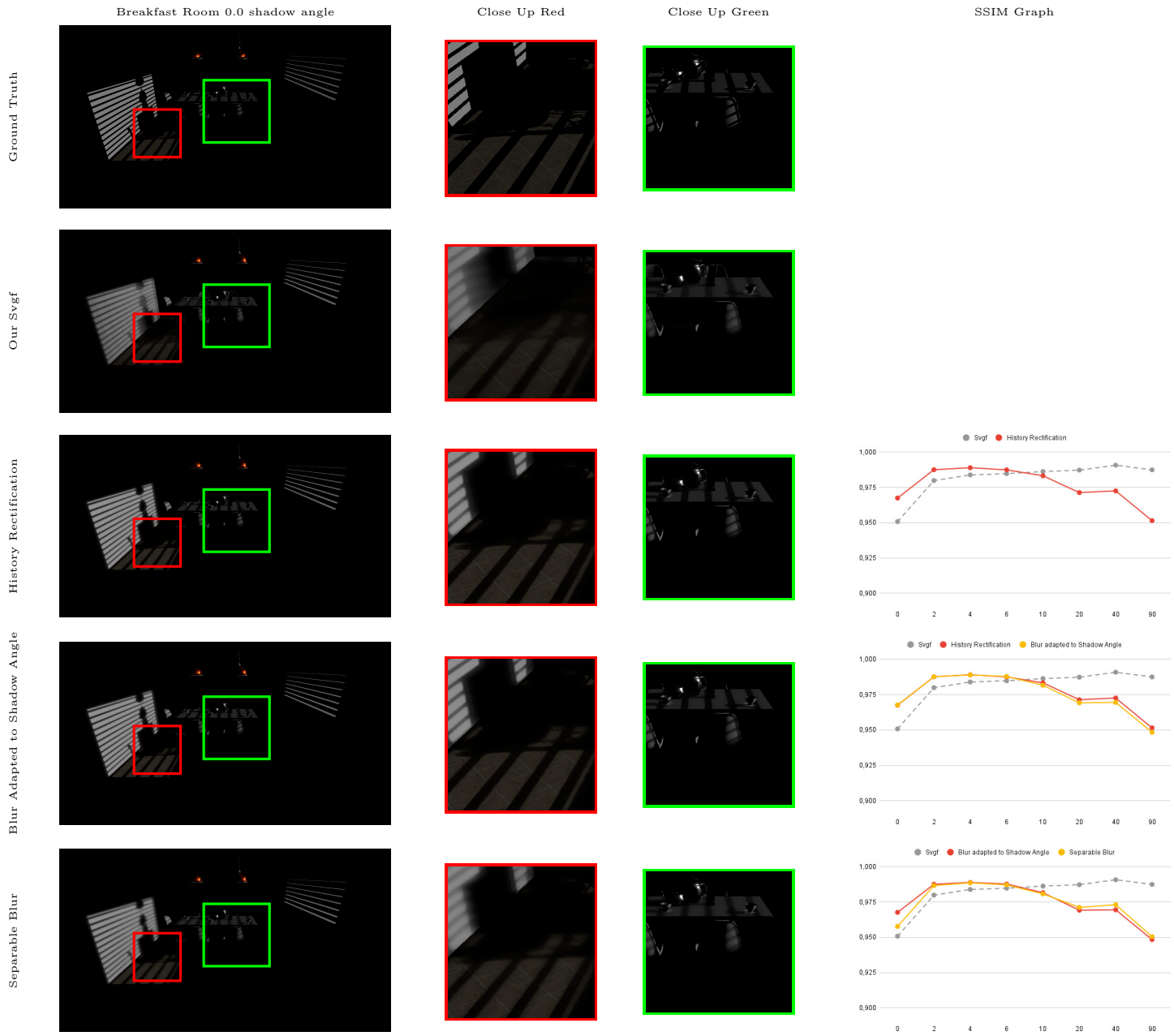


Table 5.18: Moving Objects and Light, breakfast room quality evaluation. The graphs y axis represents the SSIM metric and the x axis represents the shadow angle.

5.2.3 Shadows Analysis

History Rectification:

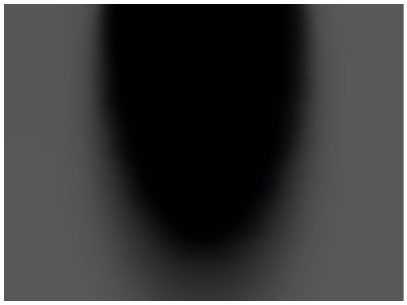
- **Static:** The quality metrics show improvements towards the low shadow angles, and a gradual worsening towards the high shadow angles. This worsening can be justified by the same reasoning as the reflections, the removal of low probability paths. Jagged lines and noise become more noticeable (see Figure 5.8).
- **Light and Object Movement:** The quality metrics show a similar trend to the results provided by the static camera, but more positive, as artifacts related to ghosting are removed (See shadow objects scene, close up green).

Blur Adapted To Shadow Angle: As can be seen, the quality results from this technique remain very similar to the previous entry, however, a reduction in the amount of noise and jagged lines can be noted (See Figure 5.8).

Separable Blur: Similar to the previous entry, the quality results stay similar with only the low shadow angles being affected due to over blurring (see shadow objects scene, close up green).

(a) Our Svgf

(b) History rectification



(c) Blur Adapted to Shadow Angle

Figure 5.8: Comparison between Our Svgf, History rectification and Blur adapted to Shadow Angle on Shadow Objects scene with shadow angle with value 10.0. Images have increased contrast in order to better notice the jagged lines

5.3 Performance Metrics

Cubes Distance Timings

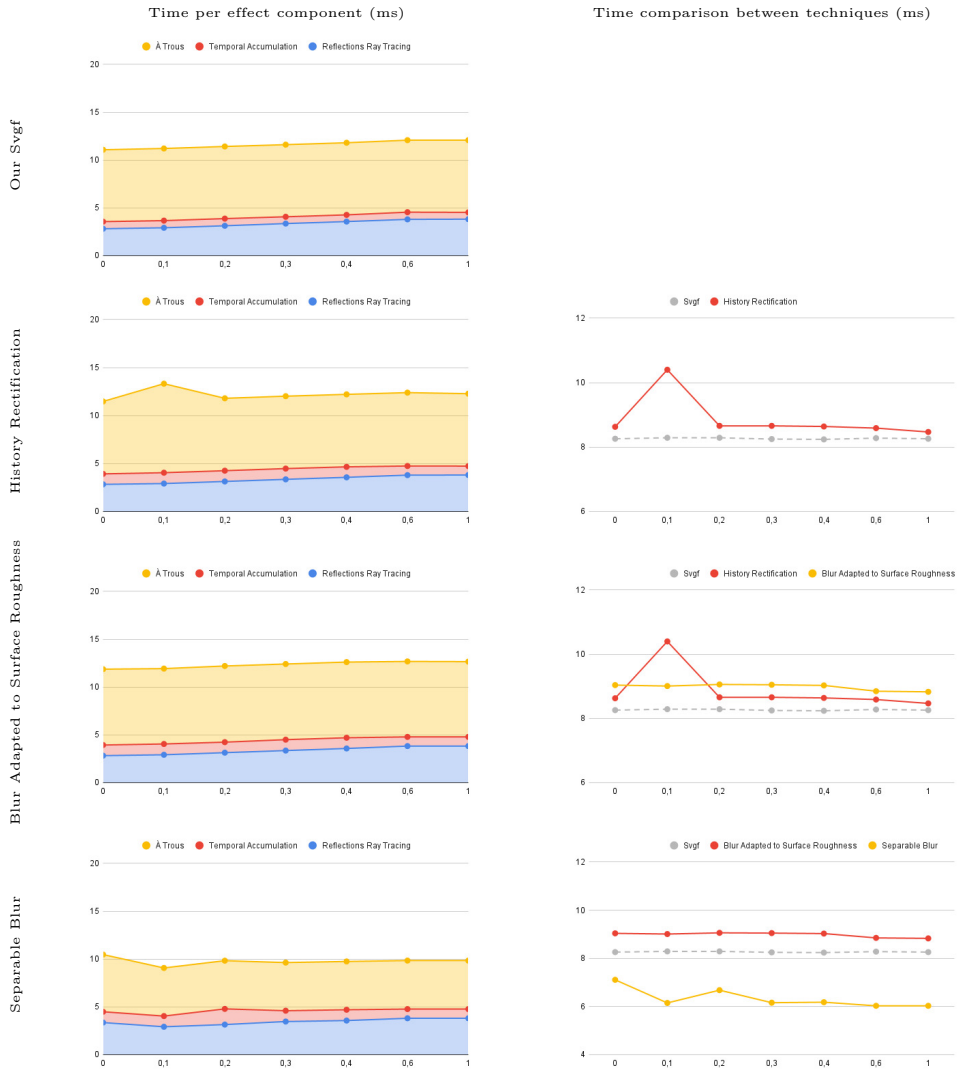


Table 5.19: Cubes distance test performance evaluation

Icosphere Timings

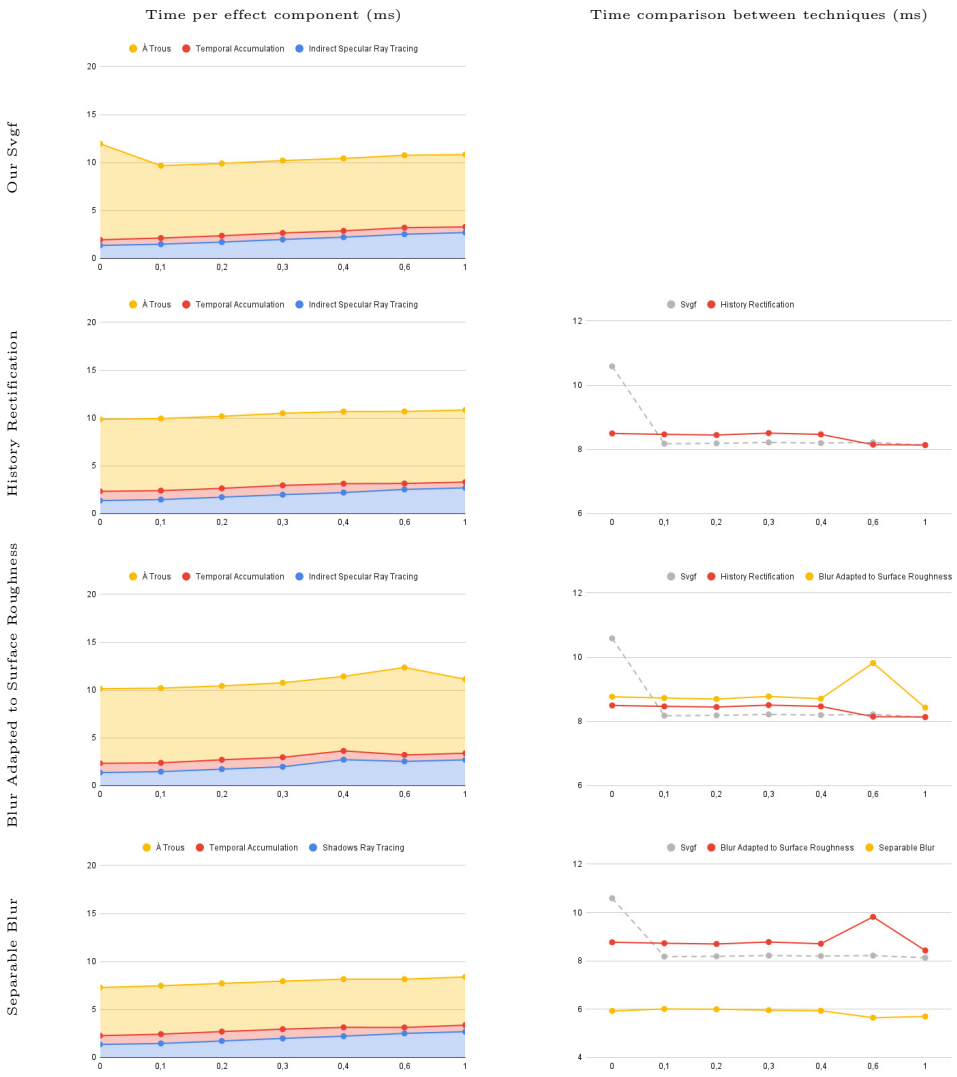


Table 5.20: Icosphere performance evaluation

Sponza Timings

Svgf	Indirect Specular Raytracing (ms)	Temporal Acumulation Reflection (ms)	À Trous (ms)
Glossy Sponza Scene 1	10,62	0,97	10,22
Glossy Sponza Scene 2	10	0,82	7,8
Glossy Sponza Scene 3	12,1	1,03	7,72
History rectification			
Glossy Sponza Scene 1	9,71	1,17	8,84
Glossy Sponza Scene 2	9,98	1,24	7,81
Glossy Sponza Scene 3	10,7	1,32	7,72
Blur Adapted To Surface Roughness			
Glossy Sponza Scene 1	8,53	1,08	9,06
Glossy Sponza Scene 2	11,6	1,26	10,65
Glossy Sponza Scene 3	10,71	1,31	8,57
Separable Blur			
Glossy Sponza Scene 1	9,9	1,18	5,14
Glossy Sponza Scene 2	12,11	1,29	5,35
Glossy Sponza Scene 3	10,7	1,29	5,22

Table 5.21: Sponza performance evaluation

Shadow Objects Timings

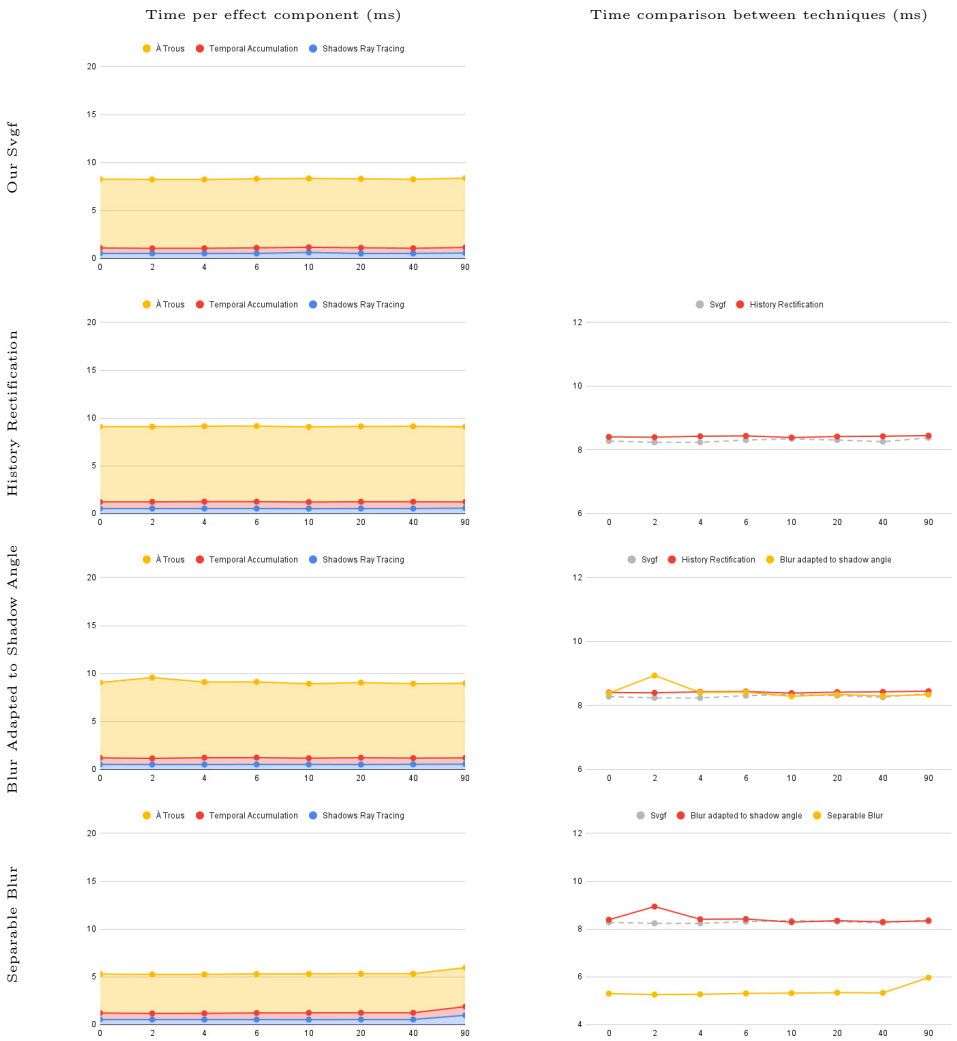


Table 5.22: Shadow objects performance evaluation

Pillars Performance

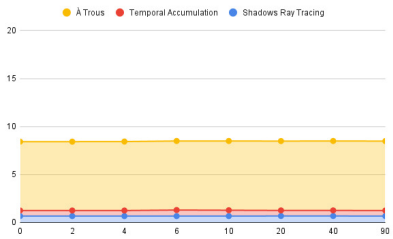
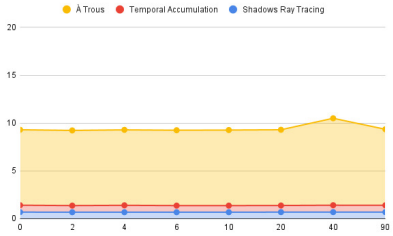
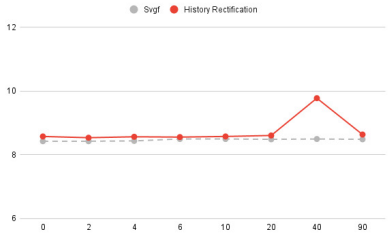
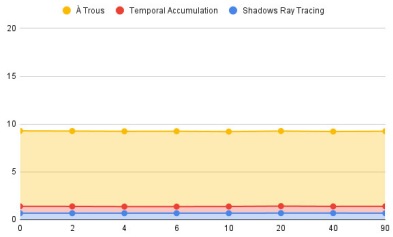
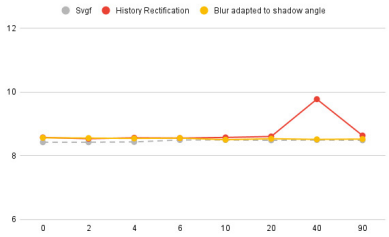
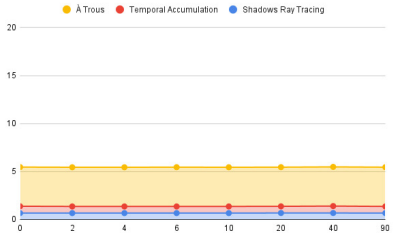
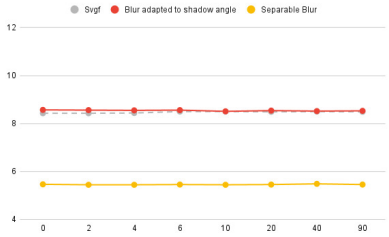
	Time per effect component (ms)	Time comparison between techniques (ms)
Our Svgf		
History Rectification		
Blur Adapted to Surface Roughness		
Separable Blur		

Table 5.23: Pillars performance evaluation

Breakfast Room Timings

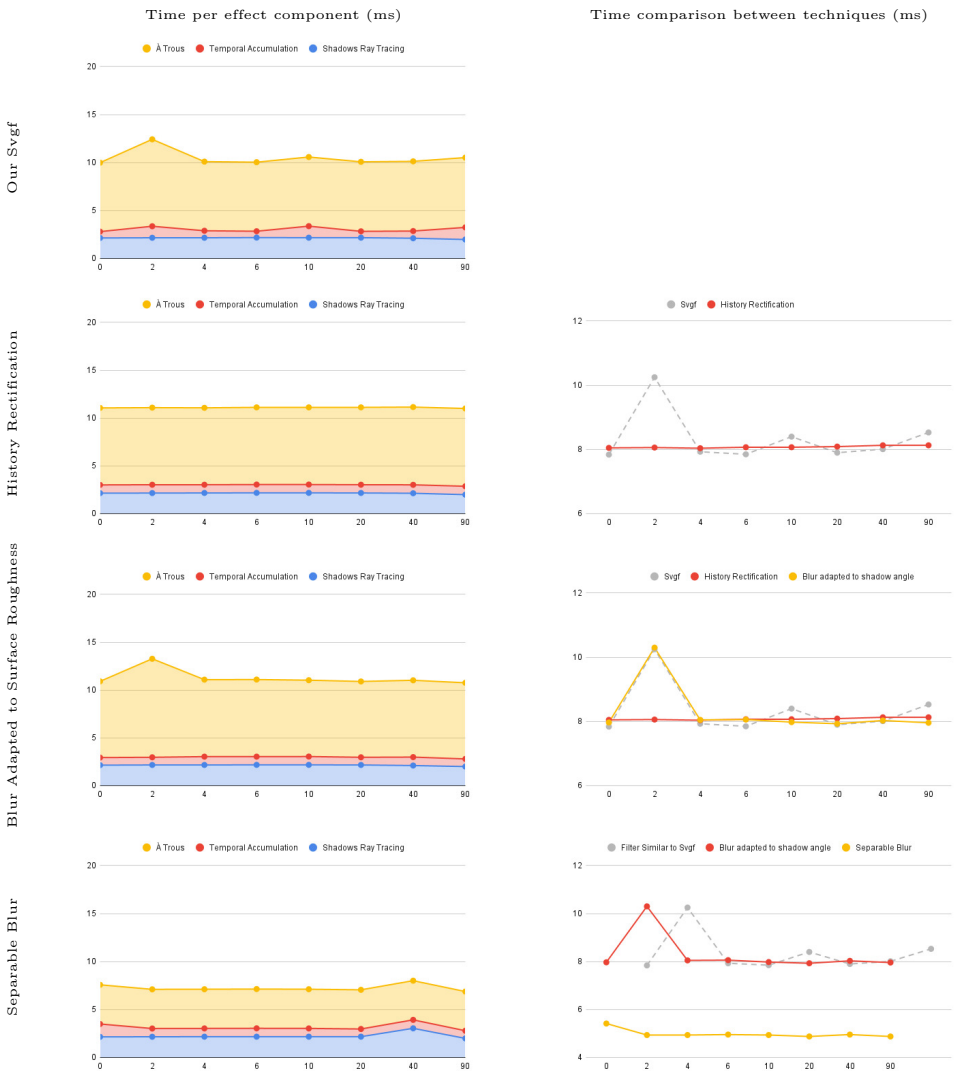


Table 5.24: Breakfast room performance evaluation

5.3.1 Timing Analysis

As can be observed, the results follow the expectations. History Rectification introduces a slight reduction in speed, the blurs adapted to the effect features remain similar and the separable blur greatly improves the timings. A few interesting observations can also be made. Firstly, all the blur iterations of the À Trous filter take a similar time (For Svgf on the Icosphere scene with 0.3 surface roughness, [1.93ms, 1.88ms, 1.88ms, 1.86ms]). This is surprising due to À Trous visiting progressively further away pixels, and as such, due to data locality, it was expected to increase the timings. Another important observation, is the impact of the surface roughness on the time it takes the ray tracing of the indirect specular channel. As can be observed, there is an increase in time proportional to the surface roughness. This is, most likely, a consequence of data locality. As the surface roughness increases, the rays have wider differences in directions and hit points, being more divergent between them. A way to reduce the impact of this is to use a ray binning strategy.

Timings	Cubes Distance	Icosphere	Glossy Sponza 1	Glossy Sponza 2	Glossy Sponza 3	Shadow Objects	Pillars	Breakfast Room
max	80,1	79,8	51,9	45,6	44,5	86,8	83,7	45,5
avg	79,2	77,2	51,5	45,3	43,4	84,4	82,6	45,1
min	76,5	76,1	51,2	45,2	43,1	83,8	82,1	44,9

Using all the techniques previously mentions, our goals of 30fps where accomplished, although the introduction of the indirect diffuse component would probably be too costly for achieving our goals.

Chapter 6

Conclusion

6.1 Conclusion

This project ended up being a very challenging yet rewarding experience, requiring a lot of different areas and involving a multitude of disciplines, rendering techniques, shading and image processing. Given our initial proposal of "creating a hybrid implementation with photo realistic quality and real time performance (>30 fps)" we have achieved most of our goals with exception to the indirect diffuse component of lighting due to time constraints.

We also introduce a set of techniques over the original Svgf, allowing the algorithm to perform better in motion and have an increase in quality (SSIM) in some cases (low surface roughness and low shadow angle). The improvements in motion are due to History Rectification (Variance Color Clamping), resulting in reduced ghosting artifacts and a better response to movement and changes in the scene. This technique, however, reduces the efficiency of temporal accumulation, and as such introduces more noise and flickering into the scene.

To counteract this problem, the initial blur size is adapted to the features of each lighting component (surface roughness and shadow angle), increasing when a large amount of noise is expected (high surface roughness and high shadow angle). This also further shows the importance of separating the denoising procedure into multiple channels per lighting component.

Reinhard Tone Mapping is also shown to be a valuable way of improving denoising on the indirect specular component of lighting, trading ground truth correctness for less perceived flickering.

Blurring operations over multiple iterations are a very expensive operation and as such we introduced a separable blur, where the normal 5x5 blur kernel is separated into a vertical and horizontal pass, greatly increasing the performance of the algorithm. However, artifacts like banding become more prevalent and the results become sightlier blurred (reducing some artifacts related to insufficient blurring but also over blurring some regions).

6.2 Future Work

Looking at the results of this project, it is possible to see that improvements can still be made.

History Rectification impacts decrease towards high shadow angles and high surface roughnesses, and, as such, adapting this technique along this properties should result in a reduction in the negative impacts it causes.

Separable blur is also another technique that can be further explored, as the performance gains are very significative, but the extra blurring it causes may not be desirable. The aggravation of banding artifacts are also another problem introduced by this, but this should be resolved by adding random offsets to the À-Trous Wavelet Transform procedure.

Furthermore, it is also possible to observe that the amount noise of both shadows and reflections depends on the average distance the rays have to travel. As such, this property can probably be exploited in order to guide the blurring procedure, reducing over blurring artifacts.

The indirect diffuse component was also not implemented into the system and as such, is also passed has future work.

Bibliography

[1] E. Cerica, https://www.artstation.com/enricocerica.

[2] Kri, https://commons.wikimedia.org/wiki/File:Rendering_eq.png.

[3] Jurohi, https://en.wikipedia.org/wiki/File:BSDF05_800.png.

[4] M. A. Romain Guy, "Physically based rendering in filament," https://google.github.io/filament/Filament.md.html#about/authors.

[5] J. de Vries, https://learnopengl.com/PBR/Theory.

[6] K. lu and C.-F. Chang, "Acceleration of photon mapping : Adaptive sampling of irradiance cache by components," *APSIPA ASC 2009 - Asia-Pacific Signal and Information Processing Association 2009 Annual Summit and Conference*, 10 2009, https://eprints.lib.hokudai.ac.jp/dspace/bitstream/2115/39637/1/MA-L1-3.pdf.

[7] G. H. Matt Pharr, Wenzel Jakob, *Physically Based Rendering: From Theory to Implementation.* Morgan Kaufmann, Morgan Kaufmann, http://www.pbr-book.org/3ed-2018/Light_Transport_I_Surface_Reflection/Path_Tracing.html.

[8] J. "llbit" Öqvist, https://chunky.llbit.se/path_tracing.html.

[9] A. Mihut, "Exploring ray tracing techniques in wolfenstein: Youngblood," Nvidia, https://www.khronos.org/blog/vulkan-ray-tracing-best-practices-for-hybrid-rendering.

[10] D. Koch, "Vulkan ray tracing final specification release," Nvidia, https://www.khronos.org/blog/vulkan-ray-tracing-final-specification-release.

[11] A. Galvan, https://alain.xyz/blog/raytracing-denoising.

[12] D. Lin, https://dqlin.xyz/tech/2016/12/03/03_render/.

[13] K. lu and C.-F. Chang, "A survey of temporal antialiasing techniques," *Eurographics*, 2020. [Online]. Available: http://behindthepixels.io/assets/files/TemporalAA.pdf

[14] "A gentle introduction to bilateral filtering and its applications," https://people.csail.mit.edu/sparis/bf_course/course_notes.pdf.

[15] W. C. Zheyuan Xie, Yan Dong, "Cuda svgf," https://github.com/ZheyuanXie/CUDA-Path-Tracer-Denoising.

[16] C. Schied, A. Kaplanyan, C. Wyman, A. Patney, C. R. A. Chaitanya, J. Burgess, S. Liu, C. Dachsbacher, A. Lefohn, and M. Salvi, "Spatiotemporal variance-guided filtering," 2017, https://cg.ivd.kit.edu/publications/2017/svgf/svgf_preprint.pdf.

[17] C. Schied and A. Panteleev, "Real-Time Path Tracing and Denoising in Quake II," 2019, https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s91046-real-time-path-tracing-and-denoising-in-quake-2.pdf.

[18] Y. U. Tomasz Stachowiak, "Stochastic screen-space reflections," Electronic Arts, https://www.ea.com/frostbite/news/stochastic-screen-space-reflections.

[19] D. Immel, M. Cohen, and D. Greenberg, "A radiosity method for non-diffuse environments," *ACM SIGGRAPH Computer Graphics*, vol. 20, pp. 133–142, 08 1986, https://www.researchgate.net/publication/220720201_A_Radiosity_Method_for_Non-Diffuse_Environments.

[20] J. T. Kajiya, "The rendering equation," *SIGGRAPH Comput. Graph.*, vol. 20, no. 4, p. 143–150, Aug. 1986. [Online]. Available: https://doi.org/10.1145/15886.15902

[21] F. O. Bartell, E. L. Dereniak, and W. L. Wolfe, "The Theory And Measurement Of Bidirectional Reflectance Distribution Function (Brdf) And Bidirectional Transmittance Distribution Function (BTDF)," in , ser. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, G. H. Hunt, Ed., vol. 257, Mar. 1981, pp. 154–160, https://ui.adsabs.harvard.edu/abs/1981SPIE..257..154B/abstract.

[22] B. Burley, "Physically-based shading at disney," https://media.disneyanimation.com/uploads/production/publication_asset/48/asset/s2012_pbs_disney_brdf_notes_v3.pdf.

[23] B. Karis, "Real shading in unreal engine 4," *Acm Siggraph Computer Graphics*, 2013, https://cdn2.unrealengine.com/Resources/files/2013SiggraphPresentationsNotes-26915738.pdf.

[24] C. Schlick, "An inexpensive brdf model for physically-based rendering," *Comput. Graph. Forum*, vol. 13, pp. 233–246, 1994, https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.2297&rep=rep1&type=pdf.

[25] E. Heitz, "Understanding the masking-shadowing function in microfacet-based brdfs," *Journal of Computer Graphics Techniques (JCGT)*, vol. 3, 06 2014, http://jcgt.org/published/0003/02/03/paper.pdf.

[26] N. Benty, K.-H. Yao, P. Clarberg, L. Chen, S. Kallweit, T. Foley, M. Oakes, C. Lavelle, and C. Wyman, "The Falcor rendering framework," 03 2020, https://github.com/NVIDIAGameWorks/Falcor.

[27] M. Deering, S. Winner, B. Schediwy, C. Duffy, and N. Hunt, "The triangle processor and normal vector shader: A vlsi system for high performance graphics," in *Proceedings*

of the 15th Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH '88. New York, NY, USA: Association for Computing Machinery, 1988, p. 21–30. [Online]. Available: https://doi.org/10.1145/54852.378468

[28] E. Clua, A. Montenegro, P. Pagliosa, T. Sabino, P. Andrade, and L. Lattari, "Efficient use of in-game ray-tracing techniques," 11 2011, https://www.researchgate.net/publication/281762363_Efficient_Use_of_In-Game_Ray-Tracing_Techniques.

[29] M. Pharr, *On the Importance of Sampling*. Apress, 2019, ch. 15, http://raytracinggems.com.

[30] A. Wolfe, https://blog.demofox.org/2017/05/29/when-random-numbers-are-too-random-low-discrepancy-sequences/.

[31] A. Benyoub, Unity, https://auzaiffe.files.wordpress.com/2019/05/digital-dragons-leveraging-ray-tracing-hardware-acceleration-in-unity.pdf.

[32] M. F. Iliyan Georgiev, "Blue-noise dithered sampling," *Acm Siggraph Computer Graphics*, p. 35, 2016, https://www.arnoldrenderer.com/research/dither_abstract.pdf.

[33] L. J. F. Pedersen, "Temporal reprojection anti-aliasing in inside," *Game Developers Conference*, 2016, https://www.gdcvault.com/play/1022970/Temporal-Reprojection-Anti-Aliasing-in.

[34] B. Karis, "High quality temporal supersampling," *Acm Siggraph Computer Graphics*, 2014, http://advances.realtimerendering.com/s2014/.

[35] T. Sousa, "Graphics gems from cryengine 3," *Acm Siggraph Computer Graphics*, 2014, http://advances.realtimerendering.com/s2013/.

[36] M. Salvi, "An excursion in temporal supersampling," *Game Developers Conference*, 2016, https://developer.download.nvidia.com/gameworks/events/GDC2016/msalvi_temporal_supersampling.pdf.

[37] C. P. Christoph Schied and C. Dachsbacher, "Gradient estimation for real-time adaptive temporal filtering," 2018, https://cg.ivd.kit.edu/publications/2018/adaptive_temporal_filtering/adaptive_temporal_filtering.pdf.

[38] H. Dammertz, D. Sewtz, J. Hanika, and H. P. A. Lensch, "Edge-avoiding À-trous wavelet transform for fast global illumination filtering," in *Proceedings of the Conference on High Performance Graphics*, ser. HPG '10. Goslar, DEU: Eurographics Association, 2010, p. 67–75, https://jo.dreggn.org/home/2010_atrous.pdf.

[39] A. Kaplanyan and C. Dachsbacher, "Path space regularization for holistic and robust light transport," *Computer Graphics Forum*, vol. 32, 05 2013, https://cg.ivd.kit.edu/publications/p2013/PSR_Kaplanyan_2013/PSR_Kaplanyan_2013.pdf.

[40] E. Kerzner and M. Salvi, "Streaming g-buffer compression for multi-sample anti-aliasing," in *High Performance Graphics*, 2014, https://software.intel.com/content/www/us/en/develop/articles/streaming-g-buffer-compression-for-multi-sample-anti-aliasing.html.

[41] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder," vol. 36, pp. 1–12, 2017, https://dl.acm.org/doi/10.1145/3072959.3073601.

[42] G. W. A. L. J. B. T. S. Colin Barré-Brisebois, Henrik Halén and J. Andersson, *Hybrid Rendering for Real-Time Ray Tracing*. Apress, 2019, ch. 25, http://raytracinggems.com.

[43] J. S. Johannes Deligiannis, ""it just works": Ray-traced reflections in 'battlefield v'," EA DICE, 2019, https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s91023-it-just-works-ray-traced-reflections-in-battlefield-v.pdf.

[44] J. C. Edward Liu, Ignacio Llamas and P. Kelly, *Cinematic Rendering in UE4 with Real-Time Ray Tracing and Denoising*. Apress, 2019, ch. 19, http://raytracinggems.com.

[45] E. Heitz, S. Hill, and M. McGuire, "Combining analytic direct illumination and stochastic shadows," in *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3190834.3190852

[46] M. W. Jakub Boksansky and J. Bittner, *Ray Traced Shadows: Maintaining Real-Time Frame Rates*. Apress, 2019, ch. 13, http://raytracinggems.com.

[47] M. McGuire, M. Mara, D. Nowrouzezahrai, and D. Luebke, "Real-time global illumination using precomputed light field probes," in *Proceedings of the 21st ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, ser. I3D '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3023368.3023378

[48] C. Peters, "Free blue noise textures," http://momentsingraphics.de/BlueNoise.html.

[49] P. W. D. Z. Jiho Choi, Jim Kjellin, "Ray traced reflections in 'wolfenstein: Youngblood'," NVIDIA, MachineGames, MachineGames, 2020, https://www.gdcvault.com/play/1026723/Ray-Traced-Reflections-in-Wolfenstein.

[50] L. J. F. Pedersen, https://github.com/playdeadgames/temporal.

[51] haarm Pieter Duiker, https://pt.slideshare.net/hpduiker/filmic-tonemapping-for-realtime-rendering-siggraph-2010-color-course.

Appendix A

Code of Project

Source Code A.1: Shading Functions

```
//Cook Torrance functions
vec3 evalFresnelSchlick(vec3 f0, vec3 f90, float cosTheta)
{
        // Clamp to avoid NaN if cosTheta = 1+epsilon
    return f0 + (f90 - f0) * pow(max(1.0f - cosTheta, 0.0f), 5.0f);
}

float DistributionGGX(float NdotH, float alpha )
{
        float a2 =  alpha * alpha;
        float d = (NdotH * NdotH) * (a2-1.0f) + 1.0f;
        return a2 / (d * d * PI);
}

float smithLambda(float wDotN, float alpha)
{
    float wDotN2 = wDotN * wDotN;
    float tanSqrd = max((1-wDotN2),0.0f) / wDotN2;

    return 0.5f * (-1 + sqrt(1 + alpha * tanSqrd));
}

float SmithHeightCorrelated(float w0DotN, float wiDotN, float alpha)
{
    if (w0DotN <= 0 || wiDotN <= 0)
        return 0;

    return 1.0/(1.0+smithLambda(w0DotN,alpha)+smithLambda(wiDotN,alpha));
}

//calculates the necessary components for the shading for this specific w0, wi and point
ShadingData getShadingData(vec3 albedo, float metal, float roughness,
                           vec3 w0, vec3 wi, vec3 normal)
{
    ShadingData sd;
    sd.metal = metal;
    sd.roughness =  max(roughness, EPSILON);
    sd.alpha = max(sd.roughness * sd.roughness, 0.0064f);
    sd.w0 = w0;
    sd.wi = wi;
    sd.normal = normal;
```

```
    float f = (1.5f - 1.0f) / (1.5f + 1.0f);
    float F0 = f * f;
    //Equivalent to multiplying kd by (1.0 - metal)
    sd.diffuseAlbedo = mix(albedo, vec3(0.0f), metal);
    //reflectance at normal incidence
    sd.specularAlbedo = mix(vec3(F0), albedo, sd.metal);

    sd.h = normalize(w0 + wi);
    sd.wiDotN =  max(dot(wi, normal), 0.0f);
    sd.w0DotN = max(dot(w0, normal), 0.0f);
    sd.hDotN = max(dot(sd.h, normal), 0.0f);
    sd.hDotWi = max(dot(sd.h, wi), EPSILON);
    sd.w0DotH = max(dot(w0, sd.h), 0.0f);
    return sd;
}

//calculates the specular shading
SpecularData getSpecularData(ShadingData sd)
{
    SpecularData specData;
    //Normal distribution function
    specData.NDF = DistributionGGX(sd.hDotN, sd.alpha);
    //Geometry Function
    specData.G = SmithHeightCorrelated(sd.w0DotN, sd.wiDotN, sd.alpha);
    //Fresnel
    specData.F = evalFresnelSchlick(sd.specularAlbedo, vec3(1.0f, 1.0f, 1.0f), sd.w0DotH);

    if (min(sd.w0DotN, sd.wiDotN) < 1e-6)
        specData.ggx = vec3(0.0f);
    else {
        specData.ggx = specData.NDF * specData.F * specData.G / (4.0f * sd.w0DotN);
    }

    specData.pdf = max(specData.NDF * sd.hDotN / (4.0 * sd.w0DotN), EPSILON);
    return specData;
}

//calculates the diffuse shading
DiffuseData getDiffuseData(ShadingData sd, SpecularData specData)
{
    DiffuseData dd;
    dd.diffuse = sd.diffuseAlbedo / PI;
    return dd;
```

Source Code A.2: Tiny Encryption Algorithm

```
uvec2 blockCipherTEA(uint v0, uint v1)
{
    uint sum = 0;
    uint iterations = 16;
    const uint delta = 0x9e3779b9;
    const uint k[4] = { 0xa341316c, 0xc8013ea4, 0xad90777d, 0x7e95761e }; // 128-bit key.
    for (uint i = 0; i < iterations; i++)
    {
        sum += delta;
        v0 += ((v1 << 4) + k[0]) ^ (v1 + sum) ^ ((v1 >> 5) + k[1]);
        v1 += ((v0 << 4) + k[2]) ^ (v0 + sum) ^ ((v0 >> 5) + k[3]);
    }
    return uvec2(v0, v1);
}
```

Source Code A.3: Importance Sampling

```
%refs
%https://www.cs.cornell.edu/~srm/publications/EGSR07-btdf.pdf
%https://schuttejoe.github.io/post/ggximportancesamplingpart1/
vec3 importanceSampleGGX(vec2 Xi, float alpha, vec3 normal, vec3 w0)
{
    float phi = 2.0 * PI * Xi.x + rand(normal.xz) * 0.1;
    float cosTheta = sqrt((1.0 - Xi.y) / (1.0 + (alpha*alpha - 1.0) * Xi.y));
    float sinTheta = sqrt(1.0 - cosTheta * cosTheta);

    //from spherical coordinates to cartesian coordinates
    vec3 H;
    H.x = cos(phi) * sinTheta;
    H.y = sin(phi) * sinTheta;
    H.z = cosTheta;

    // Tangent space
    vec3 up = abs(normal.z) < 0.999 ? vec3(0.0, 0.0, 1.0) : vec3(1.0, 0.0, 0.0);
    vec3 tangentX = normalize(cross(up, normal));
    vec3 tangentY = normalize(cross(normal, tangentX));

    // Convert to world Space
    vec3 h = normalize(tangentX * H.x + tangentY * H.y + normal * H.z);

    vec3 wi = reflect(w0, h);
```

```
    if(dot(wi, normal) > 0.0f) {
        return wi;
    }
    else {
        return vec3(0.0f);
    }
}
```

Source Code A.4: Uniform Cone Sampling

```
vec3 uniformSampleCone(float thetaMax, float r1, float r2) {
        float cosThetaMax = cos(thetaMax * PI / 180.0);
        float cosTheta = 1 - r1 + r1 * cosThetaMax;
        float senTheta = sqrt(max(0.0,1.0 - cosTheta * cosTheta));
        float phi = r2 * PI * 2.0;

        return vec3(cos(phi) * senTheta, sin(phi) * senTheta, cosTheta);
}
```

Source Code A.5: Coordinate Frame Creation

```
//Create coordinate frame with z vector pointing towards N
mat3 createCoordinateFrame(vec3 N) {
        vec3 dx0 = vec3(0.0f,N.z,-N.y);
        vec3 dx1 = vec3(-N.z,0.0f,N.x);
        vec3 dx = normalize(dot(dx0,dx0) > dot(dx1,dx1) ? dx0 : dx1);
        vec3 dy = normalize(cross(N,dx));
        return mat3(dx,dy,N);
}
```

Source Code A.6: Hit Point Velocity Estimation

```
    vec3 fakeHitPoint = texture(fakeHitPointsImage, inUV).xyz;
    vec4 prevProjected = params.prevFrameProj *
                         params.prevFrameView *
                         vec4(fakeHitPoint, 1.0);
    prevProjected.xyz /= prevProjected.w;

    vec4 currProjected = params.currFrameProj *
                         params.currFrameView *
                         vec4(fakeHitPoint, 1.0);
    currProjected.xyz /= currProjected.w;
```

```
prevProjected = prevProjected * 0.5f + 0.5f; // convert from [-1,1] to [0,1]
currProjected = currProjected * 0.5f + 0.5f; // convert from [-1,1] to [0,1] range

vec2 displacement = currProjected.xy -
                    prevProjected.xy -
                    vec2(params.jitter.x, params.jitter.y);
```

Source Code A.7: Temporal Accumulation Reprojection Tests

```
bool checkNormals(vec3 prevNormals, vec3 currNormals) {
    return dot(prevNormals, currNormals) > cos(PI / 4);
}

bool checkIds(int prevId, int nextId) {
    return prevId == nextId;
}

bool checkDepth(vec2 prevUv) {
    float currDepth = texture(currDepthImage, inUV).x;
    float prevDepth = texture(prevDepthImage, prevUv).x;
    float ddxDepthCurr = abs(texture(currDepthImage, inUV).y);
    float ddyDepthCurr = abs(texture(currDepthImage, inUV).z);

    return !(abs(prevDepth - currDepth) / (max(ddxDepthCurr, ddyDepthCurr) + 1e-2) > 30.0);
}
```

Source Code A.8: Temporal Accumulation Neighbouring Pixels Test

```
for(int i = -1; i <= 1; i++) {
        for(int j = -1; j <= 1; j++) {

                //(...)
                //(test sample)
                //(...)

                res += texture(prevImage, uv).rgb;
                colorMom.xy += texture(prevMoments, uv).xy;
                numValidSamples++;
        }
}
```

Source Code A.9: History Rectification

```
//calculate color min (cmin) and color max (cmax) in pixel neighborhood
vec3 colorAvg = vec3(0.0);
vec3 colorVar = vec3(0.0);
for(int i = -1; i <= 1; i++) {
    for(int j = -1; j <= 1; j++) {
        vec2 uv = inUV + vec2(texelSize.x * i, texelSize.y * j);
        vec3 color = RGBToYCgCo(texture(currImage, uv).xyz);
```

```
        colorAvg += color;
        colorVar += color * color;
    }
}
colorAvg /= 9.0f;
colorVar /= 9.0f;
vec3 sigma = sqrt(max(vec3(0.0f), colorVar - colorAvg * colorAvg));
vec3 colorMin = colorAvg - params.colorBoxSigma * sigma;
vec3 colorMax = colorAvg + params.colorBoxSigma * sigma;
```

Source Code A.10: AABB Clipping

```
vec3 clipAABB(vec3 cmin, vec3 cmax, vec3 colorAvg, vec3 prevColor) {
        vec3 r = prevColor - colorAvg;
        vec3 rmax = cmax - colorAvg.xyz;
        vec3 rmin = cmin - colorAvg.xyz;

        if (r.x > rmax.x + EPSILON)
                r *= (rmax.x / r.x);
        if (r.y > rmax.y + EPSILON)
                r *= (rmax.y / r.y);
        if (r.z > rmax.z + EPSILON)
                r *= (rmax.z / r.z);

        if (r.x < rmin.x - EPSILON)
                r *= (rmin.x / r.x);
        if (r.y < rmin.y - EPSILON)
                r *= (rmin.y / r.y);
        if (r.z < rmin.z - EPSILON)
                r *= (rmin.z / r.z);

        return colorAvg + r;
}
```

Source Code A.11: Edge-Avoiding À-Trous Wavelet Transform

```
//if separable blur, each kernel side has 0 width alternatively
if(params.useSeparatedBilateral) {
        if(params.currBlurIteration % 2 == 0)
                kernelSizeX = 0;
        else
                kernelSizeY = 0;
}

(...)

//visit around the center of the pixel
for (int i = -kernelSizeX; i <= kernelSizeX; i++) {
        for (int j = -kernelSizeY; j <= kernelSizeY; j++) {

                vec2 uv = coords +
                      vec2(texelSize.x * float(i) * blurScale,
                       texelSize.y * float(j) * blurScale);

                //check if sample is outside of the screen
                bool outside = (uv.x < 0.0f || uv.y < 0.0f
                             || uv.x > 1.0f || uv.y > 1.0f);
                if(!outside) {
                        float weight = kernelWeight[abs(i)] * kernelWeight[abs(j)];
                        vec4 color = texture(currentImage, uv);
                        vec3 normal = texture(normalImage, uv).xyz;
                        float depth =  texture(depthDerivativesSampler, uv).x;

                        //calculate edge weights and multiply
                        //+ store them in the weight variable
                        if(params.useNormalWeight) {
                                float w = normalWeight(originNormal, normal,
                                params.reflectionNormalSigma);
                                weight *= w;
                        }

                        if(params.useLumaWeight) {
                                float w = luminanceWeight(originColor.xyz, originVar,
                                color.xyz);
                                weight *= w;
                        }
```

```
                        if(params.useDepthWeight) {
                                float w = depthWeight(depth, originDepth, phiDepth *
                                length(vec2(i, j)));
                                weight *= w;
                        }

                        sumColor += color.xyz * weight;
                        varSum += color.w * weight * weight;
                        weightSum += weight;
                }
        }
}
```

Source Code A.12: À-Trous Edge Functions

```
float normalWeight(vec3 originNormal, vec3 normal) {
        return pow(max(0.0f, dot(originNormal,normal)),
        params.reflectionNormalSigma);
}

float luminanceWeight(vec3 originColor, float var, vec3 color) {
        float numerator = abs(luma(originColor) - luma(color));
        float varEps = 1e-10;
        float denominator = params.reflectionsLuminanceSigma *
                                sqrt(max(var + varEps, 0.0f));
        return exp(-numerator / denominator);
}

float depthWeight(float depth, float originDepth, float phiDepth) {
        float weightZ = (phiDepth == 0.0) ? 0.0f :
                        abs(originDepth - depth) / phiDepth;
        return exp(-max(weightZ * params.depthSigma, 0.0f));
}
```