MASTER THESIS GAME AND MEDIA TECHNOLOGY

Physically Accurate Noise Free Real-time Rendering

Author:
Mauro van de Vlasakker

Supervisor:
Dr. Robby T. Tan

THESIS NUMBER:
ICA-3857662

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

at the

DEPARTMENT OF INFORMATION AND COMPUTING SCIENCES
FACULTY OF SCIENCE,
UTRECHT UNIVERSITY

Universiteit Utrecht

November 11, 2013

Abstract

This is the master thesis project performed by Mauro van de Vlasakker under supervision of Dr. Robby T. Tan at Utrecht University. Physically based rendering is a widely studied topic in the field of computer graphics. The goal is to find the average color of each pixel by solving the rendering equation. The rendering equation describes light transport mathematically. Path tracing is a way to solve the rendering equation and is done by shooting rays from the camera through each pixel on the screen. Many rays are needed to get a good estimate for the final pixel color. If the number of rays per pixel is low, this will show up as noise in the image. This project is focused on post processing the output image where the goal is to make the post process fast enough to enable real-time (24 fps) path traced scenes.

C++ and OpenCL are used to implement a path tracer which is used as a platform to implement the filter. Random parameter filtering is implemented to directly compare our filters quality with theirs. The main problems we solve are: real-time performance, filtering reflections/refractions and filtering complex Monte Carlo effects like soft shadows and depth of field. Using the variance in scene information we can detect noise. By splitting direct and indirect illumination we can apply separate filtering on each to achieve better results. Skylights and sky-boxes are taken into account since they usually have no normals and are infinitely far away which can cause the filter to over blur parts of the scene.

We evaluate the speed of the filter at different resolutions with different test scenes. The results show that the speed of the filter is fast enough to achieve real-time performance and scales linearly with resolution. Then, the filter is carefully evaluated to determine the optimal filtering parameters for all test scenes. With the optimal parameters we compare our filter against Random Parameter Filtering in terms of quality. We show that our filter is able to filter reflection/refractions, depth of field, soft shadows and sky-boxes/skylights in real-time. For future work there are some interesting extensions that can be made such as adaptive sampling and detecting when the filtering process can stop.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Since the introduction of computer graphics, many fields of study and entertainment find it useful to use a virtual world for visualization. In the early days, computer graphics was mostly used in production movies and later for computer games, architectural design and other areas. There are two major methods to display computer generated images namely: rasterization and ray-tracing. The first is mainly focused on speed and the lather on accuracy i.e. games vs. movies. With increasing computational power it becomes viable to use ray-tracing in computer games [1, 2, 3]. The main advantage of ray-tracing is the capability to simulate complex effects like depth-of-field, motion blur and global illumination without the need of approximations. In recent years many researchers have investigated ways to speed-up ray-tracing [4, 5, 6, 7].

The most common way to generate global illumination and other effects such as depth-of-field and motion blur is path tracing proposed by James Kajiya in 1986 [8]. Path tracing requires many rays per pixel to get a smooth and noise-free result. In recent years path tracing has become increasingly fast, but not fast enough to be noise-free in real-time. The Brigade renderer from Otoy [9] is currently one of the fastest real-time path tracers out there focused on games. Several techniques like adaptive sampling and image filtering have been developed to deal with the noise, but they are often too slow to be used in real-time sceneario's [10, 11, 12, 13]. Until hardware becomes fast enough to produce noise-free path tracing in real-time it is necessary to have a filter that is fast enough, while still preserving detail, to be used in real-time path tracing (24 frames/second).

1.1 Goals and Constraints

The driving force behind this projects are the recent advances in real-time ray tracing that make it possible to use path tracing for interactive applications like games. The reason we want to use path tracing is that it allows for very complex effects to be simulated as opposed to modern day games that use complex approximations to get a nice result. The main problem with path tracing is the noise at low sampling rates so this will be the focus of the project. To achieve real-time performance we will focus on GPU as the main platform of implementation to keep the CPU free for other processes. The path tracer will be used as the backbone for implementations of the filtering algorithms. The path tracer and filter are subjected to the following constraints:

- Run at a frame rate of at least 24 frames per second (fps)
- Render simple and complex geometry i.e. spheres and triangle meshes

- Support for Diffuse, Specular, Refractive and Glossy materials
- Support for several light sources including area lights and sky lights
- The filter should run as a post-process next to the path tracer
- The filter has to be fast enough to allow the path tracer to keep its constraints
- The filter has to be similar in quality to recent filtering algorithms at low sampling rate i.e. 8 samples (rays) per pixel

The possible applications for fast and qualitative filtering algorithms are:

- Noise-free real-time path traced games
- Fast and noise-free pre-visualization for production rendering without the need for render farms

The main goal for this project is to *create a filter that reduces the noise to acceptable levels at low sampling rates (8 spp), while preserving real-time performance*. To achieve this goal we need to investigate what GPGPU language to use and what type of filter we want use. Next section will shortly discuss the thesis outline.

1.2 Thesis Outline

This thesis is divided into three main parts. Part I discusses background information and work related to path tracing and image filtering. In chapter 2 we discuss physically based rendering, path tracing and GPGPU. The focus of this chapter will be on the background theory. Chapter 3 will be a short overview of existing filters and their background.

Part II will contain the main contributions of this thesis and will be focused on implementation. Chapter 4 will discuss the implementation of the path tracer using OpenCL. This implementation is the basis for Random parameter filtering (chapter 5) and the new data driven filter (chapter 6). Random Parameter Filtering is the main paper that is the basis for my data driven filter. The data driven filter will solve several problems that Random Parameter Filtering has. The focus will be on speed since the filter is used for real-time applications.

Part III will discuss the results of the data driven filter and compares it with Random Parameter Filtering in terms of quality. The data driven filter will also be evaluated on speed an quality.

Part I

Preliminaries and Related work

Chapter 2

Physically based rendering

This chapter is an introduction to various topics related to this thesis. First we introduce some basic concepts related to light transport in section 2.1. After this we will shortly discuss the rendering equation (2.2), Monte Carlo sampling (2.3) and path tracing (2.4). In the last section a short introduction on GPGPU and OpenCL is given (2.5).

2.1 Radiometry

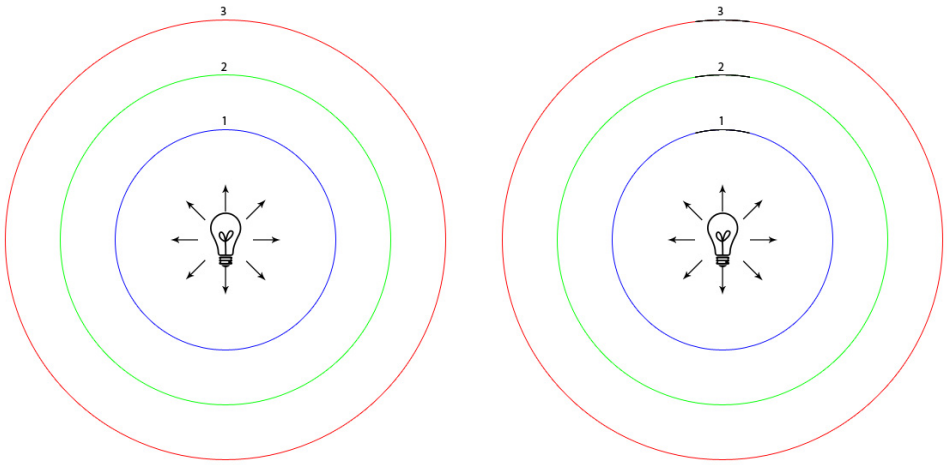
Radiometry is a way to describe the flow of energy through space. Since we confine ourselves to computer graphics this energy represents radiance. In radiometry, light is a packet of energy with a position, direction and wavelength. The wavelength of the photon represents the energy state it is in i.e. its color. Radiometry makes certain assumptions about light transport that limits its use (More detail in PBRT [14]):

- Linearity: The output is the sum of inputs e.g. combining two colors to make one color.
- Energy conservation: A photon scattering around a scene will never gain energy unless it hits a light source.
- No polarization: Electromagnetic properties of light are ignored. The distribution of colors (wavelengths) is not ignored.
- No fluorescence or phosphorescence: Every color will behave the same.
- Steady state light: Distribution of light through the scene does not change over time.

These assumptions makes it difficult but not impossible, to model diffraction and interference that causes the rainbow colors on cd's and soap bubbles respectively.

2.1.1 Flux

Flux is the total amount of energy passing through a region of space per unit time. The units are Joules per second (J/s) or Watts (W) and the symbol Φ is often used. Figure 2.1(a) depicts how the flux does not change when measured at different surfaces/regions of the space (assuming steady state light). From the figure we can see that the total amount of flux stays the same everywhere, but the amount of flux going through a specific part of the sphere does change.



(a) Radiant flux going through 1, 2 and 3 is the same

(b) Irradiance measured at 1, 2 and 3

Figure 2.1

2.1.2 Irradiance and radiant exitance

Irradiance is the amount of flux arriving at a unit area over a unit of time. Figure 2.1(b) depicts this in a similar way as before, but now the measurements are done on a part small of the sphere. When the measurements are done on a larger surface area, the irradiance will be lower. Radiant exitance is the amount of flux leaving a unit area over unit time. This is essentially the same measure describing two different situations. Irradiance is usually written as E and radiant exitance as M. Both have units of Watts per unit area (W/m^2). We can say that $E = \Phi/A$ where A is the unit area.

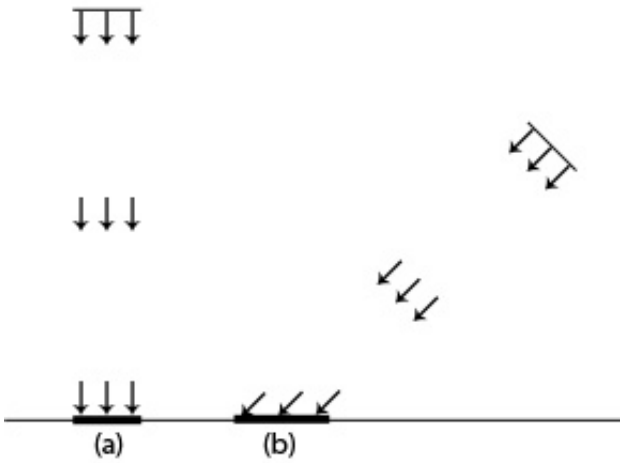


Figure 2.2: Irradiance at different angles

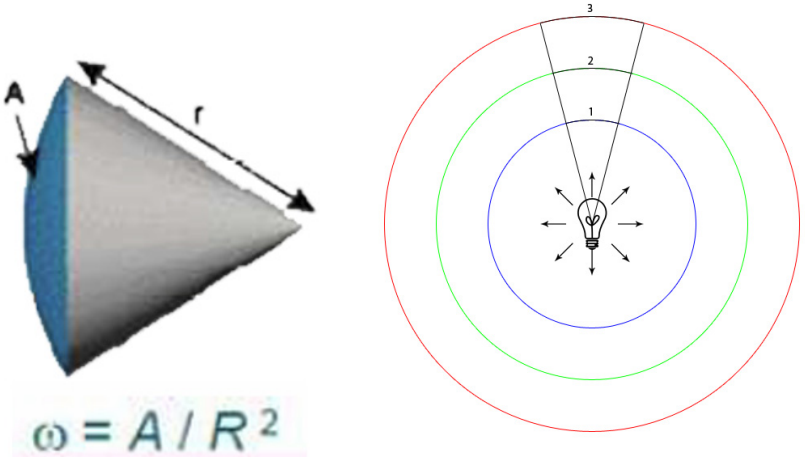
Irradiance can also be described as the density of flux on a unit area of a surface, which is a small portion of the total surface area (dA). The portion of flux that dA receives is $d\Phi$. If the direction of the received $d\Phi$ is perpendicular to dA (Figure 2.2a), the irradiance is $E = \Phi/A$. When we receive $d\Phi$ at an angle with the surface normal of dA (Figure 2.2b), the irradiance will

be scaled with a factor of $\cos(\theta)$. Since radiant exitance is $d\Phi$ leaving dA we can write can write the equation as:

$$M = E = \frac{\cos(\theta)\Phi}{A} = \frac{d\Phi}{dA} \tag{2.1}$$

2.1.3 Intensity

Intensity I is the amount of $d\Phi$ over a solid angle $d\omega$. A solid angle is the 3-dimensional version of an angle. It is the area on a unit sphere while looking in a specific direction. It can be seen as a cone pointing to a direction on the unit sphere. Figure 2.3(a) depicts the cone that represents a solid angle.



(a) Solid angle (b) Intensity

Figure 2.3

Figure 2.3(b) shows an example of intensity in two dimensions. The intensity stays the same because the amount of flux over that angle does not change, so the amount of $d\Phi$ per $d\omega$ is I also called flux density per unit solid angle:

$$I = \frac{d\Phi}{d\omega} \tag{2.2}$$

Although it is useful to know the basic quantities of radiometry, for the purpose of rendering intensity is only meaningful when dealing with point lights.

2.1.4 Radiance

The final and most important unit in radiometry is radiance L. Radiance is a measure of how much flux is emitted by a reflecting or emitting object from dA towards $d\omega$. Under the assumption of energy conservation we can say that the radiance emitted from a source is the same as the radiance received by a detector observing that source e.g. the eye or a camera. As with irradiance and radiant exitance there are two types of radiance. The radiance arriving at a point is called incident radiance L_i and the radiance leaving a surface at a point is called the outgoing radiance L_o.

In general a reflecting surface change the radiance i.e. radiance gets absorbed. If there is no reflecting surface, energy is conserved:

$$L_o(p, \omega) = L_i(p, -\omega), \tag{2.3}$$

where $L_o(p, \omega)$ is the outgoing radiance at point p in direction ω and $L_i(p, -\omega)$ is the incident or incoming radiance at point p coming from direction $-\omega$. The general way to write radiance is:

$$L = \frac{d^2\Phi}{d\omega dA \cos\theta}, \tag{2.4}$$

where $\cos\theta$ is the angle between the surface normal of dA and $d\omega$.

2.2 Rendering Equation

Light transport is approximated using the rendering equation, introduced by James Kajiya in 1986 [8]. The rendering equation describes how radiance arriving at a point x'' from a surface at point x' relates to the radiance arriving at x' from point x. A geometric example of the equation is depicted in Figure 2.4 and is defined by an integral over surface area by:

$$L(x' \to x'') = L_e(x' \to x'') + \int_M L(x \to x') f_s(x \to x' \to x'') G(x \leftrightarrow x') dA_M(x) \tag{2.5}$$

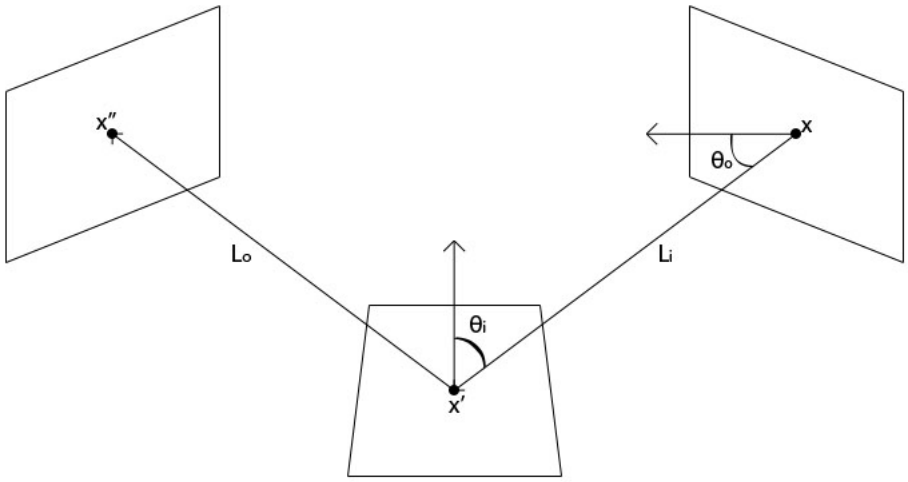


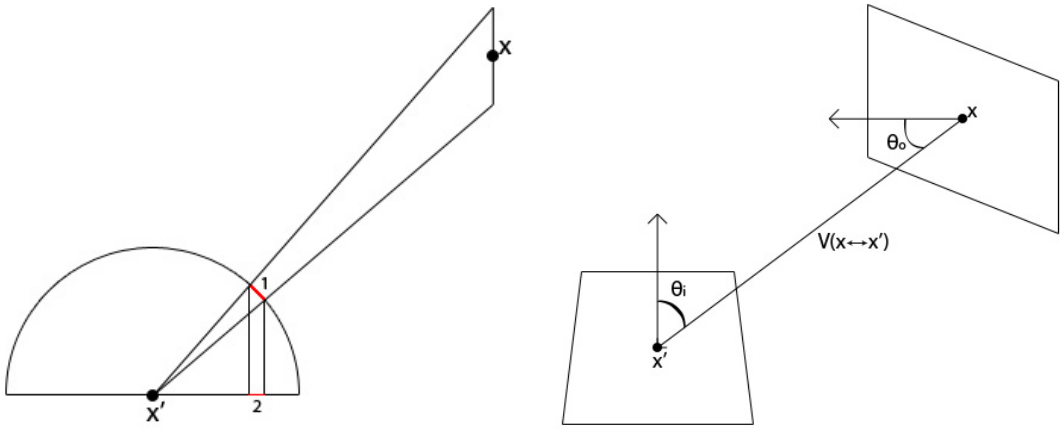
Figure 2.4: Light Transport

In this equation $L(x' \to x'')$ is the radiance arriving at point x'' coming from surface point x'. Radiance arriving at x' comes from multiple surfaces M, where M are all the surfaces in the scene. $L_e(x' \to x'')$ is the radiance that is emitted from surface point x' that arrives at point x'' i.e. a light source. $L(x \to x')$ is the radiance arriving at x' coming from a different surface point x and $f_s(x \to x' \to x'')$ is the Bidirectional Scattering Distribution Function (BSDF) of the surface material at x'. The BSDF describes how much radiance from point x hitting surface point x' is scattered towards x'', which can be transmission or reflection. The BSDF can consist of multiple Bidirectional reflectance distribution functions (BRDF) and/or Bidirectional

transmittance distribution functions (BTDF). $G(x \leftrightarrow x')$ is a geometric term to convert form unit projected solid angle to unit surface area. We need this term because we are interested in surface areas at points x and x' instead of solid angles. Figure 2.5(a) depicts how the unit projected solid angle (1) is converted to unit surface area (2).

$$G(x \leftrightarrow x') = V(x \leftrightarrow x') \frac{|\cos(\theta_o)\cos(\theta_i)|}{\|p - r\|^2} \tag{2.6}$$

In this geometric term, θ_o and θ_i are the angles between the local surface normals and the incoming and outgoing light directions and $V(x \leftrightarrow x')$ is the visibility term that is 1 if x and x' are mutually visible otherwise it is 0. This is illustrated in Figure 2.5(b).



(a) Unit projected solid angle to unit surface area

(b) Geometric term

Figure 2.5

In computer graphics, the light transport equation is often called the rendering equation which is usually expressed as an integral over unit projected solid angles instead of surface areas. A more general way to write the rendering equation is an integral over paths instead of area. This formulation describes the sum over all light carrying paths of different lengths by repeatedly substituting the right hand side of Equation 2.5 into the $L(x \rightarrow x')$ term inside the integral. The goal in computer graphics is to solve this integral to find the radiance arriving at each pixel on the image plane. Path tracing is one way to solve the rendering equation and is discussed in paragraph 2.4.

2.3 Monte Carlo Sampling

The integrals discussed in paragraph 2.2 do not have an analytic solution, so we must solve them using numeric methods. We prefer a numeric method that is independent of the dimensionality of the integral, since the length of a light transport path can be infinitely long i.e. infinity dimensional integral. Monte Carlo integration provides one solution to this problem. This method uses random numbers to evaluate the multidimensional integral and is independent from dimensionality. For example, if we want to compute the amount of reflected radiance on a surface at point x' towards point x'' we must integrate over a certain direction on the unit hemisphere using it's BRDF in order to get the right solution as depicted in Figure 2.6.

The main disadvantage of Monte Carlo sampling is the amount of samples needed to converge to a correct result. The algorithm converges to a correct result at a rate of $O(n^{-1/2})$ i.e. if we want to cut the estimation error by half we need to take four times the amount of samples.

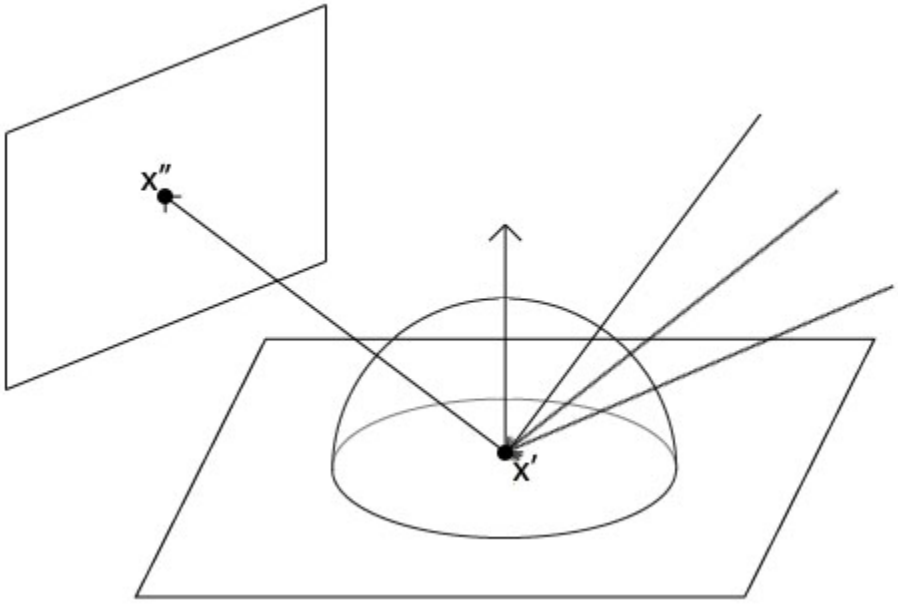


Figure 2.6: Evaluate radiance at x' on a glossy surface, with 3 samples

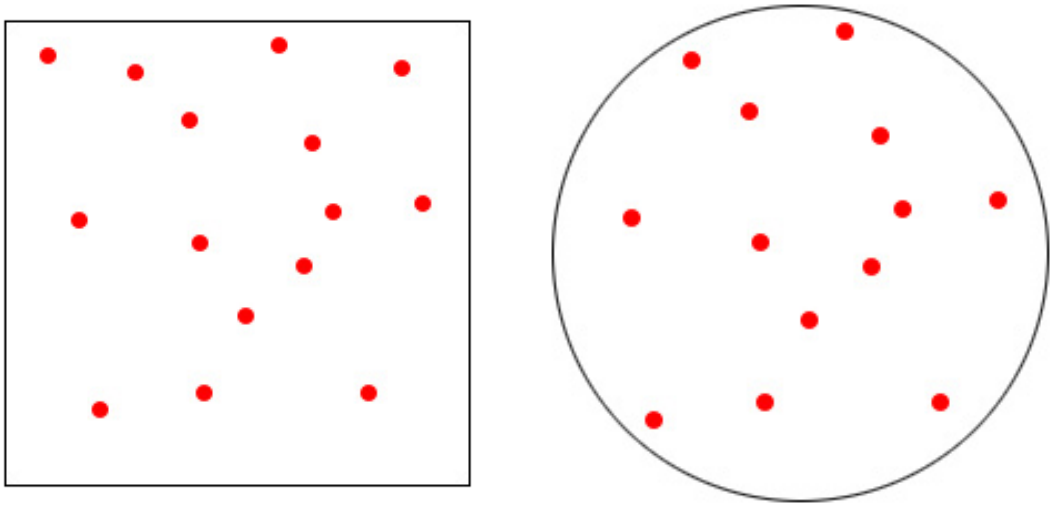
Another example is pixel sampling, where the colors of sub-pixels are used to estimate the final pixel color. The 2-dimensional integral for pixel sampling would be:

$$P(x,y) = \int_{x-0.5}^{x+0.5} \int_{y-0.5}^{y+0.5} f(x,y)dydx, \tag{2.7}$$

where $P(x,y)$ is the pixel. To solve a one-dimensional integral given a supply of uniform random variables $X_i \in [a,b]$ the Monte Carlo estimator is:

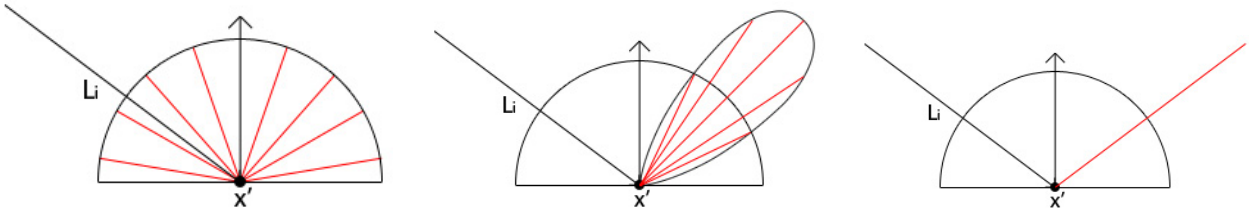
$$F_N = \frac{b-a}{N} \sum_{i=1}^{N} f(X_i), \tag{2.8}$$

where the estimate of F_N ($E[F_N]$) would be the same as the value of the integral. Here N is the number of samples and $f(X_i)$ is the random sample. The proof that the estimator is correct is given in (PBRT chapter 13.2). With this estimator we can solve equation 2.7. Pixel sampling is depicted in Figure 2.7(a). The example shows that taking more samples will give a better estimate of the pixel color. Figure 2.7(b) depicts a similar case where lens sampling is done (assuming a circular lens). Figure 2.8 shows more examples of Monte Carlo sampling that are useful for the purposes of path tracing.



(a) Pixel sampling (b) Lens sampling

Figure 2.7



(a) Diffuse BRDF sampling (b) Glossy BRDF sampling (c) Specular BRDF sampling

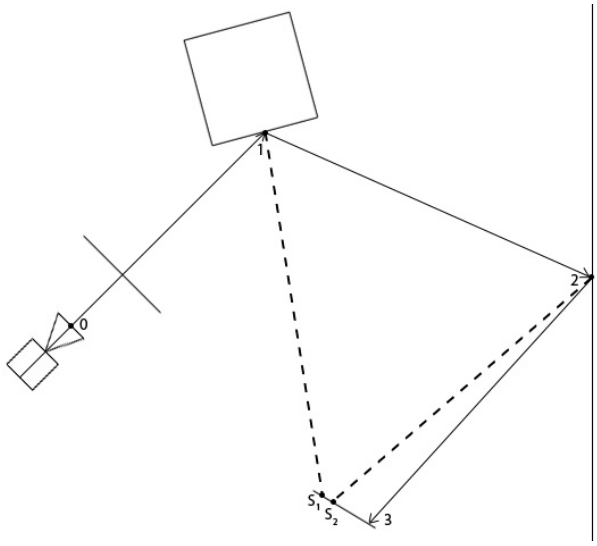
Figure 2.8: BRDF sampling

2.4 Path Tracing

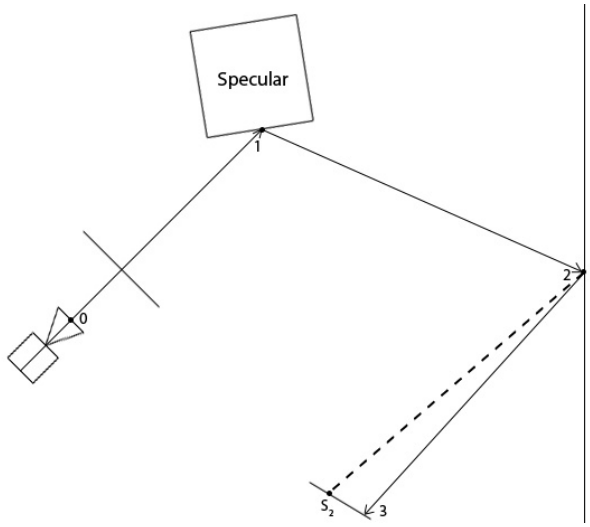
Path tracing was proposed by James Kajiya to solve the rendering equation using Monte Carlo methods. Rays are shot from the eye through the image plane into the scene where they hit objects and eventually reach a light source or terminate. Figure 2.9(a) shows a single path being traced. We start by tracing a path (ray) from the camera through the image plane where the path starts with a depth of zero. After the first bounce the path depth is one and a new direction for the path is calculated. The path is repeatedly extended until it reaches a light source, is terminated by Russian roulette or misses all geometry in the scene. Russian roulette is a way to allow paths to go infinitely deep (with very low probability) and therefore keep the algorithm unbiased. Russian roulette will be explained in 2.4.2

Each time a path is extended, an explicit connection is made with the light source (shadow ray), by taking a random position on the light source. If this connection is possible, the object and light are mutually visible and a complete path is found i.e. path from camera to light. This explicit connection can only be made when a diffuse object is hit since it makes no sense to create an explicit connection when a specular object is hit. Figure 2.9(b) shows why. When a ray hits a specular object, the probability that it will go in the direction of the light source is zero (in the example) since the outgoing direction is the mirrored incoming direction Figure 2.8(c). When

a path reaches the light source, this is called an implicit connection and the path is terminated. Both explicit and implicit connections make a path complete.



(a) Tracing a path with one implicit connection at depth 3 and two explicit connections S_1 and S_2



(b) Tracing a path with specular bounce. At bounce 1 all other directions except for the one displayed have zero probability of occurring

Figure 2.9: Tracing a path from the camera

Every complete path will make a contribution to the color of the pixel that is evaluated. More paths for each pixel mean a better estimate for the final color of the pixel. A more detailed description of the path tracing algorithm will be given in 2.4.2

2.4.1 Path tracing noise

Evaluating only one path for each pixel will give an inaccurate estimate for the final pixel color. The reason that this happens is the way the algorithm does its sampling. Consider the following example: A ray is shot through a random position in the pixel and hits a blue sphere in the scene. The ray bounces around until it terminates or hits a light source. The color of the pixel will become bluish since we hit a blue sphere. Now a second ray is shot through a random position inside the pixel but hits a red sphere that's behind the blue sphere. This ray will contribute red and the average color of these rays will be purple. This situation is depicted in Figure 2.10.

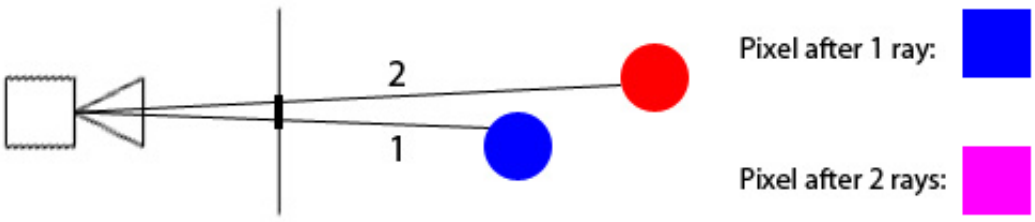


Figure 2.10: Pixel sampling noise

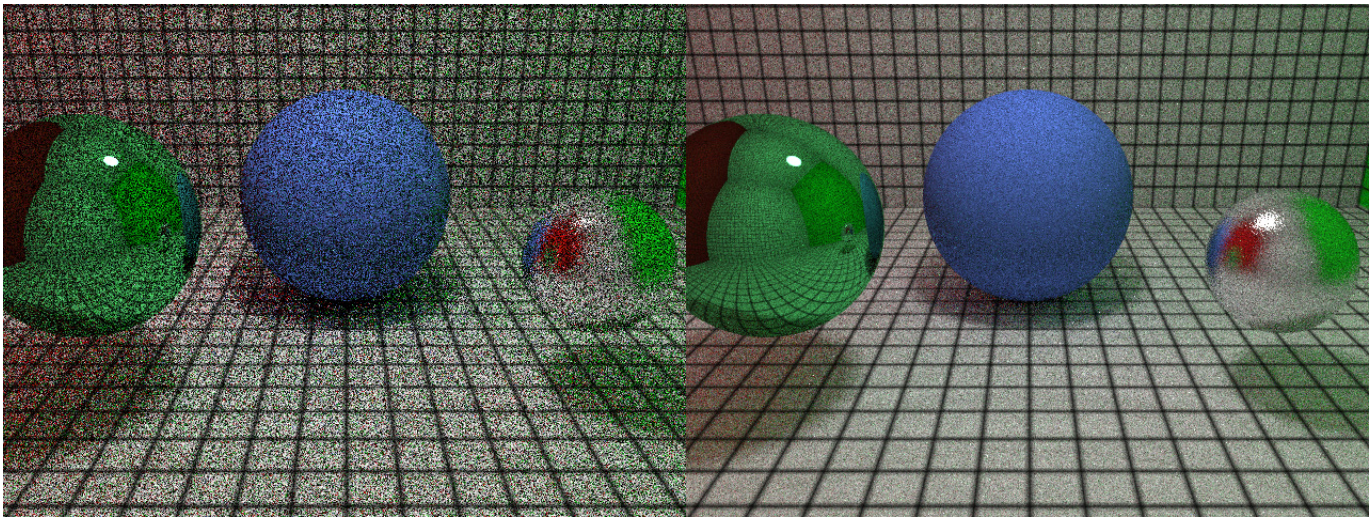


Figure 2.11: 1 spp versus 16 spp

This problem only considers the random position inside the pixel, but there are more random parameters like bounce direction after hitting an object. In order to get rid of the noise we need many samples / paths per pixel (spp) to get a noise free result. Figure 2.11 shows images comparing 1 spp versus 16 spp.

2.4.2 Path tracing algorithm

Brute force path tracing v0

Brute-force path tracing is done by tracing a ray from the camera through a pixel into the scene. At each hit-point the ray is recursively traced into the scene over a hemisphere of directions. This means the chosen direction of the ray can have a probability of zero. The ray will terminate when a light source is hit or does not hit any geometry in the scene. For indoor scenes this means a ray can recurs to a depth of infinity which causes the algorithm to keep going forever.

Russian roulette v0.5

We are interested in rays that contribute radiance to a pixel. This means we can ignore rays that carry little or no radiance thus giving us a way to terminate a ray when it is below a certain radiance threshold. To keep the algorithm unbiased we use Russian roulette, to give low radiance rays a chance to survive. When the weight of a ray is below a certain threshold Russian roulette randomly chooses to revive or terminate a ray.

$$w = \begin{cases} mw \text{ if } \xi \leq \frac{1}{m} \\ \\ 0 \text{ if } \xi > \frac{1}{m} \end{cases}, \tag{2.9}$$

where m determines the survival probability and ξ is a random number between 0 and 1. If we choose $m = 10$ the probability of survival will be 0.1 where after survival the output weight w is scaled by m to account for $9/10$ terminated rays that had some weight but are lost due to termination.

BRDF sampling v0.75

Instead of taking an arbitrary direction on the hemisphere at the hit point, we can take the material (BRDF) into account when calculating a new direction for the ray as depicted in Figure 2.8. This effectively means all directions that give a very low probability of being sampled will be ignored.

Direct light sampling v1.0

After all these changes to the algorithm there is still the problem of finding a light source. To solve this we take light samples when we hit a diffuse object. This is called an explicit connection or shadow ray. This was already explained in the introduction of 2.4.

Path tracing algorithm v1.0 pseudo code

Algorithm 1 shows the path tracing algorithm. The output radiance L is initialized to black and the weight of the path ray to 1 or white. The weight (often referred to as path throughput) keeps track of the total non-absorbed weight of the ray. We keep tracing the ray until it is terminated (line 4). The first step is Russian roulette which only applies when a certain depth is reached. In this example this minimum depth is the maximum depth set by the programmer/user (line 5). If the ray misses all geometry in the scene, the sky is sampled and the algorithm returns the radiance (line 12). When the ray does hit geometry, the BRDF is sampled to update L and w. The new direction the ray gets depends on the BRDF (line 17). If needed the lights are sampled (explicit connection) before we continue to the next iteration of the while loop (line 18).

Algorithm 1 Path tracing algorithm

```
1:  function TracePath(Ray r)
2:      L ← Black
3:      weight ← White
4:      while r ≠ terminated do
5:          if r.depth ≥ maxDepth then
6:              RussianRoulette(r, w)
7:              if w ≤ 0 then
8:                  terminate r
9:                  return L
10:             end if
11:         end if
12:         if !r.hit then
13:             SampleSky(L, w, r)
14:             terminate r
15:             return L
16:         end if
17:         SampleBRDF(L,w,r)
18:         SampleLights(L,w,r)
19:     end while
20:     return L
21: end function
```

2.5 GPGPU

Graphics processing units (GPU's) become increasingly important in many fields of science including computer graphics. In recent years the GPU is used for more general purpose computations like ray tracing. Since the introduction of CUDA by NVIDIA in 2006, it became easier to harness the full power of modern GPU's. With CUDA developers where able to do more general purpose computations on the GPU (General Purpose Graphics Processing Unit GPGPU). A major downside of CUDA is its platform dependency, since it only works on NVIDIA hardware. In 2008 the Khronos Group brought OpenCL to the market. OpenCL is similar to CUDA, but it is not limited to a specific vendor like NVIDIA or AMD. Many researchers prefer CUDA over OpenCL because CUDA is more mature and OpenCL is always lagging a bit behind. However, OpenCL has the major advantage of being vendor/platform independent which makes it the perfect candidate for future work. For this reason my GPU implementations are done with OpenCL.

OpenCL C uses a subset of C99 programming language with some additions like 3-vectors and image buffers. There are four basic models that describe OpenCL:

- Platform model (2.5.1)
- Execution model (2.5.2)
- Memory model (2.5.3)
- Programming model (2.5.4)

2.5.1 Platform model

The platform consists of a host connected to one or more OpenCL compute devices. Each compute device has compute units (CU's) that contain one or more processing elements (PE's). The actual computations are done on these individual processing elements. Each processing element within a compute unit processes one single instruction (Single Instruction Multiple Data SIMD). The platform is depicted in Figure 2.12.

2.5.2 Execution model

The execution of an OpenCL program occurs on the Platform (host program) and the Compute devices (kernels). The host program controls the setup of the devices and the execution of a kernel on a compute device. When the host program executes a kernel, it needs to define a space to execute it in. Each processing element or work-item[1] is put into a work-group that in turn executes one single kernel. When work-items of a single work-group diverge in the code (e.g. if-else statement), other work-items need to wait until the diverged work-items are done executing the code. This causes work-items in a work-group to do nothing until they can continue, which is a waste of resources. This problem is important to keep in mind when implementing algorithms on the GPU.

Each work-item has a unique id within the global space (all work-groups) and local space (a single work-group). This gives a single work-item the capability to process different data compared to other work-items in a work-group. The space in which the work-groups are executed is called NDRange. Figure 2.13 shows the execution model with a 2-dimensional NDRange.

[1]These are the same and are only used to distinguish the Platform model from the Execution model

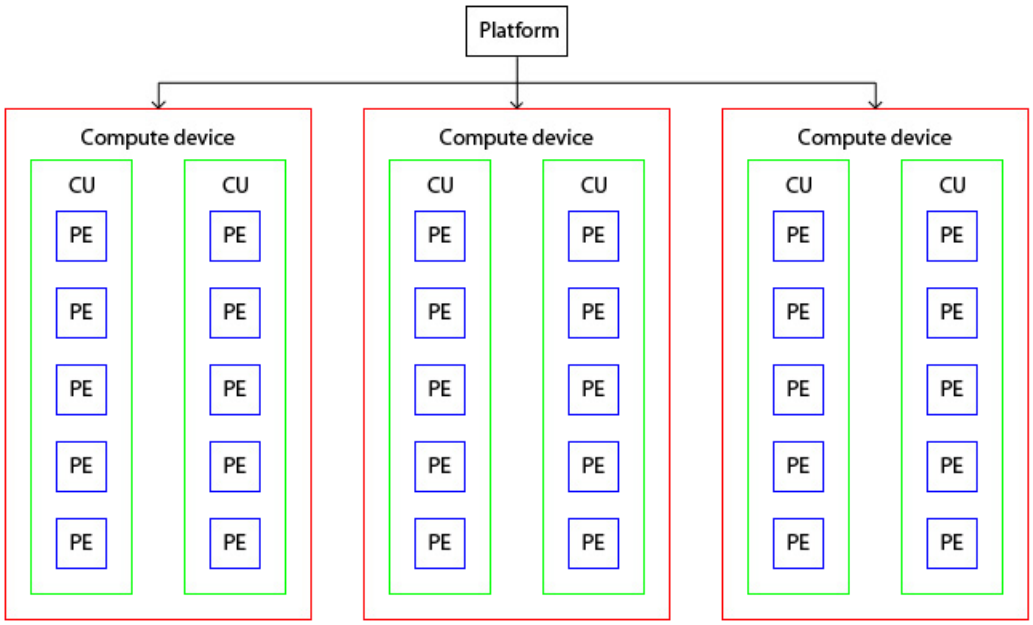


Figure 2.12: Platform model

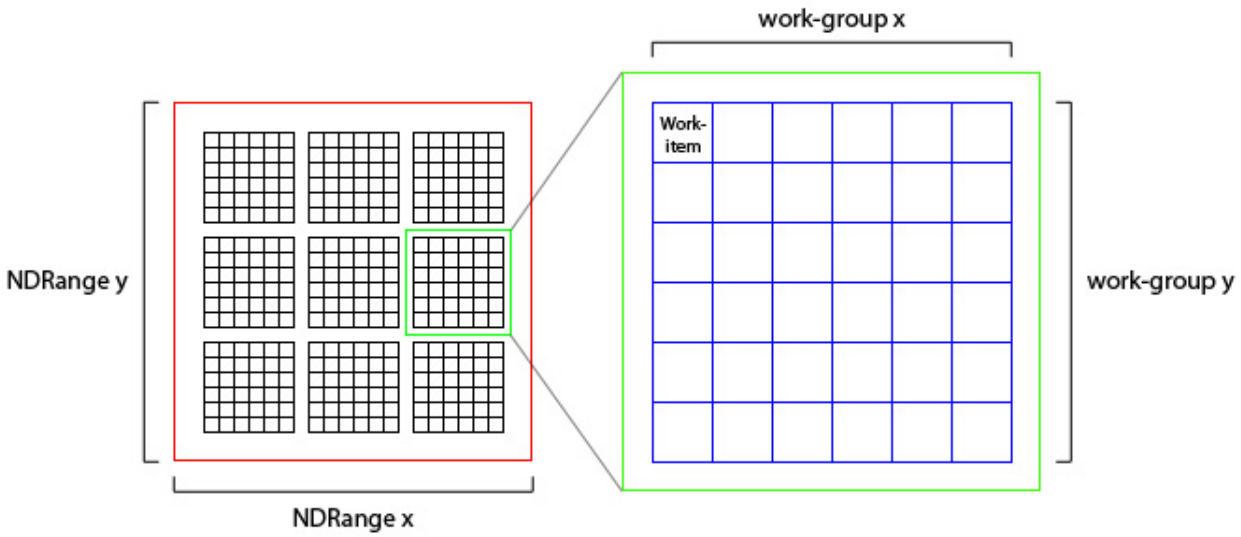


Figure 2.13: Execution model

2.5.3 Memory model

The memory model of OpenCL contains four memory types.

1. Global memory is dynamically or statically allocated in the host program. This memory is slowest but largest kind of memory. Global memory is local to the compute device and is mainly used to allocate large data structures like textures. If the data structure is constant, it should allocated on constant memory if possible.

2. Constant memory is part of the global memory which is read only. The host program has read/write access and the kernel on which it is executed (work-item) can only read

and statically allocate constant memory. Constant memory can be used for small data structures that do not have to change within the lifetime of a work-item. This type of memory is local to the compute device. An advantage of constant memory over global memory is the ability to broadcast data. Broadcasting happens when many threads access one single memory location all at once. When using constant memory, the data will be sent back to all threads at once.

3. Local memory is local to a work-group. This type of memory is fast compared to global memory[2]. It is advised to use this local memory when dealing with shared resources within a work-group. The host program can dynamically allocate local memory, but cannot access it. Only the work-items inside a work-group can read/write and statically allocate it.

4. Private memory is local to a work-item and can only be statically allocated on the work-item. Usually private memory is stored inside the register coupled to this work-item, but if the data is too large it is re-allocated to global memory which significantly slows down performance. It is advised to store only small amounts of data inside private memory to prevent global memory being allocated.

These memory spaces are depicted in Figure 2.14 which shows an additional memory space (L2 cache) that cannot be accessed in any way except by the hardware itself. This memory is used as an intermediate buffer for the global and constant memory. It is important to know these concepts when programming on a GPGPU device.

2.5.4 Programming model

Data parallel is the most common programming model in OpenCL. To give an example: We have a vector of 16 integers. The goal is to calculate the square of each value and store the result in an output vector. We let 16 work-items inside a work-group do the calculation by giving 1 element of the vector elements to each work-item. If the work-item is done with the calculation it stores the result in an output vector. It is useful to try and minimize the amount of memory read/writes that need to be done. Figure 2.15(a) shows an example with actual numbers where memory reads/writes are not optimized and Figure 2.15(B) shows an example where memory reads/writes are 'optimized'.

In general this is a good way to use the local memory, but in this example it gives the unnecessary overhead due to local memory loading. One should think carefully about using local memory. This scheme would work if the work-items need to share the data from the vector. The examples show how every work-item processes one element of the vector. When doing path tracing we can do the same, where each work-item processes one path or one pixel.

When work-items in a single work-group need to synchronize, a barrier can be used. A barrier ensures that each work-item is done before executing more code and needs to be placed inside a part of the code that all work-items reach to prevent dead-locks i.e. no barrier in an if-else statement. OpenCL does not provide a way to synchronize work-groups.

[2] AMD Radeon 7970: L1 cache = 2 TB/s versus DDR5 284 GB/s memory bandwidth

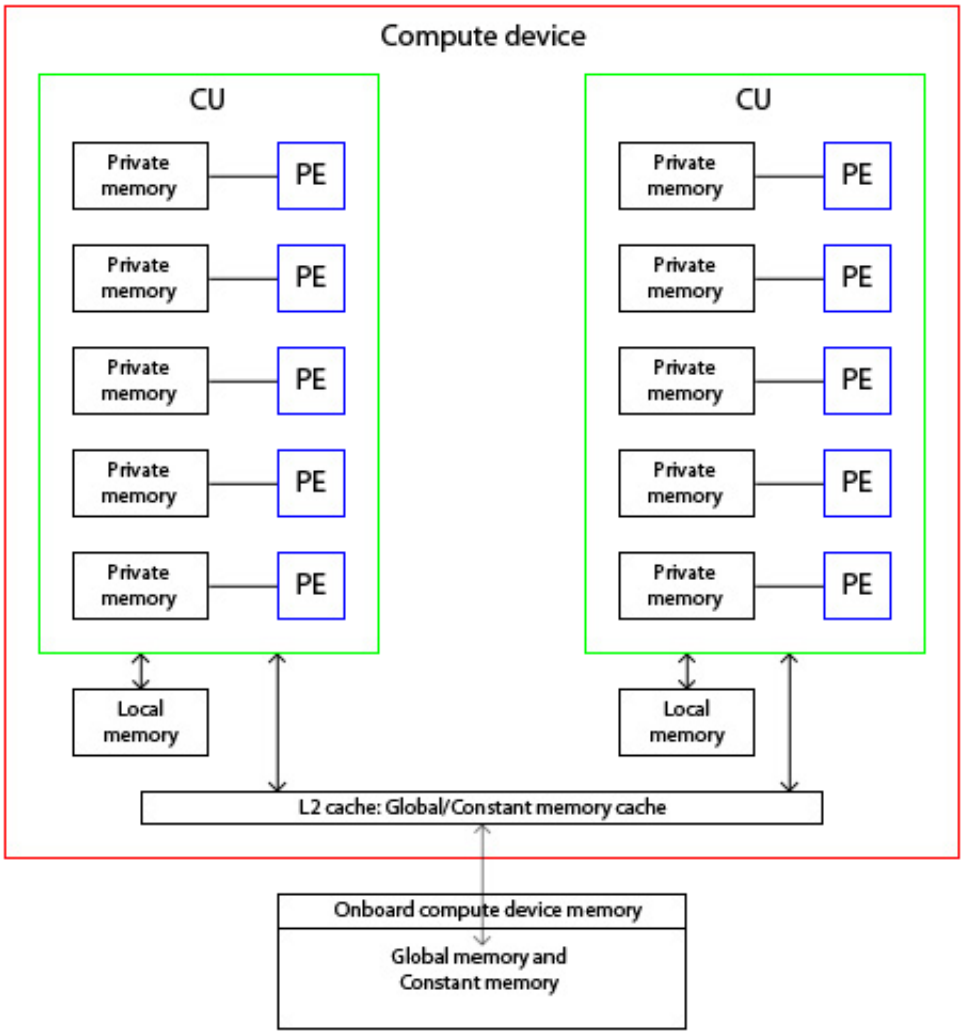
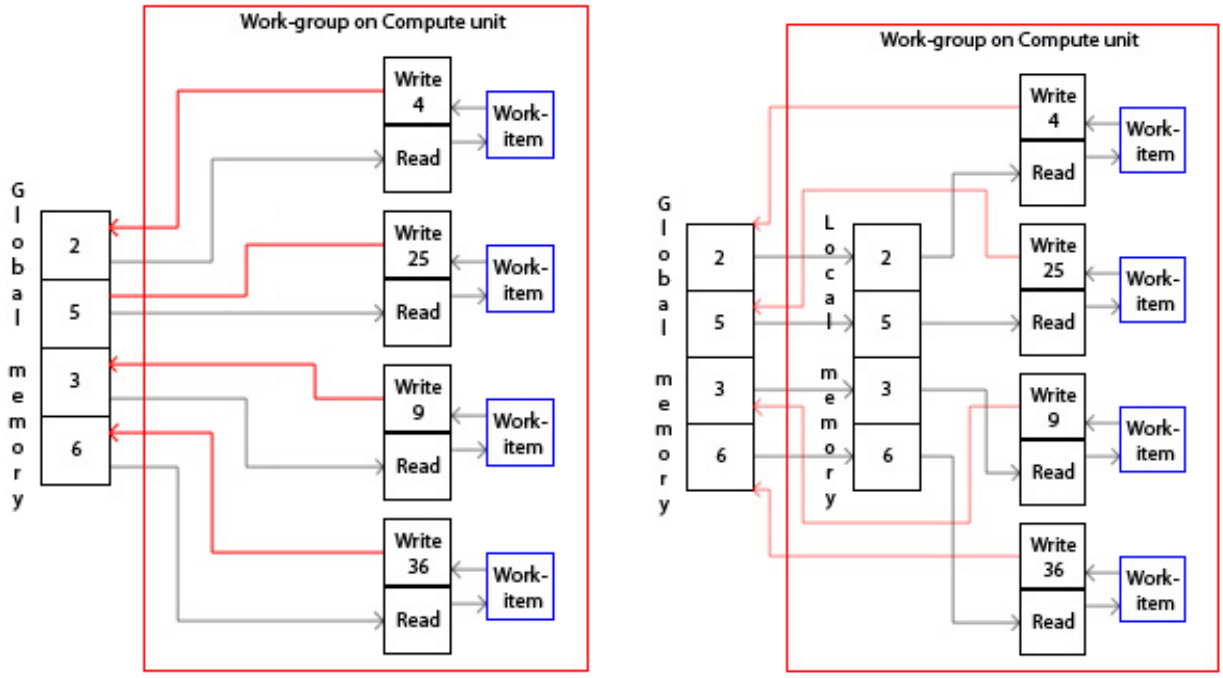


Figure 2.14: Memory model



(a) Un-optimized

(b) Optimized using local memory

Figure 2.15: Two examples that accomplish the same goal

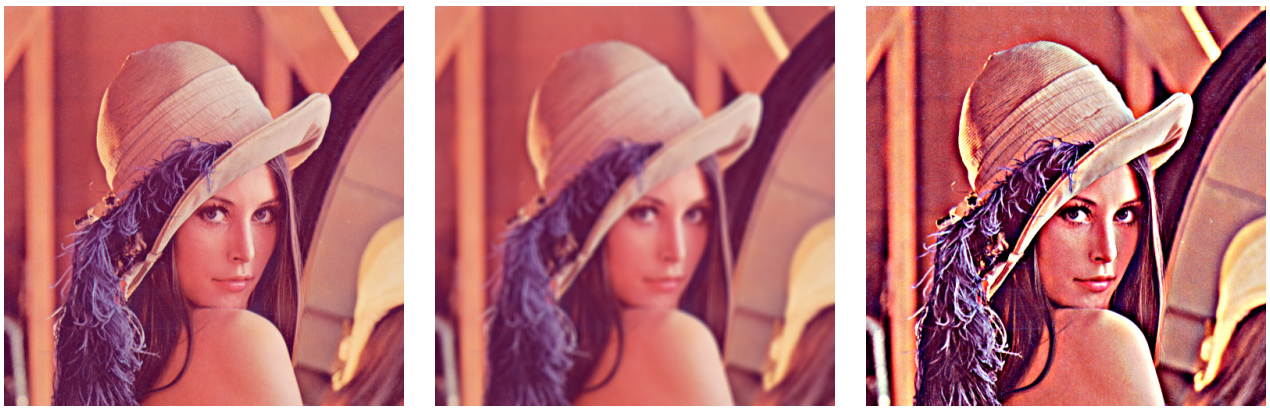
Chapter 3

Related work

Image processing is widely used in computer science to solve several problems like image enhancement, noise reduction, object detection and more. In this chapter we will focus on noise reduction filters. Every filter we discuss has at least a complexity of $O(W \times H)$, where W and H are the screen width and height in pixels respectively. This is because each filter is computed over all pixels in the image. The complexity of the filter increases when a neighborhood of pixels is used to find the filtered color of the output pixel.

3.1 Non-gaussian based filters

Most image filtering algorithms look at neighboring pixels to extract information that can be used in the filtering process. The basic idea is that a neighboring pixel often exhibits similar characteristics like color and gradients. These filtering algorithms are often used for blurring, edge detection or noise removal. Some examples are shown in Figure 3.1.



(a) No filter applied (b) Blur filter applied (c) Edge sharpening

Figure 3.1

This Figure applies several filtering algorithms to a unfiltered noise-free image. If we want to reduce noise in an image, there has to be noise in the input image. In this paragraph, two common filtering algorithms are discussed.

3.1.1 Box filter

The box filter uses a box around the pixel being filtered. The final color of the filtered pixel is the average of all pixels inside this box. The assumption here is that every pixel gets an equal weight assigned to it, which causes the pixel to be blurred. Figure 3.2 shows several custom box filters with different weights applied to the same image. Figure 3.2(a, e) show the unfiltered image, which is an average of itself since all the neighbors get a weight of 0. A box or neighborhood of weights that determine the behavior of the filter is called a filter kernel. In the examples a filter kernel of $3x3$ is used to filter the image. When the kernel size is increased, the computational time increases exponentially since a weighted average of all the pixels in the NxN neighborhood needs to be computed. This means the running time of the box filter is $O(N^2)$, where N is the width of the kernel in pixels.

0	0	0
0	1	0
0	0	0

(a) Unfiltered

0	0	-2
0	2	0
1	0	0

(b) Raised weights

-1	-1	-1
-1	8	-1
-1	-1	-1

(c) Edge weights

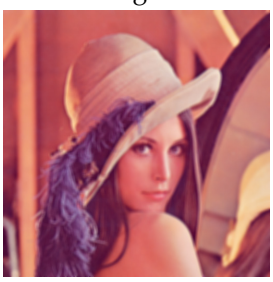
1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

(d) Pure box filter weights

(e) Original image

(f) Raised image

(g) Edge detection



(h) Pure box filter

Figure 3.2

Figure 3.3 shows a graph of weights when applying a box filter. The graph clearly shows how the box filter does not care about the distance of the neighboring pixels, since the weights for all pixels in the kernel are the same.

3.1.2 Triangle filter

Just like the box filter, a triangle filter applies a kernel with specific weights to the image. The weights of the kernel are linearly interpolated and depend on the size of the kernel. Figure 3.4 shows a graph of weights when applying a triangle filter. The basic idea of this filter is that pixels further away from the center get a smaller weight.

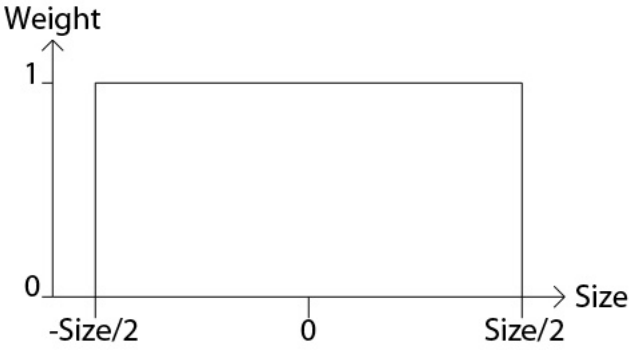


Figure 3.3

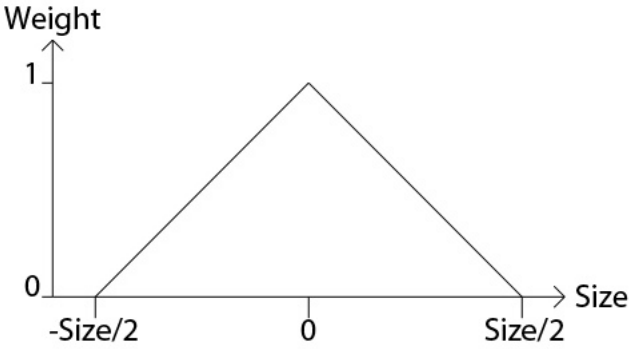


Figure 3.4

3.2 Gaussian based filters

The Gaussian function is named after "Carl Friedrich Gauss" and is written in its 2-dimensional circular form as:

$$f(x,y) = \frac{1}{2\pi\sigma^2} exp(-\frac{(x-\mu_x)^2 + (y-\mu_y)^2}{2\sigma^2}), \tag{3.1}$$

where σ is the standard deviation and μ_x and μ_y are the means of x and y. A 1-dimensional Gaussian function (probability density function pdf) is often depicted as a 2 dimensional bell-shaped curve as depicted in Figure 3.5.

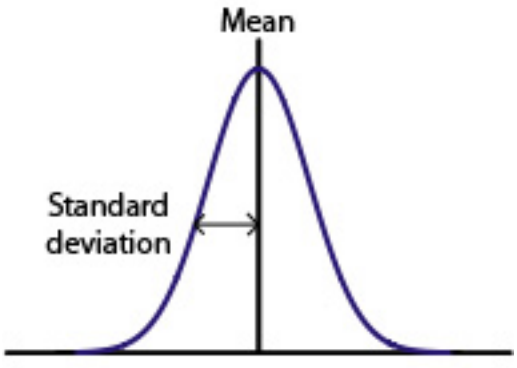


Figure 3.5: Gaussian function

This particular Gaussian function is a Normal distribution which means all probabilities

sum up to one. When a Gaussian is normally distributed 68% of all the probabilities are within one σ from the mean, 95% of all probabilities are within 2σ and 99.7% within 3σ. When σ gets larger the Gaussian function will get wider.

3.2.1 Gaussian filter

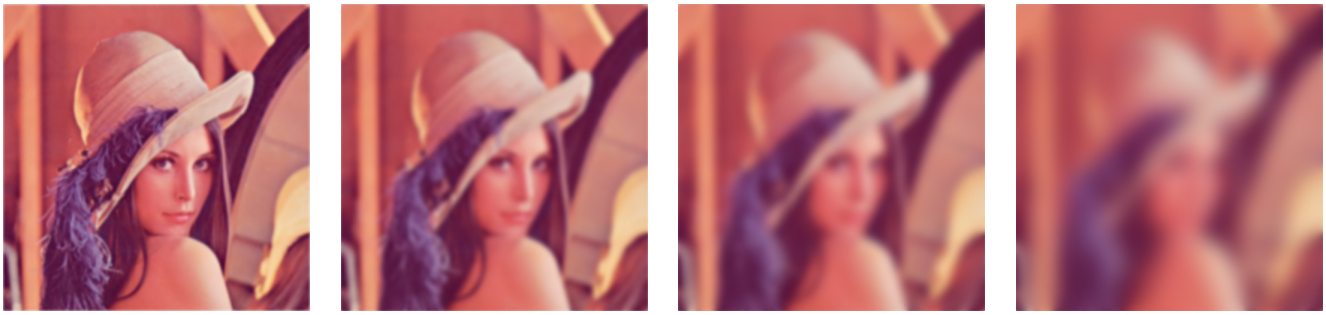
Gaussian filtering is based on the same principal as the triangle filter. Pixels inside the kernel get a lower weight based on their distance from the center i.e. mean μ. Since the filtering kernel is always centered on the pixel that's being filtered, μ is 0. The Gaussian filter does not care about edges in the image and for this reason it is often called the Gaussian blur filter.

$$G(x, y) = \frac{1}{2\pi\sigma^2} exp(-\frac{(x)^2 + (y)^2}{2\sigma^2}). \tag{3.2}$$

When applying a Gaussian filter kernel to a pixel, we loop over all pixels in the image. The weight that the neighboring pixel contributes is calculated with equation 3.2.

$$F(I)_p = \sum_{q \in S} G_\sigma(|p - q|) I_q, \tag{3.3}$$

where $F(I)_p$ is the filtered color of pixel p, I_q is the color of neighborhood pixel q and S is the image containing all pixels. σ basically defines the extend of the neighborhood i.e. the width of the kernel, since all pixels more than 3σ away from μ give little contribution. Figure 3.6 shows some examples of different values of σ.



(a) $\sigma = 4$ (b) $\sigma = 8$ (c) $\sigma = 16$ (d) $\sigma = 32$

Figure 3.6

3.2.2 Bilateral filter

The bilateral filter builds on the same principles as the Gaussian filter but extends it with an additional term. The bilateral filter includes spatial range and intensity range. The intensity range tries to preserve edges by using the intensities of all neighboring pixels. When two pixels are close neighbors, they do not necessarily have similar intensities. The bilateral BF is defined as follows:

$$BF(I)_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(|p - q|) G_{\sigma_r}(|I_p - I_q|) I_q, \tag{3.4}$$

where W_p is a normalization factor that ensures all weights sum up to one:

$$W_p = \sum_{q \in S} G_{\sigma_s}(|p-q|) G_{\sigma_r}(|I_p - I_q|). \tag{3.5}$$

G_{σ_s} determine the spatial weight of neighboring pixels and S are all pixels in the image. G_{σ_r} will determine the influence of neighboring pixels when comparing intensities of the pixel (range weight). When the neighboring pixel is very close but the intensity difference is high, this neighboring pixel will not contribute much to the filtered pixel. Until now the neighborhood of pixels were all the pixels in the image but Weijer and Boomgaard [15] and Weiss [13] observed that looking at a local neighborhood of pixels works well for bilateral filtering, since neighboring pixels more than 3σ away from the center pixel are neglect-able. Now the bilateral filter from equation 3.4 can be re-defined as:

$$BF(I)_p = \frac{1}{W_p} \sum_{q \in N_p} G_{\sigma_s}(|p-q|) G_{\sigma_r}(|I_p - I_q|) I_q, \tag{3.6}$$

where σ_s is usually set to $width(N)/4$, N_p is the neighborhood around pixel p and W_p is the normalization factor that ensures all weights in neighborhood N sum up to one:

$$W_p = \sum_{q \in N_p} G_{\sigma_s}(|p-q|) G_{\sigma_r}(|I_p - I_q|). \tag{3.7}$$

Figure 3.7 shows how the weights of neighboring pixels are produced and applied to an input to produce a smooth output. The resulting weight clearly shows an edge (cut of Gaussian function).

The range weight works well for some situations but consider a scenario where we have a sky-blue material against a blue sky. In this case the edge will not be detected since both intensities are almost identical. The cross bilateral filter extends the idea of using different inputs (Depth, normals, etc.) for edge detection to create an even better filter.

3.2.3 Cross Bilateral filter

In 2004 two separate articles introduced the joint Petschnigg et al. [17]/cross Eisemann and Durand [18] bilateral filter. Instead of using one input image, another image is used to represent the edges. This idea can be extended to use all kinds of additional images like depth and normals. For the duration of this thesis we will refer to these additional images as features or scene features F often referred to as edge-stopping functions. The cross bilateral filter (CBF) with normal and depth as additional features can be written as:

$$CBF(I)_p = \frac{1}{W_p} \sum_{q \in N_p} G_{\sigma_s}(|p-q|) G_{\sigma_r}(|I_p - I_q|) G_{\sigma_f}(|F_p - F_q|) I_q, \tag{3.8}$$

$$G_{\sigma_n}(|N_p - N_q|) G_{\sigma_d}(|D_p - D_q|). \tag{3.9}$$

The first and second term of equation 3.8 are the spatial and range weights respectively. The last term is equation 3.9. Each feature has its own standard deviation σ_n and σ_d. These standard deviations will allow more or less different neighboring features. Figure 3.8 show examples of normal and depth images used for edge detection.

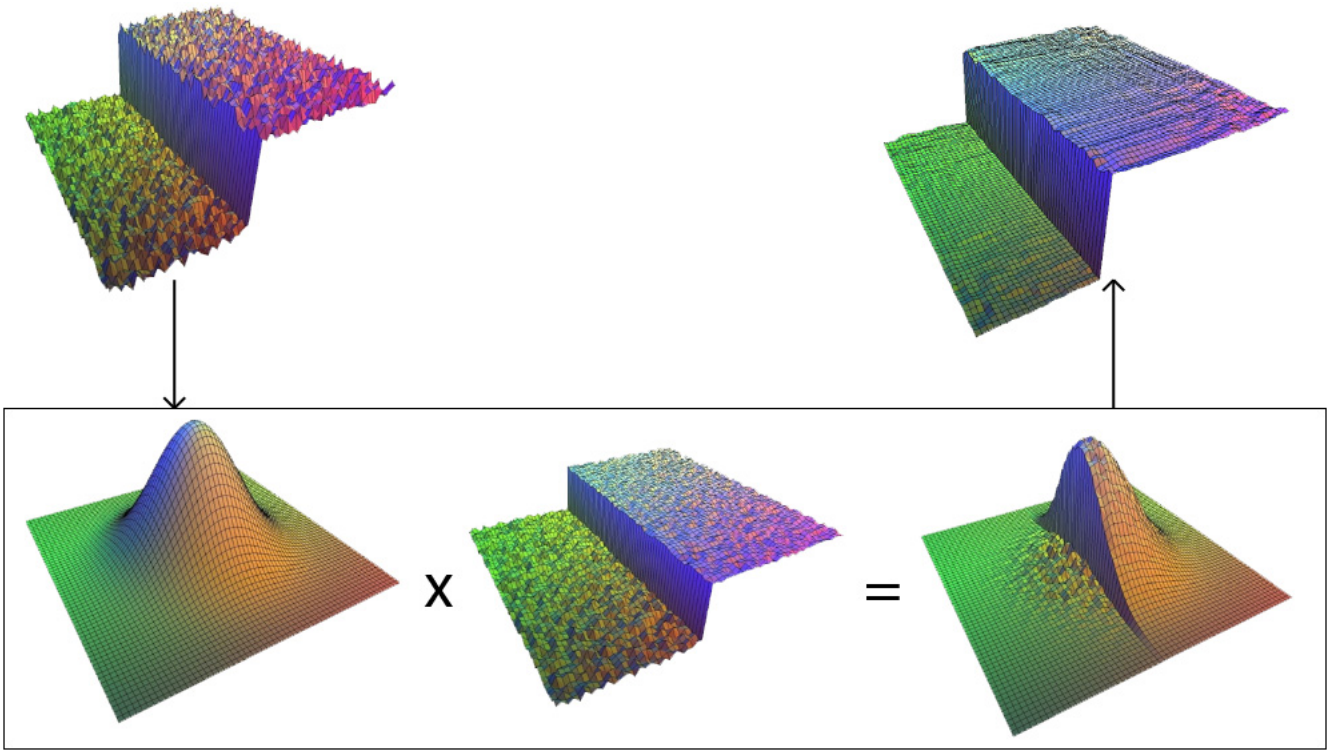


Figure 3.7: This figure is reproduced from [16]. The spatial weight is multiplied with the range weight to create the resulting output weights.

3.3 Cross bilateral based filters

In recent years much research is done on image denoising and image enhancement. The latter is usually focused on computer vision problems such as fog and haze removal. Image denoising is becoming an increasingly interesting topic for removing noise in path traced images. In chapter 2.4.1 the topic of noise in Monte Carlo path tracing was briefly discussed. The main focus lies on filtering low sample images that exhibit variance which makes the output unacceptably noisy. A path tracer can output features needed for a cross bilateral filter with ease. The normal and depth buffer can be easily obtained from the renderer (rasterisation or ray-tracing) by just storing them in separate buffers.

Filters that use features like depth and normals are called geometry aware filters. Holger Dammertz [10] uses a geometry aware Á-Trous (with holes) wavelet transform to filter noisy images produced by Monte Carlo path tracing, in real-time. The filter uses normals, world positions and direct illumination as features to detect edges in scene geometry as is done with cross bilateral filtering. The Á-Trous transform allows the filter to run in real-time with large kernels. Impressive results are achieved when dealing with simple materials, without camera effects like depth of field and motion blur. However, the filter fails in situations where there is very high geometry detail which causes the same problem as depicted in Figure 2.10. The algorithm also tends to over blur texture detail especially for non-diffuse materials. Despite the drawbacks, results achieved with only one sample per pixel are impressive which makes this filter useful for real-time path tracing. Figure 3.9 shows the pipeline of the method.

Karsten Schwenk [11] took a different approach by filtering only the high variance; blend this filtered image with the unfiltered high variance to eventually get the final result. By mixing

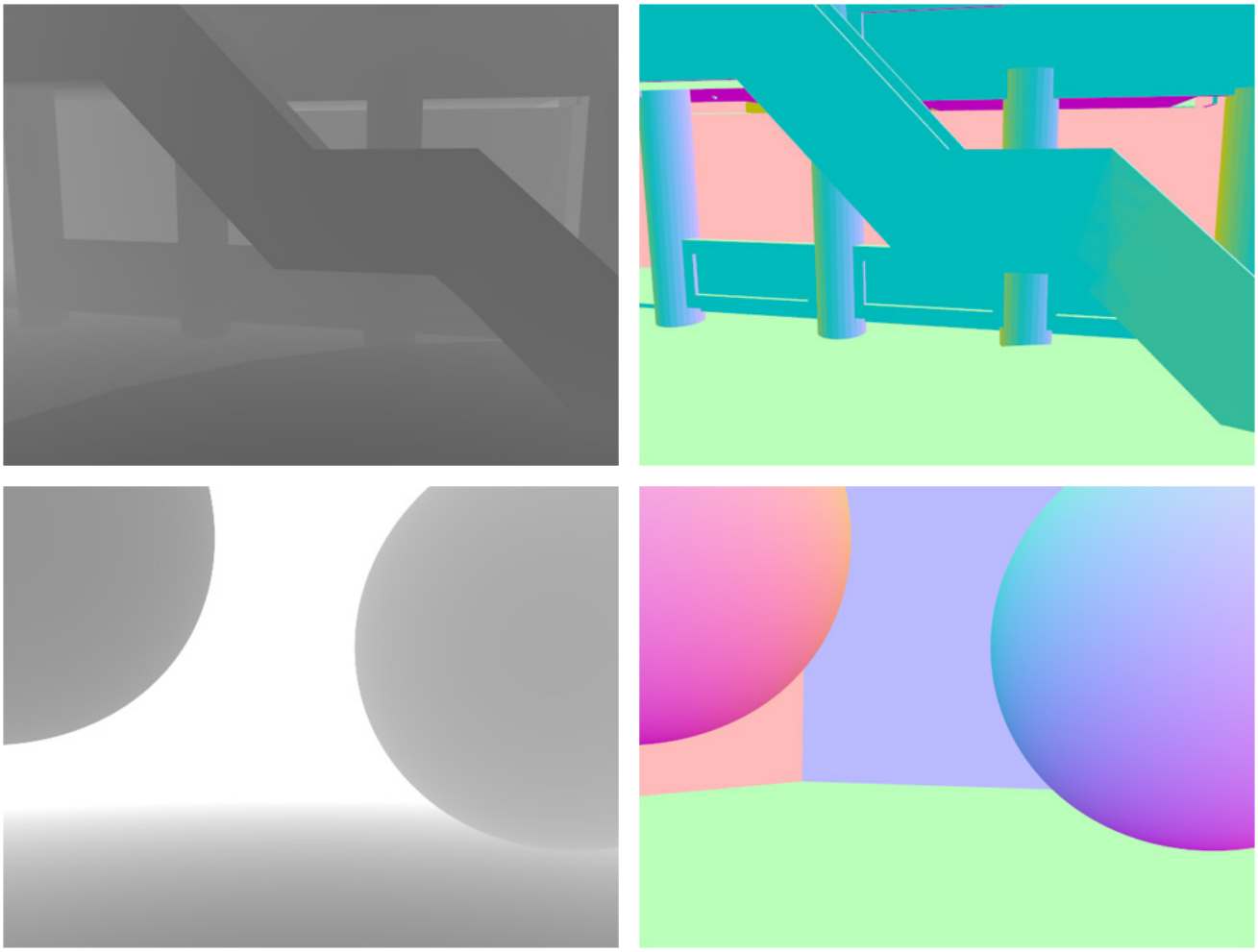


Figure 3.8: Left column are depths and right column are normals

the filtered with the unfiltered image with some blending operator (perceived variance), the output will not exhibit filtering artifacts while still looking noise free. The key idea of this paper is to split high variance from low variance to then mix the filtered high variance image with the unfiltered high variance image based on human perception of variance. When the unfiltered image starts to converge to a low variance result (based on a variance threshold), the filter will stop. For the filtering they use a cross bilateral filter with depth as additional scene features. As an optimization they only apply the filter every 2^i samples which means they can apply a large kernel (above 35 pixels wide) to get better filtering results. Just like the Á-Trous filter, this filter is designed to work with real-time path tracing. The filter handles specular reflection and transmission as well as texture detail very well. The main drawback is the user input that is needed to choose the perceived variance and high variance input. Monte Carlo effects like depth of field and motion blur are not discussed in the paper. Figure 3.10 shows the pipeline of the method.

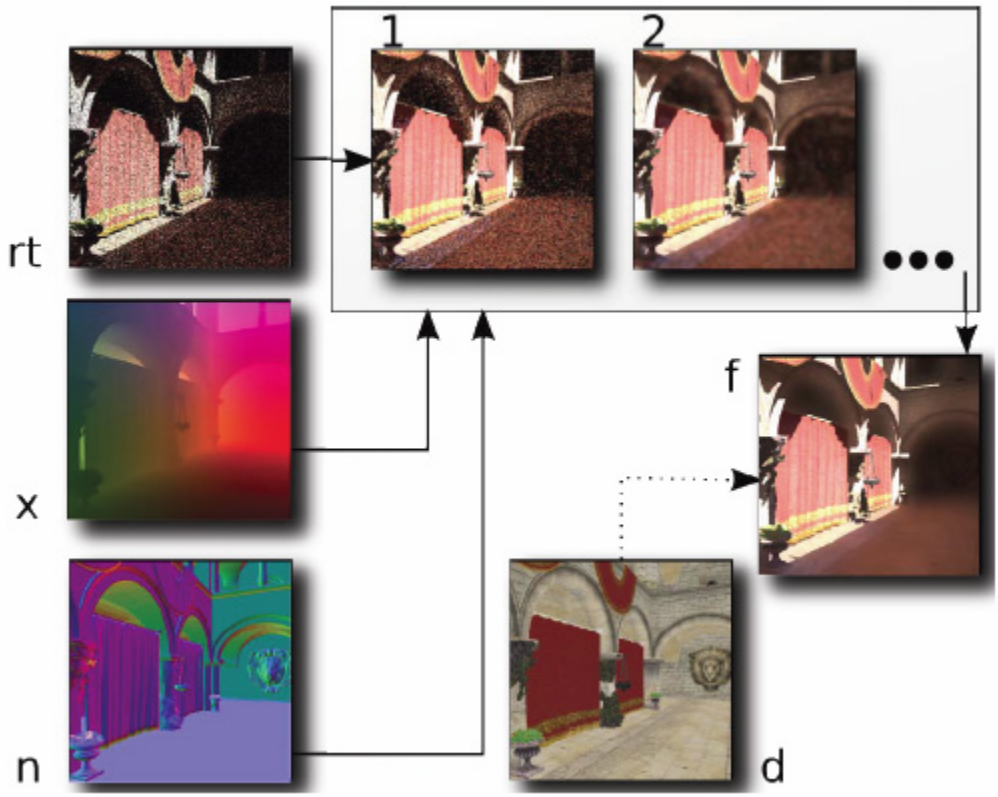


Figure 3.9

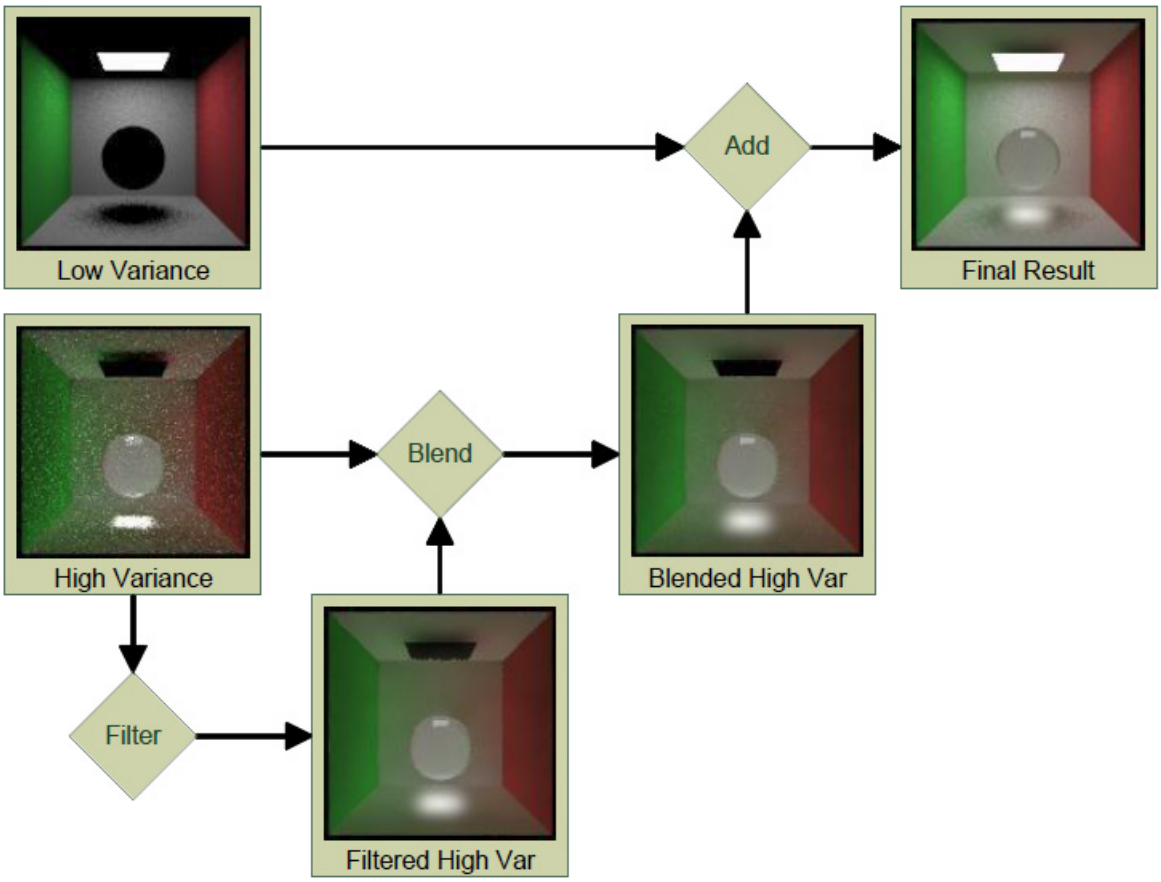

Low Variance
Add
Final Result
High Variance
Blend
Blended High Var
Filter
Filtered High Var

Figure 3.10

Part II

Implementation and Contribution

Chapter 4

OpenCL Path tracer

This chapter will discuss the implementation of an OpenCL path tracer developed to test new filtering algorithms. First the motivation and goals will be discussed. Secondly, the implementation details will be discussed which include code structure and details about algorithms used to implement the path tracer i.e. Bounding Volume Hierarchy (BVH) acceleration structure, intersection algorithm, etc.

4.1 Motivation and goals

The choice of OpenCL over CUDA is shortly discussed in 2.5 however, there are more reasons why OpenCL is preferred over CUDA. OpenCL is platform and Vendor independent meaning everyone with a modern device (pc, smart-phone, etc) can run OpenCL applications. Here is a list of advantages for OpenCL vs. CUDA:

1. Hardware independent i.e. CPU, GPU or dedicated devices
2. Vendor independent i.e. AMD, NVIDIA, Intel, etc.
3. Interoperability with OpenGL (Khronos group)

Here is a list of drawbacks for OpenCL vs. CUDA:

1. OpenCL runs slow on NVIDIA GPU's
2. OpenCL lags behind CUDA in terms of support and features

There are some OpenCL path tracers available such as "LuxRays" which is a part of "LuxRender" and "Laguna". Most path tracers are aimed towards production rendering. The most recent path tracer focused on games is "Brigade" from OTOY originally created by Jacco Bikker. This path tracer is designed to be fast meaning no tricks for faster convergence such as adaptive sampling are used. Currently they have an OpenCL version of the Brigade renderer which is not released to the public. It is more convenient to create a custom path tracer that can be used to develop/implement filtering algorithms more easily. The following goals were set for the path tracer:

- Render at a frame-rate of at least 20 frames per second
- Render complex geometry i.e. triangle meshes

- Texture support
- Diffuse, Glossy and Specular materials (BSDF's)
- Area light and environment mapping

4.2 Host implementation details

There are always two parts of an OpenCL application, namely, host and device(s). OpenCL setup is done on the host and OpenCL kernels are executed on the device(s). (CPU, GPU, etc.). The host handles most of the logic, whereas the device runs all the algorithms such as path tracing and filtering. Figure 4.1 shows a simplified diagram of the host side of the application.

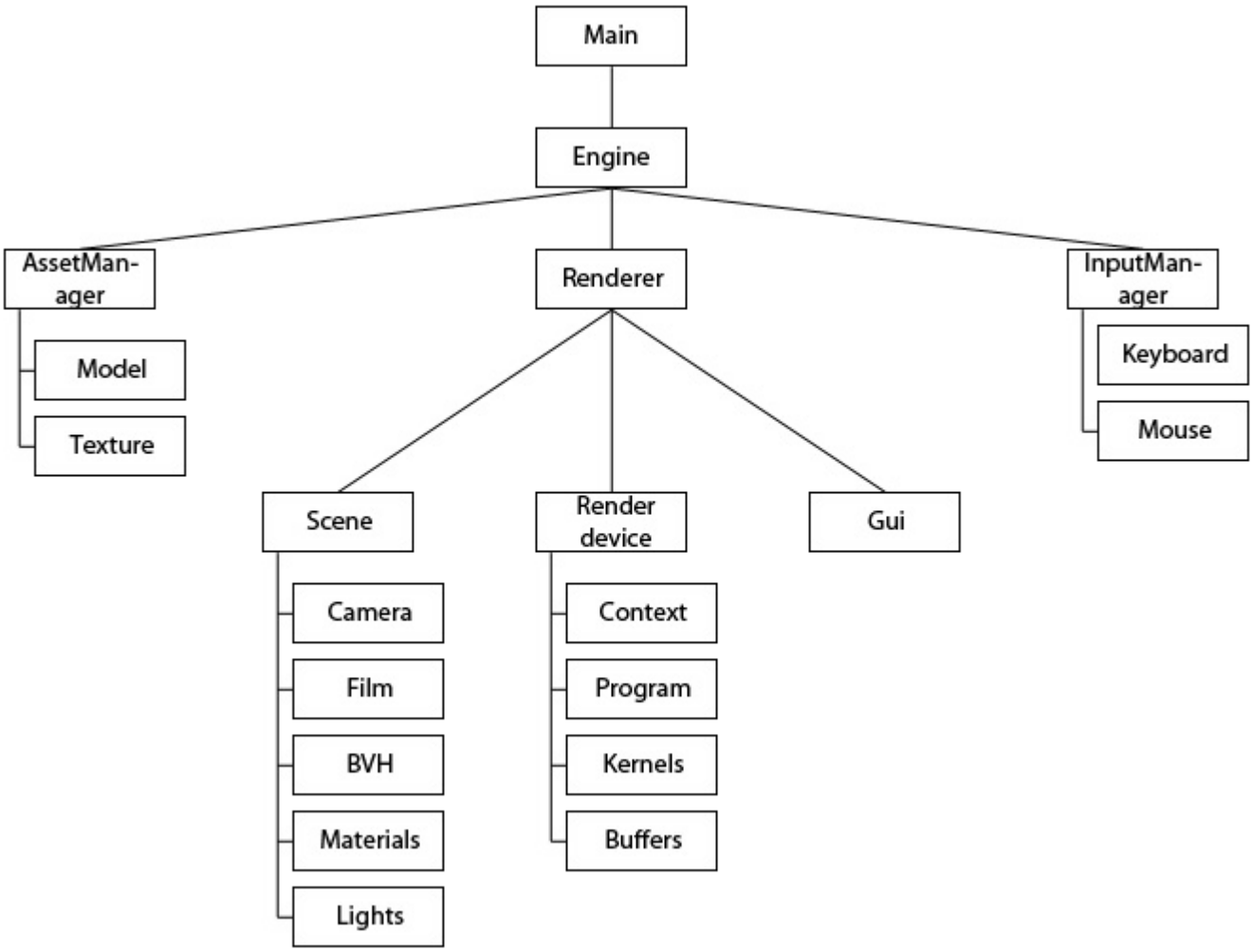


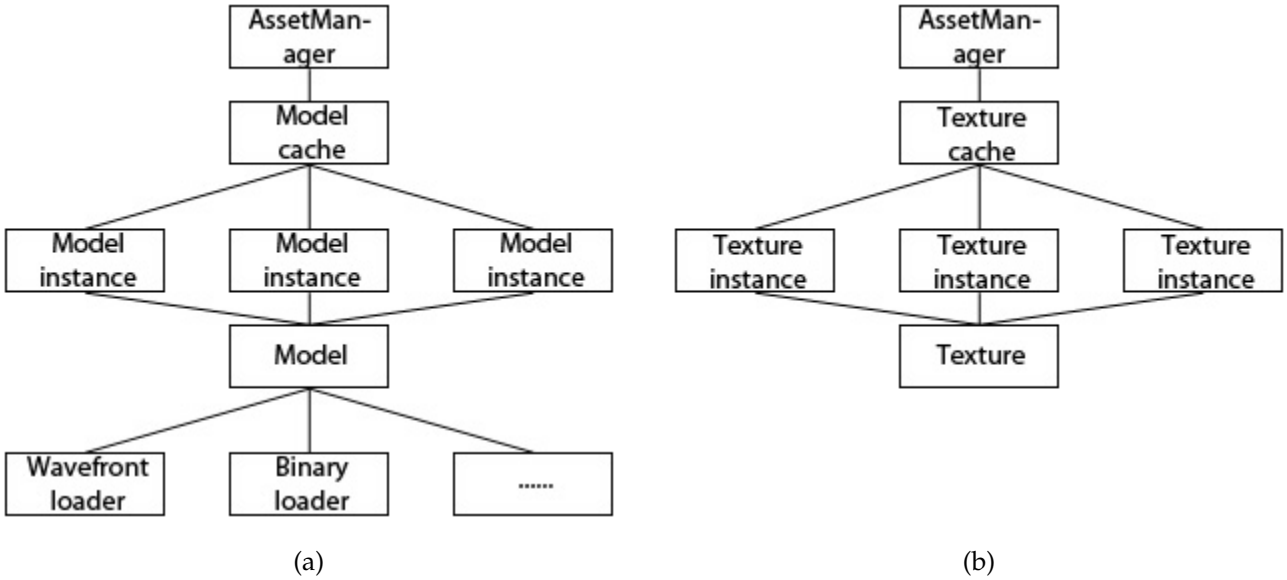
Figure 4.1: Simplified diagram of the OpenCL renderer

The renderer takes care of the scene, the render devices and the graphical user interface (GUI). The renderer starts of by initializing OpenGL and OpenCL. This means setting up the display (OpenGL) and the available render devices (OpenCL). If we have more than one render device, the renderer will create multiple render devices where each one gets its own thread

(multi-threading). Each render device is assigned to a specific part of the screen depending on the speed of the device. If we have two identical GPU's, each render device (one for each GPU) will get 50% of the pixels. Each important block from Figure 4.1 will be discussed.

4.2.1 Managers

The Asset and Input managers are abstractions to handle assets and controls. The asset manager is only a portal to load models and textures. Figure 4.2(a) shows how a model is loaded and Figure 4.2(b) shows how a texture is loaded. Only one instance of the AssetManager and InputManager can be created which is done by the Engine.



(a) (b)

Figure 4.2: AssetManager

When LoadModel is called on asset manager, it will ask the model cache for an instance. Instead of loading the same model multiple times, we only load it once. If another instance uses the same model, only a different transformation matrix is stored and a pointer to the already loaded model is used. If a model is not yet loaded, the model class will decide which loader to use by looking at the extension of the model filename e.g. Wavefront loader is for .obj files. The texture system works practically the same but instead of a transformation matrix, other parameters like scale and shift are used. When a texture needs to be loaded, we use the FreeImage library to load our textures from file.

The input manager uses The OpenGL Utility Toolkit (glut) to take care of the controls. Every object can subscribe to a keyboard or mouse listener, which gives each object the capability to implement its own controls. When a specific key or mouse event is triggered by glut, all objects subscribed to a listener will be notified. For keyboard or mouse polling we can use the AssetManager which updates every time a keyboard or mouse event is triggered. This abstraction makes it easy to implement controls where needed.

4.2.2 Scene

The renderer creates the scene which takes care of scene loading, camera's, acceleration structures and more. The scene initializes by loading a custom scene file that describes what the scene looks like i.e. Scene Description Language SDL. The following can be loaded with the scene file:

- General. Here we can specify some parameters that can be useful for experimentation. The 'Image writing' parameter can be either turned on or off and it is used to automatically save the screen to an image file.
- Camera. This states that a camera should be initialized. Each camera has an origin (x,y,z) and a looking direction (x,y,z). There are some additional parameters for depth of field: AutoFocus = true/false, FocalDistance (used when autofocus is disabled) and LensRadius (the larger the lens radius, the stronger the depth of field effect).
- Object. and object is defined by sphere or mesh. Sphere parameters are: radius, position and material id. The parameters for a mesh are: name (including file extension), texture id, material id, scale, position and type (static, dynamic, deformable).
- Texture. The texture parameters are name (including file extension), uv shift (shifted texture coordinates) and uv scale.
- Lights. There are two types of light that can be loaded. The first type is sphere light i.e. area light source, and takes radius, position and emittance as parameters. The next type is an infinite light which acts like a background. The parameters are: name of the texture (including file extension), gain (strength of light source), uv shift and uv scale (used for the texture)
- Material. A material describes a material with specific parameters. At the moment there are 4 types of materials. Diffuse, Perfect specular, Glass (refracting), Metal (Glossy). Diffuse and reflective materials take only color as a parameter. The glass material has a reflective color, refractive color, index of refraction outside i.e. air and index of refraction inside the object. The Metal material only has color and exponent (glossy strength) as parameters.

It is easy to add more types to the description file by adjusting the scene loading code. Using an SDL also provides fast control over the scene since the renderer does not have to be recompiled every time something is changed. In the following paragraphs the Camera, Film and BVH will be discussed.

Camera

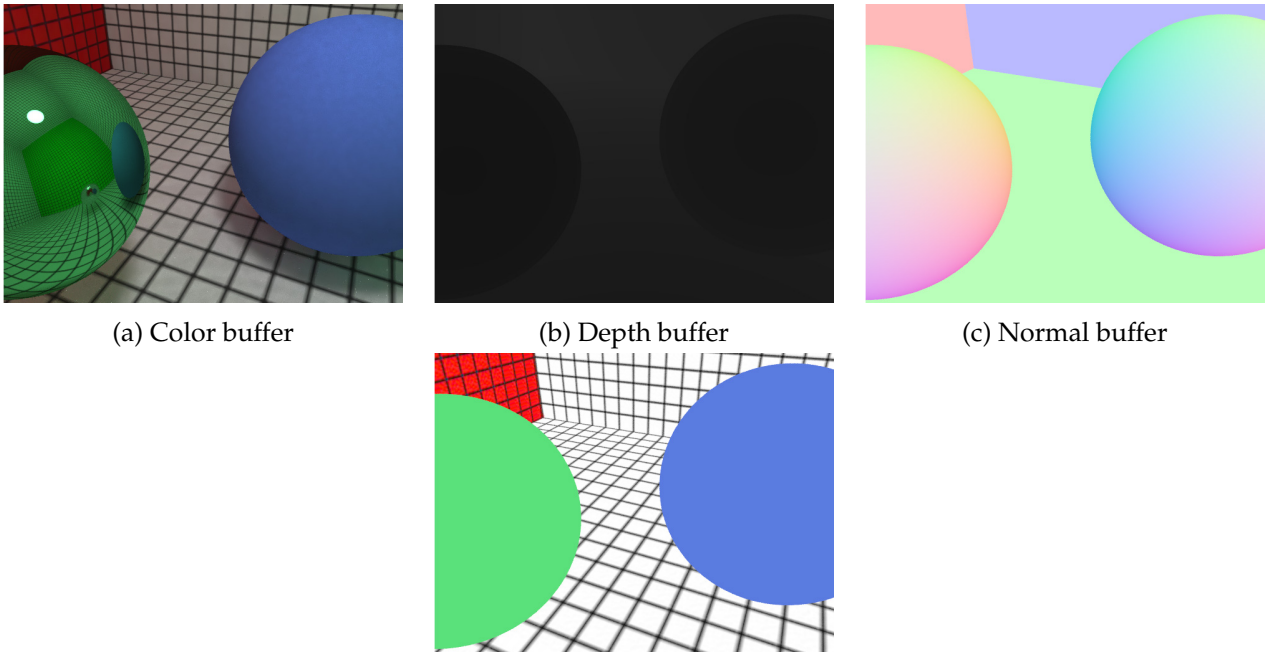
The Camera object stores the following data:

Parameter	Description
Origin	The current position in world space
Target	The current position in world space the camera is looking at. Looking direction = Target - Origin
Direction, Right and Up	Orientation of the camera
Lens radius	Used to set the depth of field strength. When 0, depth of field is turned off
Focal distance	Distance to the focal plane. When auto-focus is turned on, the focal distance is determined by the closest object towards the looking direction
Camera to World matrix	Used to transform from camera space to world space
Raster to Camera matrix	Used to transform from screen space to camera space

The camera updates its position, orientation and matrices every time it moves. All the camera controls are implemented inside the Camera class through a keyboard and mouse listener (as discussed in 4.2.1). Every time an update happens, the Film is reset and the new camera parameters are send to all devices.

Film

The Film stores the screen buffer (final output). Other buffers like normals, world positions and more are also stored inside the film. Figure 4.3 shows a visualizations of the buffers stored in the film. The screen width and height are also stored here and when they change all the buffers need to be re-created. The buffers are emptied when the camera updates.



(a) Color buffer (b) Depth buffer (c) Normal buffer

(d) Albedo buffer

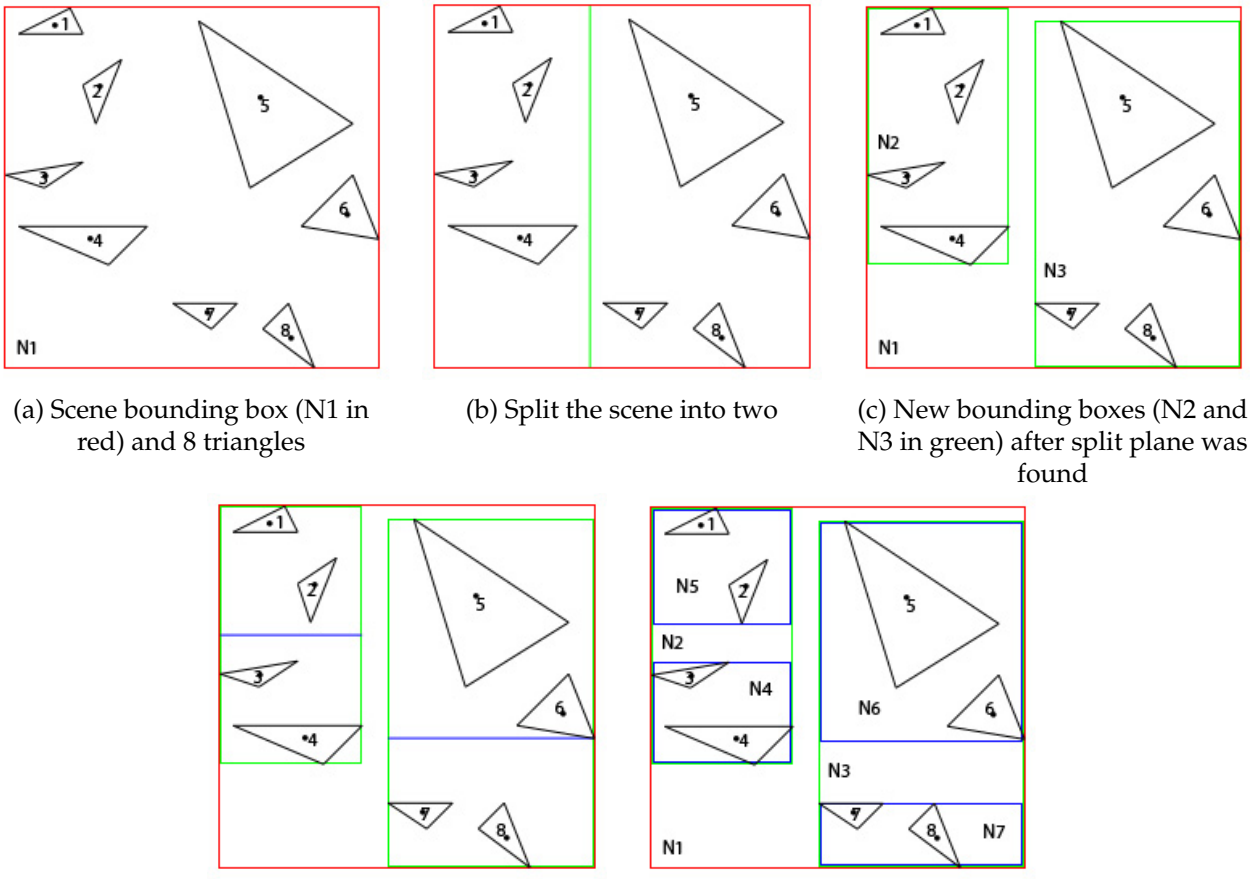
Figure 4.3: Film buffers

Whenever the screen resolution is changed all the buffers on the OpenCL devices need to be rebuild, so the Film notifies every device to do so.

BVH construction

The Bounding Volume Hierarchy is used to speed up ray-object intersections by arranging the triangles and objects in a tree like structure. The BVH acceleration structure has been the topic for many researchers since it is a great way to accelerate ray-object intersections. The first real GPU ray-traversal algorithm was introduced by Purcell et al. [19]. After the introduction of CUDA, more efficient GPU ray traversal algorithms started to emerge. In 2007 Gnther proposed an efficient ray packet traversal scheme using the BVH as data structure. Aila and Laine [4] studied BVH GPU algorithms and improved efficiency. He also noted that the traversal of camera rays, which are highly coherent i.e. share a common origin and direction, are likely to traverse the same nodes of the BVH which improves efficiency.

Compared to data structures like binary trees and quad-trees that subdivide space, the BVH groups objects with bounding volumes. The basic construction algorithm for the BVH starts by finding the scenes bounding box as depicted in Figure 4.4(a). Then a leaf node is created for every triangle in the scene. After this first step we start building the tree top-down within the scene bounds. In order to group objects together we first need to find split-planes that represent a division between objects. On both sides of the split-planes objects will be grouped.



(a) Scene bounding box (N1 in red) and 8 triangles

(b) Split the scene into two

(c) New bounding boxes (N2 and N3 in green) after split plane was found

(d) For each node find the best split plane

(e) New bounding boxes (N4-N7 in blue) after split plane was found

Figure 4.4: BVH initialization

There are several algorithms to find a split plane, but for this implementation the best aver-

age position between the triangle centers is the split-plane. In two dimensions this means the best split-plane lies on the x or y axis. Figure 4.4(b) shows that the first split-plane lies on the y-axis. Now the triangles can be grouped together on either side of the split plane as depicted in Figure 4.4(c). We keep going until we are left with two triangles in a single node in which case they are both put into a leaf node. The tree structure that results from this build is depicted in Figure 4.5. From this tree structure we build an array of nodes that we use for ray traversal. The array is constructed depth-first as depicted in Figure 4.6.

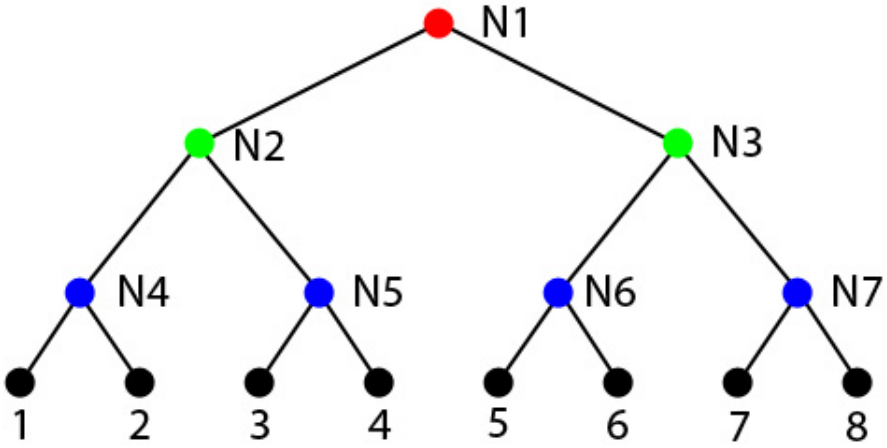


Figure 4.5: The resulting tree structure

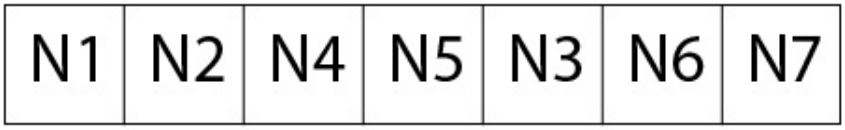


Figure 4.6: Array of nodes

Every node in the array stores the bounding box and it's the index of the triangle if it's a leaf (if it's a tree node then this index is set to -1). Additionally each node stores an index to its neighbor node which is used when a ray misses the node completely (more explanation on ray traversal later) e.g. N2 would have N3 as its skip index. If a ray misses N2, we do not have to traverse the its sub-tree, but instead skip to its neighbor N3 on continue traversal.

4.2.3 Render device(s)

The render device handles all communications with the OpenCL device and takes care of memory allocation and kernel execution. When the screen resizes the film tells the render device to reset all buffers and re-allocate them. Whenever a material changes, the material buffer is changed and uploaded to the OpenCL device. This counts for every buffer that changes during render time. The render device also takes care of source code loading which is needed to create the source code from several files, which is used by the program for compilation. We discussed some details about OpenCL in 2.5 but left out some details which will be discussed here.

Context

A Context defines the entire OpenCL environment for one or more devices i.e. CPU, GPU, etc. This includes kernels, memory management, command-queues and more. The Context is initialized by the render device. This context is used to create our Program, Kernel(s), Buffer(s) and Command-Queue(s).

Program

A Program consists of a set of kernels, functions and other code. When creating a program the files that define the program must be specified. The program is compiled at run-time.

Kernels

A kernel is a function that runs on an OpenCL device and is essentially where all the OpenCL magic happens. A kernel is executed in a wavefront which is usually around 32 to 64 threads (work-items) wide. All threads in this wavefront execute the same code. If one work-item diverges from the rest of the wavefront (if-else), the other work-items will have to wait. This causes the program to slow down which is why it is important to watch out for too much diverging code.

Buffers

Buffers are created on the host and represent allocated memory on the OpenCL device. A Buffer can be Read/Write or both at the same time. Data can be written to these buffers so they can be used inside a kernel. Each buffer is linked to a context, since the context handles device memory management.

Command queues

Command queues are used to send commands to the OpenCL device. If we want to execute a kernel or write a buffer on the OpenCL device, we add the execute and write commands to the queue. With the use of events we are able to track the process of each command.

4.3 OpenCL device implementation details

The implementation of this path tracer is often referred to as a mega kernel due to the fact that all code is executed in one kernel. No path regeneration or multiple importance sampling is done to keep the code complexity down. The path tracing kernel processes one pixel sample each time it is executed. Each work-item processes one pixel at a time i.e. the total number of tasks is resolution dependent. When the kernel is executed the global id of the work-item determines which pixel it works on. After determining the pixel, a camera ray is generated which is then used to traverse the scene (path tracing). After the ray has terminated, the accumulated radiance is deposited to the screen. The traversal of the camera ray is a large loop that keeps bouncing the ray around in the scene until it terminates. There are three ways a ray can terminate:

- Maximum depth / Russian roulette

- Nothing is hit
- Light source is hit

Figure 4.7 shows an overview of the implementation.

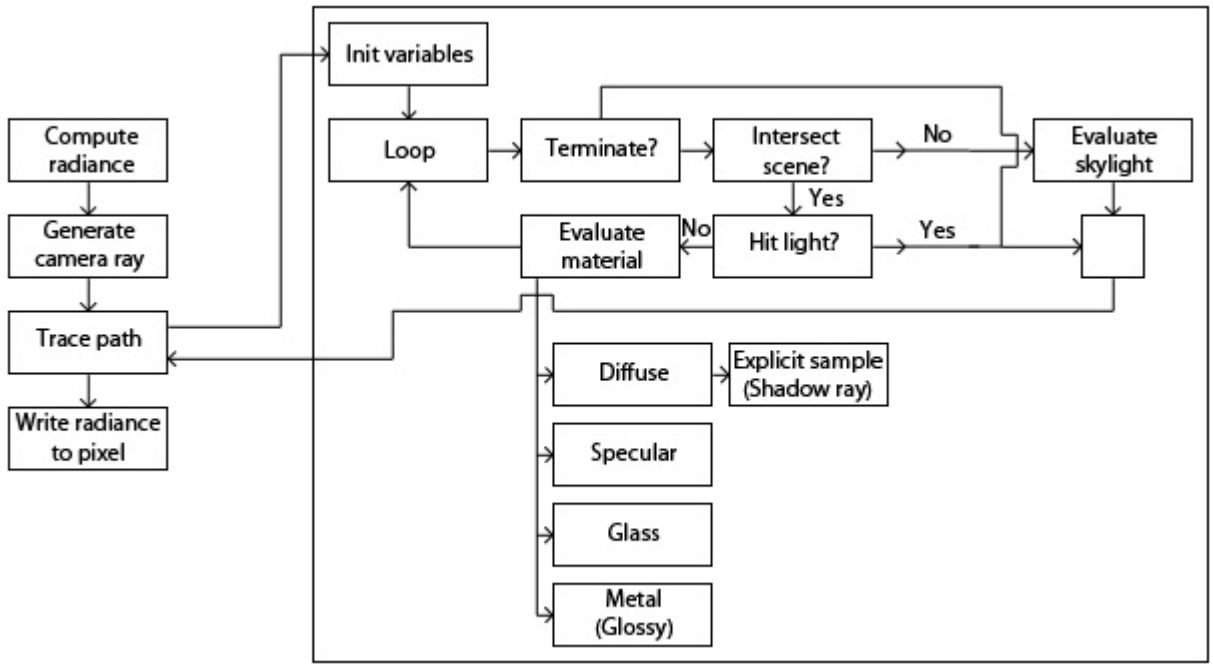


Figure 4.7: Simplified diagram of the path tracing kernel

In this section a description of camera ray generation, intersection and material evaluation will be given. Material evaluation will include explanations of diffuse reflection, specular reflection (mirror) and glass (refractive/transmissive).

4.3.1 Camera ray

We first pick a random position inside the pixel to shoot the ray through (Figure 2.7a). In order to generate a ray we need to transform it from pixel coordinates to camera coordinates and then transform it from camera to world coordinates. To do this we use the matrices discussed in 4.2.2. After these transformations a ray is created with a new origin and direction. Depth of field affects the origin of the ray and causes the ray to go in an offset direction. The camera model is depicted in Figure 4.8.

Every object that lies on the focal plane at distance f, will be in focus. When the radius of the lens is 0, everything will be in focus and when it becomes larger, objects outside the focal plane will be out of focus. A position on the lens needs to be sampled in order to get a proper offset from the origin. The most simplistic lens is circular. Sampling of a circular lens is shown in Figure 2.7b, where the radius of the lens determines how much the offset will be. Figure 4.9 shows some examples of the depth of field effect.

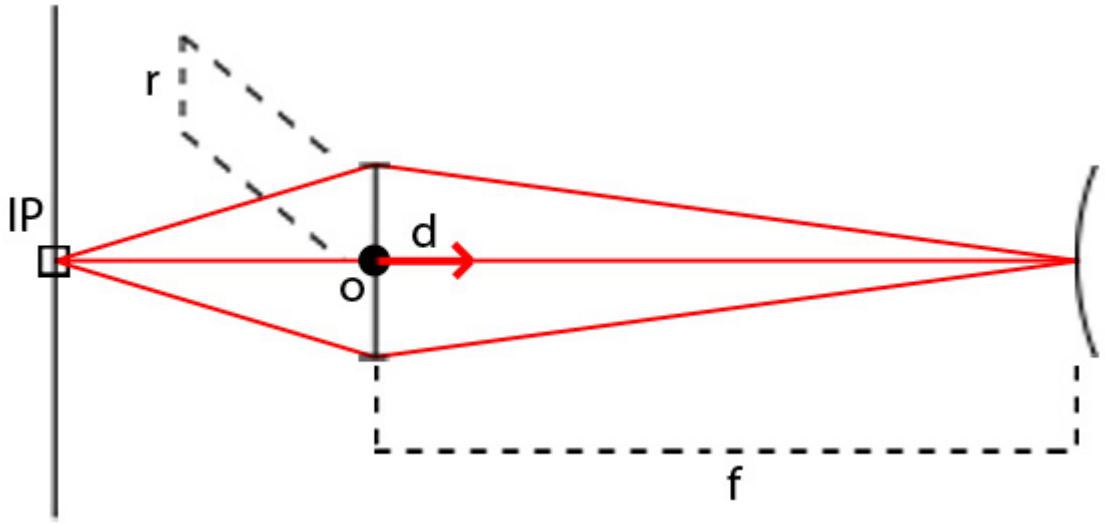
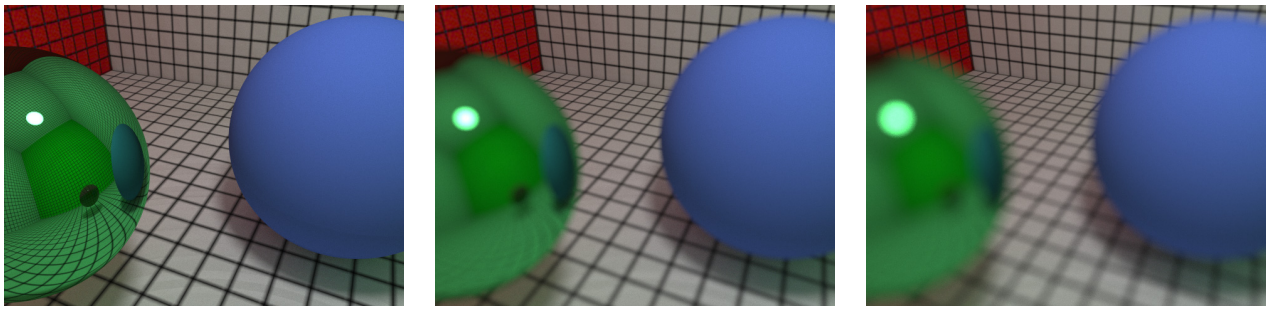


Figure 4.8: Where IP is the image pixel, o is the ray origin, d is the ray direction, r is the lens radius and f is the focal distance



(a) Lens radius 0

(b) Lens radius 1 and focal distance 150

(c) Lens radius 2 and focal distance 150

Figure 4.9: Depth of field

4.3.2 Intersection

Previously we discussed the BVH acceleration structure which is used to speed up ray-object intersections. In this paragraph we discuss how these intersections are performed. The goal of the intersection check is to find the closest hit-point of the ray which is then used to shade that point and determine a new direction depending on the material we hit (BSDF's). The BVH is built with axis aligned bounding boxes (AABB), which makes it important to do fast ray-box intersections. There are some primitives the ray can intersect like triangles, spheres, cylinders and more. We will only discuss ray-AABB and ray-triangle intersections. The equation of a ray is:

$$R = O + tD, \tag{4.1}$$

where R is the ray, O is the rays origin, D is the normalized ray direction and t is the defines the length of a ray or distance to an intersection point.

Ray-box intersection

Ray bounding box intersections have been studied by several authors. The implementation of Ray-AABB intersection is based on the work of Williams et al. [20]. An AABB is defined by six axis aligned lines that are referred to as the minimum bound and maximum bound of the bounding box (3 lines each). The only place where we use this intersection test is for BVH node intersection. Therefore we only need to know if the ray hit the AABB which makes the algorithm less complex. Algorithm 2 shows the Ray-AABB intersection.

Algorithm 2 Ray-AABB intersection

function BBOXINTERSECT(O, D, $nearPlane$, $farPlane$, $pMin$, $pMax$)
    $rayInverse \leftarrow 1/D$
    $l1 \leftarrow (pMin - O) \times rayInverse$
    $l2 \leftarrow (pMax - O) \times rayInverse$
    $tNear \leftarrow min(l1, l2)$
    $tFar \leftarrow max(l1, l2)$
    $t0 \leftarrow max(max(tNear, tNear), nearPlane)$ ▷ Nearest intersection point on AABB
    $t1 \leftarrow min(min(tFar, tFar), farPlane)$ ▷ Farthest intersection point on AABB
    return $t1 > t0$
end function

Ray-triangle intersection

Since most modern 3d applications use mainly triangles to construct the virtual world, an efficient triangle representation is needed. In computer graphics the most common way to represent a triangle are barycentric coordinates. The idea is to use the three vertices of a triangle to define every point that lies on the triangle. This is done by linearly interpolation these coordinates to get any point on the triangle. Any point P on the triangle can be found by:

$$P = wP_1 + uP_2 + vP_3, \tag{4.2}$$

where P_1, P_2, P_3 are the vertices of the triangle and u, v, w are weights for each vertex. These weights are subject to constraints:

$$u + v + w = 1. \tag{4.3}$$

We do not need three weights since we can compute any one weight using the others so we can rewrite equation 4.2 as:

$$P = (1 - u - v)P_1 + uP_2 + vP_3, \tag{4.4}$$

Figure 4.10 shows a point on a triangle defined by three vertices and three barycentric coordinates. Möller and Trumbore [21] introduced an efficient way to do ray-triangle intersection using barycentric coordinates. We want to solve equation 4.6 to find unknowns t, u and v. Algorithm 3 is the pseudo code for ray-triangle intersection.

$$O + tD = (1 - u - v)P_1 + uP_2 + vP_3, \tag{4.5}$$

$$O - P_1 = -tD + uP_2 + vP_3. \tag{4.6}$$

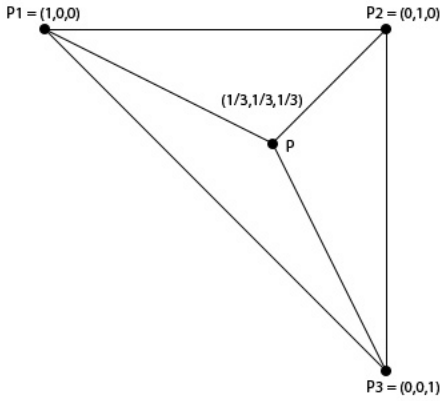


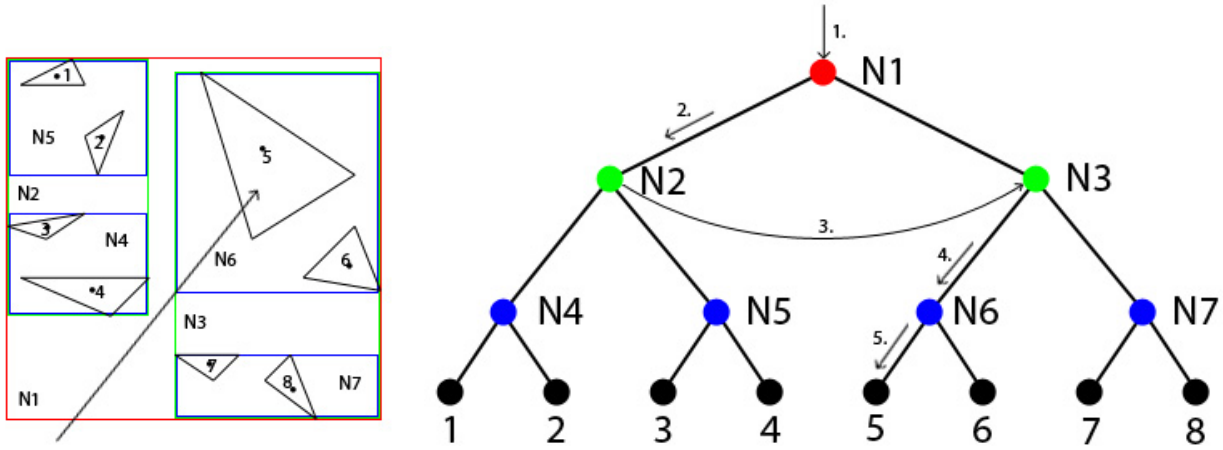
Figure 4.10: Triangle (P_1, P_2, P_3) and point P with barycentric coordinates $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$

Algorithm 3 Ray-Triangle intersection

function TRIANGLEINTERSECT(O, D, P_1, P_2, P_3)
  $E1 \leftarrow P_2 - P_1$ ▷ Triangle edge 1
  $E2 \leftarrow P_3 - P_1$ ▷ Triangle edge 2
  $Cross \leftarrow CrossProduct(D, E2)$
  $Determinant \leftarrow DotProduct(E1, Cross)$
  if $Determinant == 0$ **then**
    return 0 ▷ No intersection
  end if
  $Vec1 \leftarrow O - P_1$
  $u = DotProduct(Vec1, Cross)/Determinant$
  if $u < 0 \vee u > 1$ **then**
    return 0 ▷ No intersection
  end if
  $Vec2 \leftarrow CrossProduct(Vec1, E1)$
  $v = DotProduct(D, Vec2)/Determinant$
  if $v < 0 \vee (u + v) > 1$ **then**
    return 0 ▷ No intersection
  end if
  $t = Dotproduct(E2, Vec2)/Determinant$ ▷ Intersection found
  return t, u, v
end function

BVH intersection

When an intersection is done using the BVH, the ray needs to traverse the tree until it finds the closest triangle it intersects. The traversal is done depth first. Figure 4.11(a) shows an example of a ray intersecting the scene and Figure 4.11(b) shows how the tree is traversed.



(a) A ray intersecting the scene

(b) Traversing the BVH tree to find the closest triangle

Figure 4.11: BVH intersection

The skip index that is stored in each node is used in step 3 to skip N2's sub-tree and go straight to N3. The traversal algorithm involves a ray-AABB intersection for each tree node and a ray-triangle intersection for each leaf node (if this leaf was hit). The pseudo code for BVH intersection is described in algorithm 4

Algorithm 4 Ray-BVH intersection

function BVHINTERSECT(O, D, $triangles$, $BVHTree$, $nearPlane$, $farPlane$)
  $hitIndex \leftarrow -1$
  $hitDistance, outU, outV \leftarrow 0$
  $currentNode \leftarrow 0$
  $stopNode \leftarrow BVHTree[0].skipIndex$ ▷ Skip index of the root
  while $currentNode < stopNode$ **do**
    $node \leftarrow BVHTree[currentNode]$ ▷ Retrieve the current tree node
    if $BBoxIntersect(O, D, nearPlane, farPlane, node.pMin, node.pMax$ **then**
      if $node.index \neq -1$ **then**
        $tri \leftarrow triangles[index]$
        $distance, u, v \leftarrow TriangleIntersect(O, D, tri.p1, tri.p2, tri.p3$
        if $distance > 0$ **then**
          $hitIndex \leftarrow index$
          $hitDistance \leftarrow distance$
          $outU \leftarrow u$
          $outV \leftarrow v$
        end if
      end if
      $currentNode \leftarrow currentNode + 1$
    else
      $currentNode \leftarrow node.skipIndex$
    end if
  end while
  return $hitIndex, hitDistance, outU, outV$
end function

4.3.3 Material evaluation

The path tracer contains some physically accurate materials. A material is made out of specific components like the reflection/refraction function (BSDF) and albedo. Glass is a material type that is commonly referred to as a dielectric. Dielectrics are poor in conducting electricity and include glass, air, vacuum and ceramic.

Diffuse

The diffuse material models a Lambertian surface where the light comes in at angle θ_i and leaves at a totally random angle around the hemisphere θ_o i.e. isotropic reflection. Lambertian reflection is named after "Johann Heinrich Lambert" which introduced perfect isotropic reflection in 1760.This is depicted in Figure 2.8(a). To calculate a new direction for an incoming ray we apply the following algorithm:

Algorithm 5 Perfect diffuse reflection

function DIFFUSEREFLECT(r_1, r_2, normal)
  $azimuthal \leftarrow 2\pi \times r_1$
  $deflection \leftarrow r_2$
  $u \leftarrow (0, 0, 0)$
  $w \leftarrow normal$
  $temp \leftarrow (0, 0, 0)$
  if $|w.x| > 0.1$ **then**
    $temp \leftarrow (0, 1, 0)$
    $u \leftarrow Crossproduct(temp, w)$
  else
    $temp \leftarrow (1, 0, 0)$
    $u \leftarrow Crossproduct(temp, w)$
  end if
  $u \leftarrow normalize(u)$
  $v = CrossProduct(w, u)$
  $D \leftarrow u(cos(azimuthal) \times \sqrt{deflection}) + v(sin(azimuthal) \times \sqrt{deflection}) + w\sqrt{1 - deflection}$ ▷ Generated direction with spherical coordinates
  return D
end function

Perfect specular

An object close to a perfect specular material would be a mirror. The incoming light direction angle θ_i is the same as the outgoing light direction angle θ_o. This is depicted in Figure 2.8(c). The following formula is applied to find the new ray direction after hitting a perfect specular object:

$$D = N - (2\theta_i N), \tag{4.7}$$

where D is the outgoing direction, N is the surface normal and θ_i is the ray direction.

Glass

Glass is both reflective and refractive at the same time. The probability that a ray will reflect depends on the reflection coefficient which is evaluated before refracting the ray. This probability is compared to a random number which will decide if the ray will reflect or transmit. There is an approximation to calculate this called "Schlick's approximation" Schlick [22]. While calculating the reflection coefficient we also take care of the critical angle which will be explained next.

In the simple case of glass there are two refractive indices where the first is the index for the incident medium η_i and the second index is for the medium we transmit to η_t e.g. if we travel from air to glass $\eta_i = 1.0$ and $\eta_t = 1.33$. Snell's law states the following:

$$\eta_i \sin(\theta_i) = \eta_t \sin(\theta_t), \tag{4.8}$$

where θ_i and θ_t are the incident and transmitted angles respectively. We calculate the transmitted angle i.e. new direction of the ray, by re-arranging equation 4.8.

$$\sin(\theta_t) = \frac{\eta_i}{\eta_t} \sin(\theta_i) \tag{4.9}$$

When we travel from glass to air there is an angle called the critical angle. When this critical angle is reached the ray will reflect back inside which is called total internal reflection and can be calculated by re-writing equation 4.8.

$$\theta_c = \sin^{-1}(\eta_t/\eta_i), \tag{4.10}$$

where θ_c is the critical angle. In order to find the critical angle we filled in $\theta_t = 90$, which enables us to drop this term from the equation since $\sin(90) = 1$. Figure 4.12 shows the effect of the reflection coefficient and Figure 4.13 shows diffuse, mirror and glass materials.

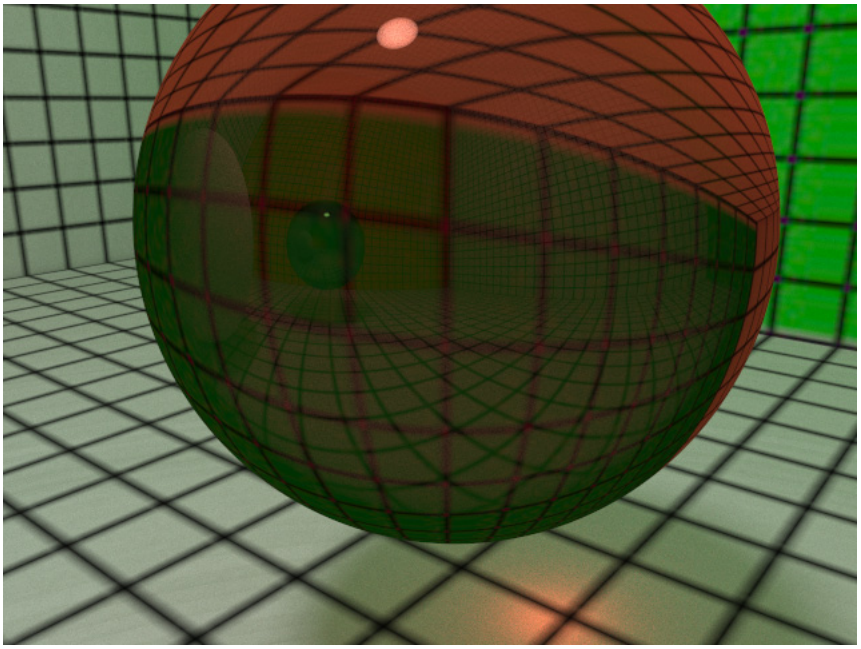
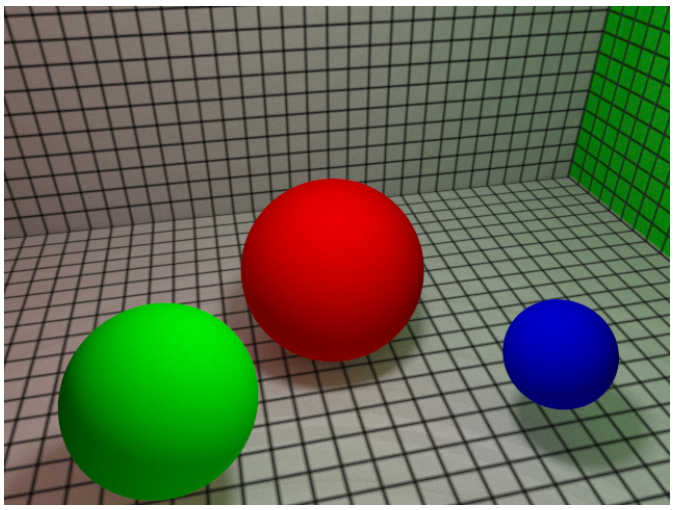
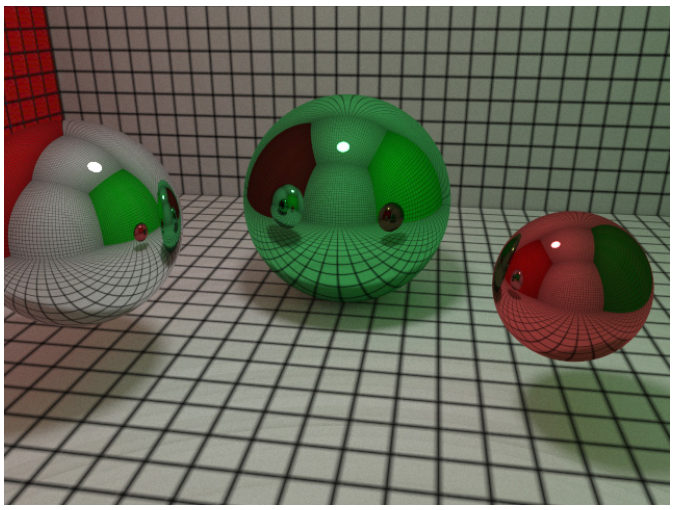


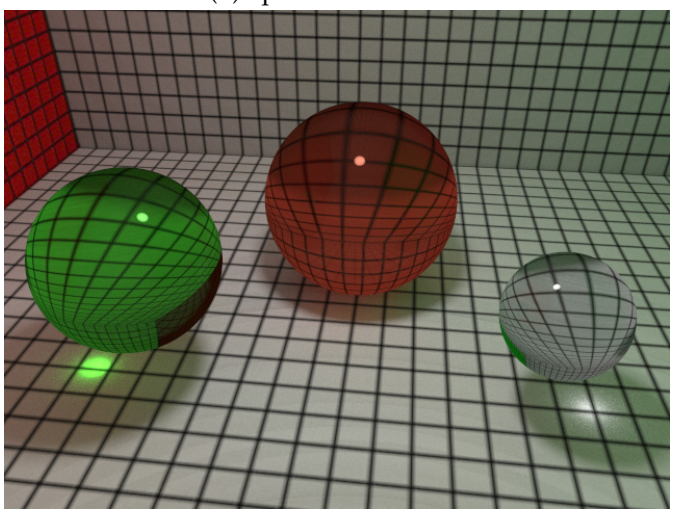
Figure 4.12: Notice how the glass reflects and transmits light



(a) Diffuse materials



(b) Specular materials



(c) Glass materials

Figure 4.13

4.4 Conclusion

The OpenCL path tracer that was created is a good platform to implement and test the upcoming filtering algorithms. There is great control over the renderer in terms of expandability. Although it is not the fastest GPU path tracer in existence, it is able to reach frame rates of 20 frames per seconds or more, depending on the scene complexity. Every goal that was set for this renderer is achieved.

Chapter 5

Random Parameter Filtering

In Chapter 3 several techniques have been discussed that try to solve the noise problem that comes with path tracing. Most existing techniques look inside a neighborhood around the pixel to estimate a correct output color. In 2011 Sen and Darabi [12] proposed a method that uses random parameters to filter the noise that plagues path tracing. They call this method Random Parameter Filtering (RPF). RPF uses random parameters that include lens coordinates u and v and t (time) for motion blur effects, but is generalized to use N random parameters. The idea of this method is to find the statistical dependencies between the random parameters and the scene features. Figure 5.1 shows how the dependency between the inputs x, y, u and v are evaluated against the outputs (scene features). The screen coordinates are also random parameters but are handled separately. Mutual information is used to determine the statistical dependency's and a cross bilateral filter is used during the filtering step of the algorithm.

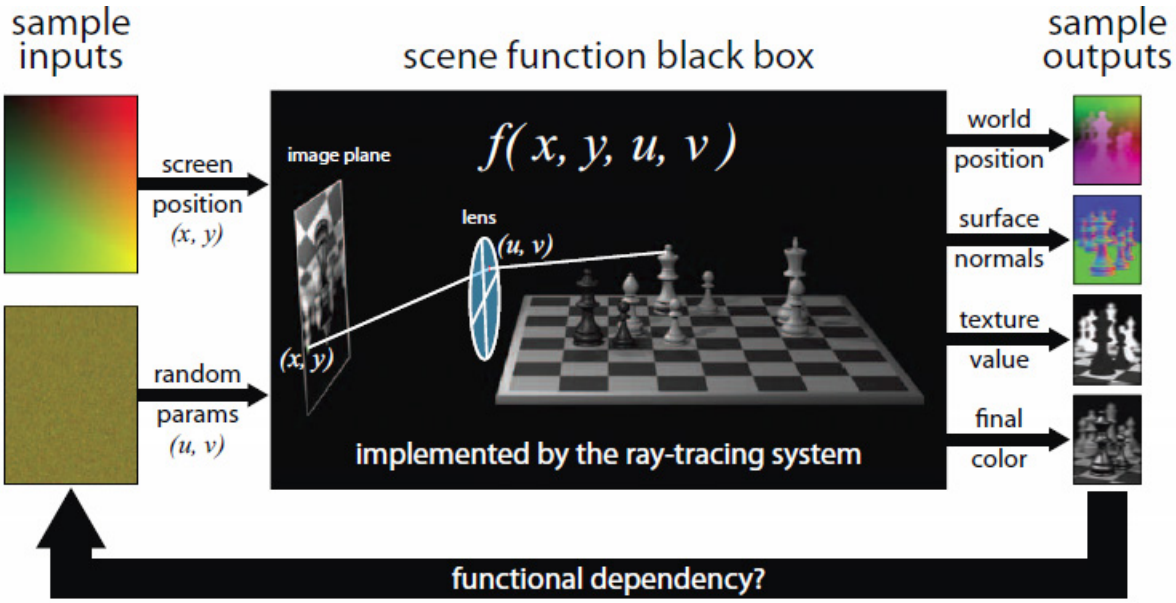


Figure 5.1

These statistical dependencies are converted to two parameters α and β that both act as additional weights for the range and scene features respectively. Their formulation of the cross-

bilateral filter weight which is similar to equation 3.9 looks like:

$$\begin{aligned} w_{ij} &= \exp[-\frac{1}{2\sigma_p^2}\sum_{1\leq k\leq 2}(\overline{p}_{i,k}-\overline{p}_{j,k})^2]\times \\ &\quad \exp[-\frac{1}{2\sigma_c^2}\sum_{1\leq k\leq 3}\alpha_k(\overline{c}_{i,k}-\overline{c}_{j,k})^2]\times \\ &\quad \exp[-\frac{1}{2\sigma_f^2}\sum_{1\leq k\leq m}\beta_k(\overline{f}_{i,k}-\overline{f}_{j,k})^2], \end{aligned} \tag{5.1}$$

where $w_{i,j}$ is the weight given to a neighbor and the first term is the spatial weight, the second term is the range weight and the last term is the geometric/ scene feature term. $2\sigma_c^2$ and $2\sigma_f^2$ will be discussed in 5.1.4. The addition to this filter over a traditional cross-bilateral filter is α_k and β_k, where k is the k-th weight for the k-th color channel and k-th scene feature respectively. The distances for each term are calculated using normalized screen positions ($\overline{p}$), sample colors ($\overline{c}$) and scene features ($\overline{f}$). This normalization process will be explained in section 5.1.2.

In section 5.1 the theory and implementation of the RPF algorithm will be discussed. Section 5.2 will discuss experimentation done with RPF and shows results. At the end of this chapter a conclusion based on the experimentation is made, where the drawbacks of RPF will also be discussed. Based on these drawbacks a new filter is developed in chapter 6 to try to overcome these drawbacks.

5.1 Theory and Implementation

The RPF algorithm is split up into four parts. First, samples have to be generated by the rendering system to provide as input for the RPF algorithm. For good results around 8 samples per pixel (spp) are needed (5.1.1). In the following step all the samples are pre-processed to keep the computational time of the third step at a minimum (5.1.2). In step three weights are generated by calculating the statistical dependencies between the random parameters and the rendering output (5.1.3). These weights are then used to guide the filtering step which is done in step 4 (5.1.4). Figure 5.2 shows the pipe-line of the RPF algorithm.

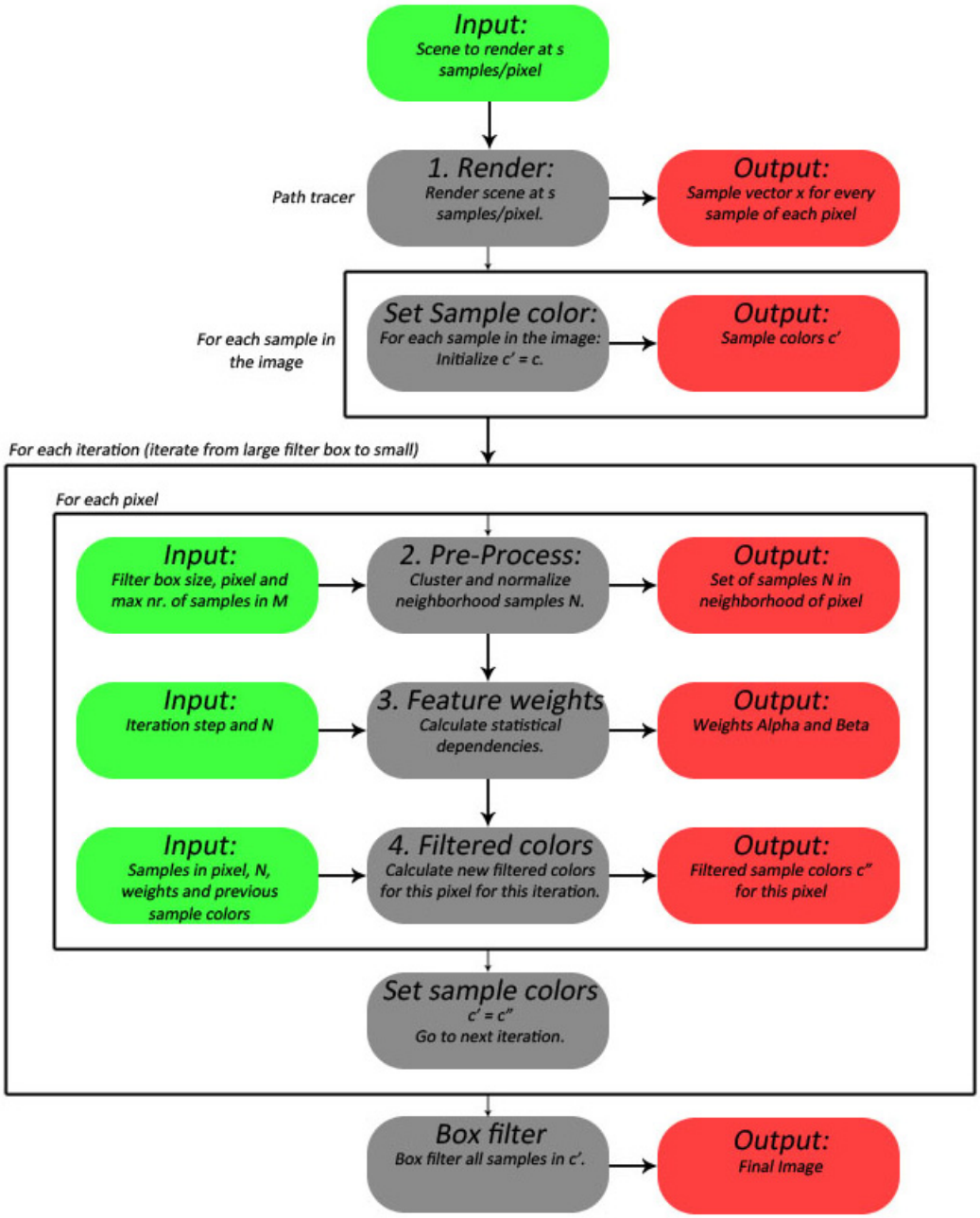


Figure 5.2

5.1.1 Sample-vector generation

The rendering system outputs a vector for each sample inside a pixel, which is depicted in Figure 5.3. The memory we need to store each sample needs to be allocated on the OpenCL device as well as the host device. For a large resolution this can become problematic if the rendering system is running on a GPU OpenCL device. Not only does this take up memory, but also bandwidth to stream the data back and forth to / from the device. Since RPF is not an algorithm designed for interactive applications, these two problems are no issue since we can compute and allocate memory in chunks or throw sufficient resources at it.

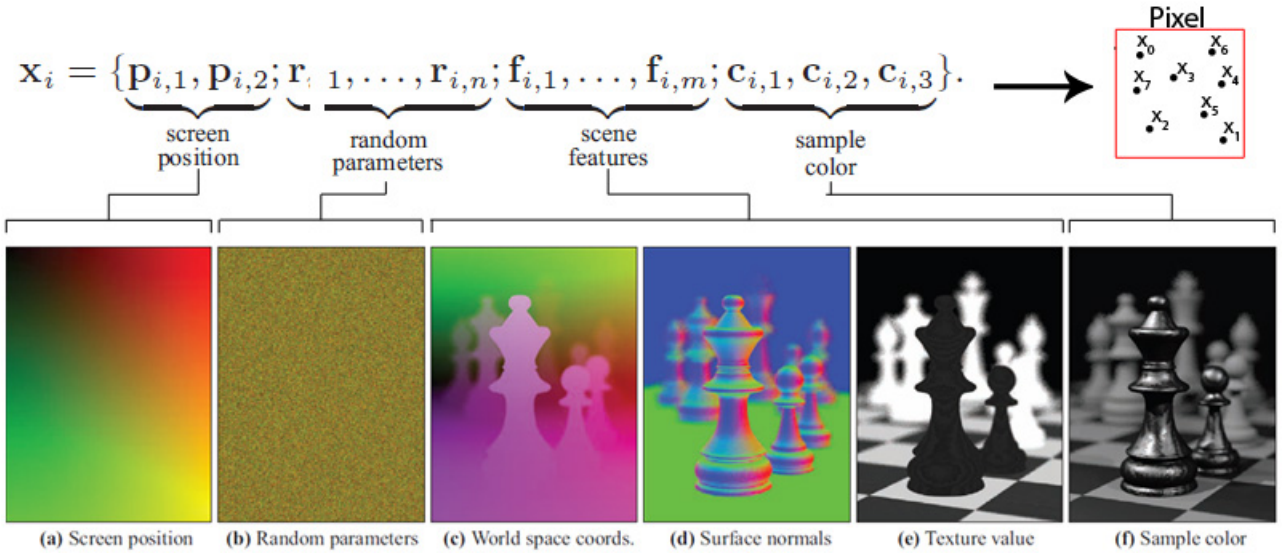


Figure 5.3: x_i is the $i'th$ sample-vector of the pixel. In this example 8 sample-vectors per pixel are stored

In our case the rendering system is the one we introduced in chapter 4. The scene features and random parameters that need to be stored are:

- World coordinates of the first and second bounce
- Normal of the first and second bounce
- Texture value of the first bounce
- u, v lens coordinates
- t for motion blur (never used)
- x, y and z for direction of the first bounce

The output color (final color) of a sample is also stored but is not part of the scene features. The algorithm is memory intensive since it needs to store 26 floats for each sample. Since every pixel needs around 8 spp to give good results, the total amount of memory needed when rendering with a resolution of 1280x720 is $26 * 4Byte * 8spp * 1280 * 720 \approx 731MB$. Each RPF sample-vector parameter is depicted in table 5.1.

Description	nr. of floats	Details
Screen position	2 floats	Random position inside a pixel
Random parameters (u, v, t, x, y, z)	6 floats	Lens coordinates, time and first bounce direction
Scene features	15 floats	Scene information
Sample color	3 floats	Color of this sample

Table 5.1

5.1.2 Pre-processing

The sample-vectors are collected for each pixel in the screen, which is where the actual filtering starts. RPF applies multiple iterations over the entire image, where every iteration has its own

neighborhood size. Just like a conventional cross bilateral filter this neighborhood is located around the pixel we are filtering. By going from a large size filter to a smaller one in multiple iterations, the algorithm can get a better estimate of the large and small scale details of the region around the pixel. In their implementation 4 iterations are used, where the neighborhood sizes for each iteration are: $55, 35, 17, 7$. Each sample inside a pixel gets filtered individually and after all iterations are done, each pixel with x samples is box filtered and written to the output buffer (final image). To cut down the filtering time a pre-processing step is applied to get rid of samples that do not share similar scene features.

We create a vector of undetermined size that will store the neighboring samples (N) that are accepted by the pre-process. First the samples from the pixel being filtered are put into N to avoid having to store them in a separate vector. Then we compute the mean and standard deviation for all the scene features in the current pixel i.e. with 8 spp and 15 scene features per sample, we get 1 mean scene feature vector and one standard deviation scene feature vector. The maximum amount of samples that can be accepted to N is: kernel-width$^2 * (8spp/2)$. That means the potential amount samples accepted (M) in N for the largest kernel-width (55) is: 12100 samples. To avoid having to store every sample twice, we just store the index to the sample instead of the whole vector.

Now we loop $M - x$ times over all the samples in the neighborhood since the first x samples are from the pixel we are filtering. We pick a random sample from the neighborhood based on a Gaussian distribution that depends on the distance of the neighboring pixel (Figure 5.4) i.e. pixels are sampled based on screen position which is the same as a spatial weight for cross-bilateral filtering. Then we randomly sample one out of x samples in the pixel. If a sample gets sampled more than once, we ignore it and continue to the next sample.

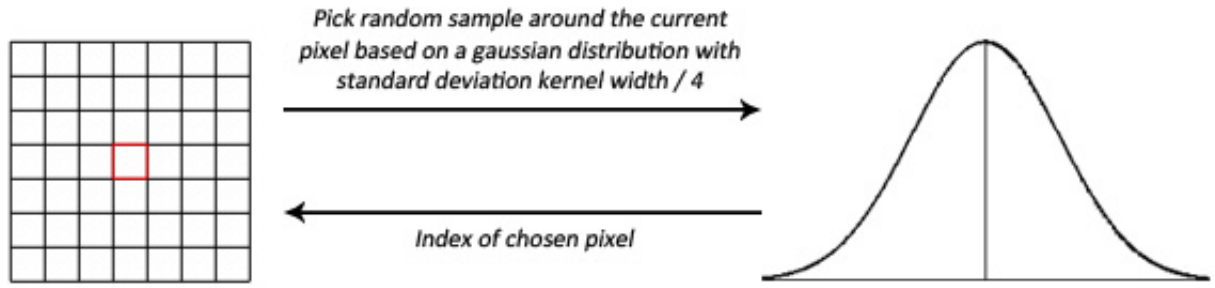


Figure 5.4

If all the scene features from the randomly picked sample fall within 3 standard deviations from the mean than the sample is added to N. The authors empirically determined that the world-position has to be within 30 standard deviations from the mean, because the range of a world-position is much higher than the normal and color ranges. After we are done looping over $M - x$ samples, each sample in N is normalized. To normalize each sample we first calculate the mean (μ) and standard deviation (σ) of every sample in N. Then we normalize each sample by: $(sample - \mu)/\sigma$. By randomly selecting neighboring samples, the first term from

equation 5.1 can be dropped which reduces the filter to:

$$w_{ij} = \exp[-\frac{1}{2\sigma_c^2}\sum_{1\leq k\leq 3}\alpha_k(\bar{c}_{i,k}-\bar{c}_{j,k})^2]\times \exp[-\frac{1}{2\sigma_f^2}\sum_{1\leq k\leq m}\beta_k(\overline{f}_{i,k}-\overline{f}_{j,k})^2], \quad (5.2)$$

The pre-processing algorithm is shown in Algorithm 6.

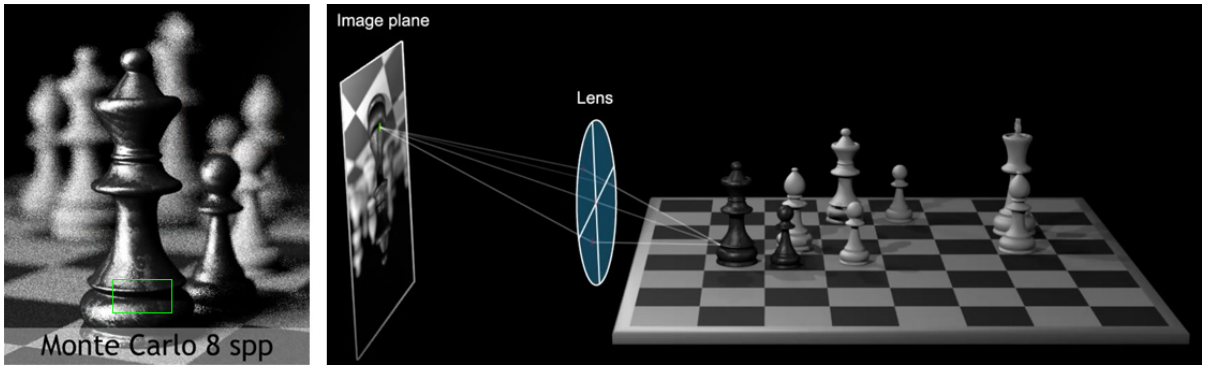
Algorithm 6 Pre-process samples

function PRE-PROCESS(Set of samples in pixel P, filter size b, maximum number of samples M)
  $\sigma_p \leftarrow b/4, N \leftarrow P$
  Compute mean (m_p^f) and standard deviation (σ_p^f) of the scene features in pixel P
  for q = 1 to M - spp **do**
    Select random sample j from sample inside neighborhood but outside P with gaussian distribution based on σ_p
    $flag \leftarrow 1$
    for scene feature k to m **do**
      if $|f_{j,k}-m_{p,k}^f| > 3|30\sigma_{p,k}^f$ and $|f_{j,k}-m_{p,k}^f| > 0.1$ or $\sigma_{p,k}^f > 0.1$ **then**
        $flag \leftarrow 0$
      end if
    end for
    if $flag = 1$ **then**
      $N \leftarrow sample\ j$
    end if
    Compute mean (m_N^x) and standard deviation (σ_N^x) of all samples in N
    for all samples i in N **do** ▷ Normalize all samples in N
      $\bar{x_i} \leftarrow (x_i - m_N^x)/\sigma_N^x$
    end for
  end for
  return N
end function

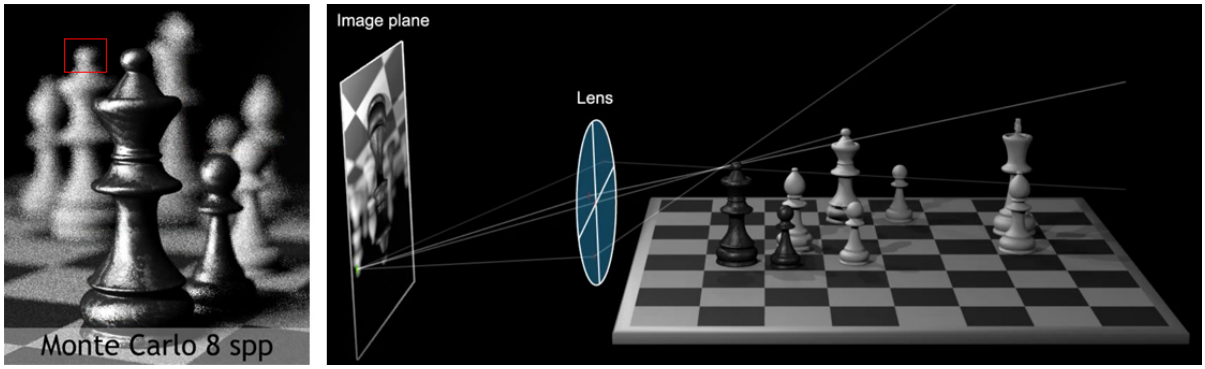
5.1.3 Calculating the statistical dependencies

Step three of the algorithm is the most expensive part in terms of computational cost. Statistical dependencies between the random parameters/screen position and rendering output are calculated using mutual information. Mutual information is used as a metric to determine how much information one set of samples share with another set of samples. If little or no information is shared the mutual information is low and vice versa. In the field of computer science it is a measure for the amount of bits shared between data. In other words: when a random parameter and a scene feature depend on each other, the mutual information is high. Figure 5.5 shows an intuitive example of the relationship between lens coordinates and sample color/ hit position. The statistical dependencies that will be calculated are listed below where the first two are used to calculate α and number four and five to calculate β.

1. Dependency of the k-th Sample color channel (r,g,b) on all random parameters ($D^r_{c,k}$)
2. Dependency of the$k$-th Sample color channel (r,g,b) all screen positions ($D^p_{c,k}$)
3. Dependency of the k-th Sample color channel (r,g,b) on all scene features ($D^f_{c,k}$)
4. Dependency of the k-th scene feature on all random parameter ($D^r_{f,k}$)
5. Dependency of the k-th scene feature on all screen positions($D^p_{f,k}$)
6. Dependency of the k-th scene feature on all sample colors ($D^c_{f,k}$)



(a) The pixel being evaluated is in-focus. When the lens coordinates change, the hit position of the ray in world-coordinates is unchanged i.e. mutual information is low, since the change in lens coordinates do not affect the rays hit position



(b) The pixel being evaluated is out-of-focus. When the lens coordinates change, the hit position of the ray in world-coordinates is changed i.e. mutual information is high, since the change in lens coordinates affect the rays hit position

Figure 5.5: Relationship between random parameters and scene features

The authors did not provide any implementation for their method, but they did give a clue to where they based their implementation of mutual information on Peng [23]. The general formulation to calculate the mutual information between a two random variables X and Y is:

$$\mu(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)}, \tag{5.3}$$

where these probabilities are calculated over the neighborhood N around a pixel e.g. X can be all world-positions (x, y or z) from N and Y can be all random parameters u (lens coordinate) from N. Algorithm 7 shows how mutual information is calculated given X, Y and N.

Algorithm 7 Calculate Mutual information between X and Y

function MUTUALINFORMATION(neighborhood of normalized samples N, X and Y)
    CreateHistograms(N, X, Y)
    $entropyX$, $entropyY$, $entropyXY$
    CalculateEntropies($entropyX$, $entropyY$, $entropyXY$)
    return entropyX + entropyY - entropyXY
end function

function CREATEHISTOGRAMS(N, X and Y)
    $size \leftarrow N.size$
    for i = 0 to size - 1 **do**
        $bucketx \leftarrow$ CalculateBucket($N[i]$)
        $buckety \leftarrow$ CalculateBucket($N[i]$) ▷ $N[i]$ does not get the whole sample-vector but only the element we need from that vector. In our example that would be x, y or z for $bucketx$ and u for $buckety$

        $histogramX[bucketx] + +$
        $histogramY[buckety] + +$
        $histogramXY[bucketx * numBuckets + buckety] + +$
        $numberOfSamples + +$
    end for ▷ Histograms and numberOfSamples are stored somewhere so that MutualInformation can acces them
end function

function CALCULATEENTROPIES($entropyX$, $entropyY$ and $entropyXY$)
    $entropyX \leftarrow 0$, $entropyY \leftarrow 0$, $entropyXY \leftarrow 0$
    for i = 0 to numberOfBuckets - 1 **do**
        if $histogramX[i] > 0$ **then**
            $probabilityX \leftarrow histogramX[i]/numberOfSamples$
            $entropyX \leftarrow entropyX + (-probabilityX \times \log_2(probabilityX))$
        end if
        if $histogramY[i] > 0$ **then**
            $probabilityY \leftarrow histogramY[i]/numberOfSamples$
            $entropyY \leftarrow entropyY + (-probabilityY \times \log_2(probabilityY))$
        end if
    end for
    for i = 0 to (numberOfBuckets * numberOfBuckets) - 1 **do**
        if $histogramXY[i] > 0$ **then**
            $probabilityXY \leftarrow histogramXY[i]/numberOfSamples$
            $entropyXY \leftarrow entropyXY + (-probabilityXY \times \log_2(probabilityXY))$
        end if
    end for
end function

To calculate the dependencies used to calculate α we use the following equations:

$$D^r_{c,k} = \sum_{1 \leq l \leq n} D^{r,l}_{c,k} = \sum_{1 \leq l \leq n} \mu(\overline{c}_{N,k}; \overline{r}_{N,l}), \tag{5.4}$$

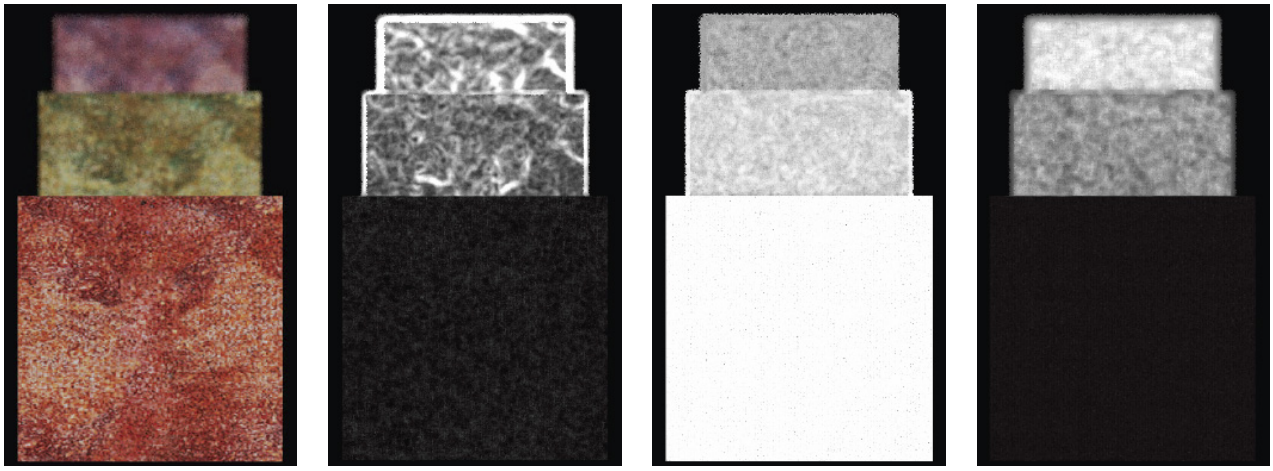
$$D^p_{c,k} = \sum_{1 \leq l \leq 2} D^{p,l}_{c,k} = \sum_{1 \leq l \leq 2} \mu(\overline{c}_{N,k}; \overline{p}_{N,l}), \tag{5.5}$$

where $D^{r,l}_{c,k}$ is the dependency of the k-th color channel and the l-th random parameter and $D^{p,l}_{c,k}$ is the dependency of the k-th color channel and the l-th screen position. $\overline{c}_{N,k}$ are all k-th color channels in N, $\overline{r}_{N,l}$ are all l-th random parameters in N and $\overline{p}_{N,l}$ are all l-th screen positions in N. To calculate the dependencies that we need for the calculation of β we use the following equations that are similar to equation 5.4 and 5.5:

$$D^r_{f,k} = \sum_{1 \leq l \leq n} D^{r,l}_{f,k} = \sum_{1 \leq l \leq n} \mu(\overline{f}_{N,k}; \overline{r}_{N,l}), \tag{5.6}$$

$$D^p_{f,k} = \sum_{1 \leq l \leq 2} D^{p,l}_{f,k} = \sum_{1 \leq l \leq 2} \mu(\overline{f}_{N,k}; \overline{p}_{N,l}), \tag{5.7}$$

Since the scene features and sample color are not only depending on the random parameters but also on screen position, we need to have a weighted average between these dependencies. These weighted averages are called fractional contributions. Figure 5.6 will show a practical example of these dependencies combined to one fractional contribution. In this example the front panel is in-focus and the back panel out-of-focus. The color depends more on screen position and less on random parameters for objects in-focus and less on screen position and more on the random parameters for objects out-of-focus. Figure 5.6(d) shows the fractional contribution of the color on the random parameters (W^r_c).



(a) Input render

(b) Visualization for the dependency of the color on the random parameters

(c) Visualization for the dependency of the color on the screen position

(d) Visualization for the fractional contribution of the color on the random parameters

Figure 5.6: Pictures taken from Sen and Darabi [12]

To calculate α we need the fractional contribution for each color channel on all the random parameters:

$$W^r_{c,k} = \frac{D^r_{c,k}}{D^r_{c,k} + D^p_{c,k} + \epsilon}. \tag{5.8}$$

α_k (weight for each color channel) can be calculated with the following equation:

$$\alpha_k = max(1 - (1 + 0.1t)W^r_{c,k}, 0), \tag{5.9}$$

where $(1 + 0.1t)$ is used to increase the fractional contribution weight with increasing filter iteration t (4 iterations). The intuition behind α is when it becomes lower, w_{ij} from equation 5.2 becomes higher meaning more filtering will occur. This directly translate into saying: when the fractional dependency of the color on the random parameters $W^r_{c,k}$ increases (in general this means more noise), α will decrease causing heavier filtering.

To calculate β we need two additional fractional contributions namely: the fractional dependency of the color on the k-th scene feature (equation 5.12) and the fractional dependency of the k-th scene feature on the random parameters (equation 5.13). We need to calculate some additional dependencies which are the dependency of all colors on the k-th scene feature (equation 5.10) and the dependency of all colors on all random parameters, screen positions and scene features (equation 5.11).

$$D^{f,k}_c = \sum_{1 \le l \le 3} D^{f,k}_{c,l} = \sum_{1 \le l \le 3} \mu(\overline{c}_{N,l}; \overline{f}_{N,k}), \tag{5.10}$$

$$D^r_c = \sum_{1 \le l \le 3} D^r_{c,k}, D^p_c = \sum_{1 \le l \le 3} D^p_{c,k}, D^f_c = \sum_{1 \le l \le 3} D^f_{c,k}, \tag{5.11}$$

$$W^{f,k}_c = \frac{D^{f,k}_c}{D^r_{c,k} + D^p_{c,k} + \epsilon}, \tag{5.12}$$

$$W^r_{f,k} = \frac{D^r_{f,k}}{D^r_{f,k} + D^p_{f,k} + \epsilon}. \tag{5.13}$$

Now we can calculate β_k with the following equation:

$$\beta_k = W^{f,k}_c \times max(1 - (1 + 0.1t) \times W^r_{f,k}, 0). \tag{5.14}$$

All values of α and β are always between 0 and 1. which is enforced by the fractional contributions and equations 5.9 and 5.14.

5.1.4 Filtering

The final step of the algorithm is the filtering process. Here we filter every sample for every pixel in the image separately. After each filtering step, the filtered pixel samples are stored and will become the new samples for the next iteration. It is important to note that for step 3 of the algorithm (5.1.3), only the original noisy samples are used to calculate the mutual information.

As previously mentioned the variance terms σ^2_c and σ^2_f will be discussed here. Wherever the color depends on a lot of random parameters we want to filter more. Equation 5.16 adjusts the noise level according to the dependence on the random parameters:

$$W^r_c = \frac{1}{3} \times (W^r_{c,1} + W^r_{c,2} + W^r_{c,3}) \tag{5.15}$$

$$\sigma_c^2 = \sigma_f^2 = \frac{\sigma^2}{(1 - W_c^r)^2}, \tag{5.16}$$

where σ^2 depends on a empirically determined number σ_8^2 which depends on whether the scene is indoor (e.g. noisy $\sigma_8^2 = 0.02$) or outside (less noise $\sigma_8^2 = 0.002$):

$$\sigma^2 = 8\sigma_8^2/spp. \tag{5.17}$$

In general more spp means less variance in the image which is why the authors use spp in equation 5.17. To filter one sample we simply apply the following equation:

$$c_{i,k}^{''} = \frac{\sum_{j \in N} w_{ij} c_{j,k}^{'}}{\sum_{j \in N} w_{ij}}, \tag{5.18}$$

where $c_{i,k}^{''}$ is the filtered sample color which is taken to the next iteration and $c_{j,k}^{'}$ is the current color of a neighboring pixel with weight w_{ij}. The numerator is the sum of all weights calculated using equation 5.2 and the denominator is the normalization term i.e. sum of all weights. The filter continues until all iterations are done. At the end of the filtering process each pixel is box-filtered, which is an average over all filtered samples in the pixel (3.1.1).

5.2 Experimentation and Results

Since our main goal is speed and accuracy, the experimentation will be focused on these two aspects. All of the experimentations are done on a system with the following specifications:

Type	Used system	Authors system
CPU	Intel Core i5-3570K (3.5GHz)	Dual quad-core (8-threads each) Xeon X5570 (3.05 GHz)
Memory	8 GB	16 GB

Table 5.2

Since the authors did not provide any implementation, it is hard to validate their results using a similar system. The authors also state their algorithm should be easy to parallelize, so I made the assumption that they used as many threads (maximum of 16) as possible. My implementation only uses 1-thread to keep the memory usage of the algorithm down to a minimum. It is also hard to validate their results using the renderer proposed in chapter 4, so new scenes are used to perform experiments. Since the renderer does not support Multiple Importance Sampling and adaptive sampling to reduce noise during render time, we can test the stability of RPF under these conditions. Since the renderer supports depth-of-field we are able to test RPF with scenes that use depth-of-field.

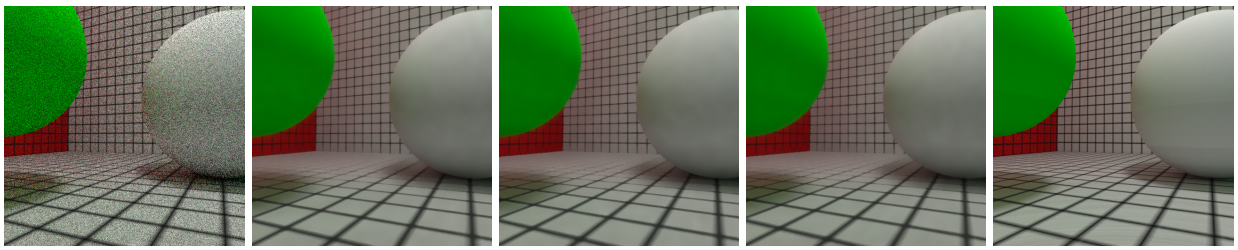
5.2.1 RPF speed

When testing for speed, we purely look at the running time of the algorithm for one filtered frame. For potential real-time purposes we want the upper and lower-bound running time of the filter to close to each other i.e. same amount of time for every filtered pixel sample. The

following experiments are setup with different scenario's in mind which apply to real-time rendering. The first scenario tests stability of the filtering time when varying the complexity of the scene i.e. more complex materials and shapes. The second scenario will test the filtering time when the scene scale changes, which means only the units get scaled e.g. from meters to inches.

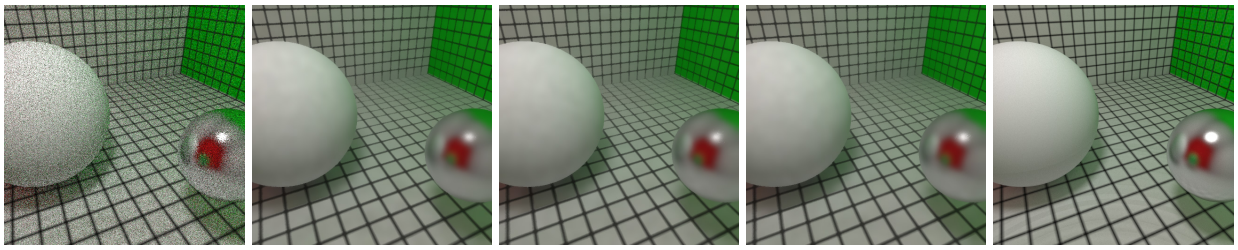
All renders will be done at a resolution of $512x512$ measured from beginning to end over an average of 3 runs. The parameters for the filter are configured as proposed in the paper: $\sigma_8^2 = 0.02$, because the scene is indoors. The parameter that rejects samples based on similarity of the scene features in the pre-processing step is set to $\times 3$ for normals and texture and $\times 30$ for the world-position i.e. unchanged.

Figure 5.7 shows the diffuse scene's unfiltered (a), filtered (b, c, d) and ground-truth (e) outputs. Figure 5.8 and Figure 5.9 show similar results for different scene's. Every filtered image depicts the filter time in seconds on one CPU core.



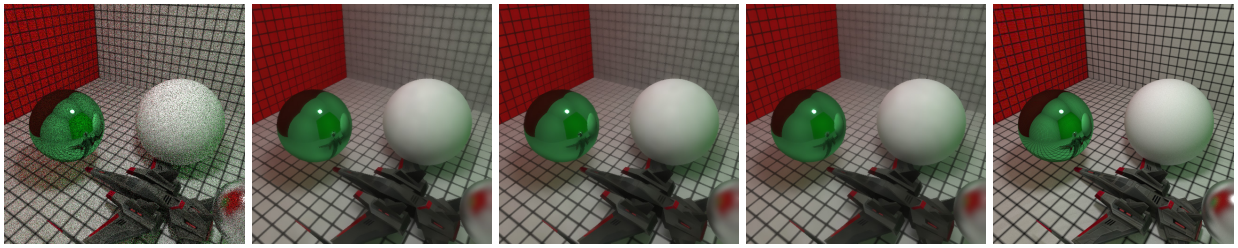
(a) Input rendered with 8spp | (b) Run 1: 4671 seconds | (c) Run 2: 4641 seconds | (d) Run 3: 4559 seconds | (e) Ground truth rendered with 2048 spp

Figure 5.7



(a) Input rendered with 8spp | (b) Run 1: 4111 seconds | (c) Run 2: 4067 seconds | (d) Run 3: 4039 seconds | (e) Ground truth rendered with 2048 spp

Figure 5.8



(a) Input rendered with 8spp (b) Run 1: 5473 seconds (c) Run 2: 5434 seconds (d) Run 3: 5331 seconds (e) Ground truth rendered with 2048 spp

Figure 5.9

We render the same scene's scaled down with factors of $\times 2$, $\times 4$ and $\times 8$ with the same parameters that we used in the first test. Just like the last experiment all tests will be performed over 3 separate runs. The filter times (in seconds) for each scene and run (including un-scaled scene) are depicted in table 5.3.

Scene	Run 1	Run 2	Run 3	μ	σ
Cornell Diffuse	4671s	4641s	4559s	4623.67	47.34
Cornell Diffuse $\times 2$	4704s	4696s	4660s	4686.67	19.14
Cornell Diffuse $\times 4$	4772s	4776s	4748s	4765.33	12.36
Cornell Diffuse $\times 8$	5678s	5654s	5624s	5652.00	22.09
Cornell Glossy	4111s	4067s	4039s	4072.33	29.63
Cornell Glossy $\times 2$	4065s	4059s	4064s	4062.67	2.62
Cornell Glossy $\times 4$	4094s	4092s	4092s	4092.67	0.94
Cornell Glossy $\times 8$	4400s	4394s	4394s	4396.00	2.83
Cornell Plane	5473s	5434s	5331s	5412.67	59.90
Cornell Plane $\times 2$	3712s	3681s	3745s	3712.67	26.13
Cornell Plane $\times 4$	3727s	3691s	3740s	3719.33	20.73
Cornell Plane $\times 8$	3745s	3740s	3741s	3742.00	2.16

Table 5.3

5.2.2 RPF quality

To determine the quality of an image we need to find a suitable metric that will tell us how good an image is compared to the ground truth. A common metric is Mean Square Error (MSE) which is calculated for noisy image K with the following equation:

$$MSE = \frac{1}{m \times n} \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} [I(i,j) - K(i,j)]^2, \tag{5.19}$$

where MSE is the variance between the ground truth image I and noisy image K and m and n are the image width and height ([24]). To determine the MSE of each filtered image we use a software package called MSU [25]. For all of the experiments we measure the MSE of all images in CIELUV colors space [26].

Here we will directly compare the quality of the filtered output for each scale with the ground truth. For each scene we will show the MSE for the ground truth vs. input and ground truth vs. filtered for each scale i.e. unscaled, scaled down $\times 2$, $\times 4$ and $\times 8$. Figure 5.10, 5.11 and 5.12 show the Diffuse, Glossy and Plane scene's respectively.

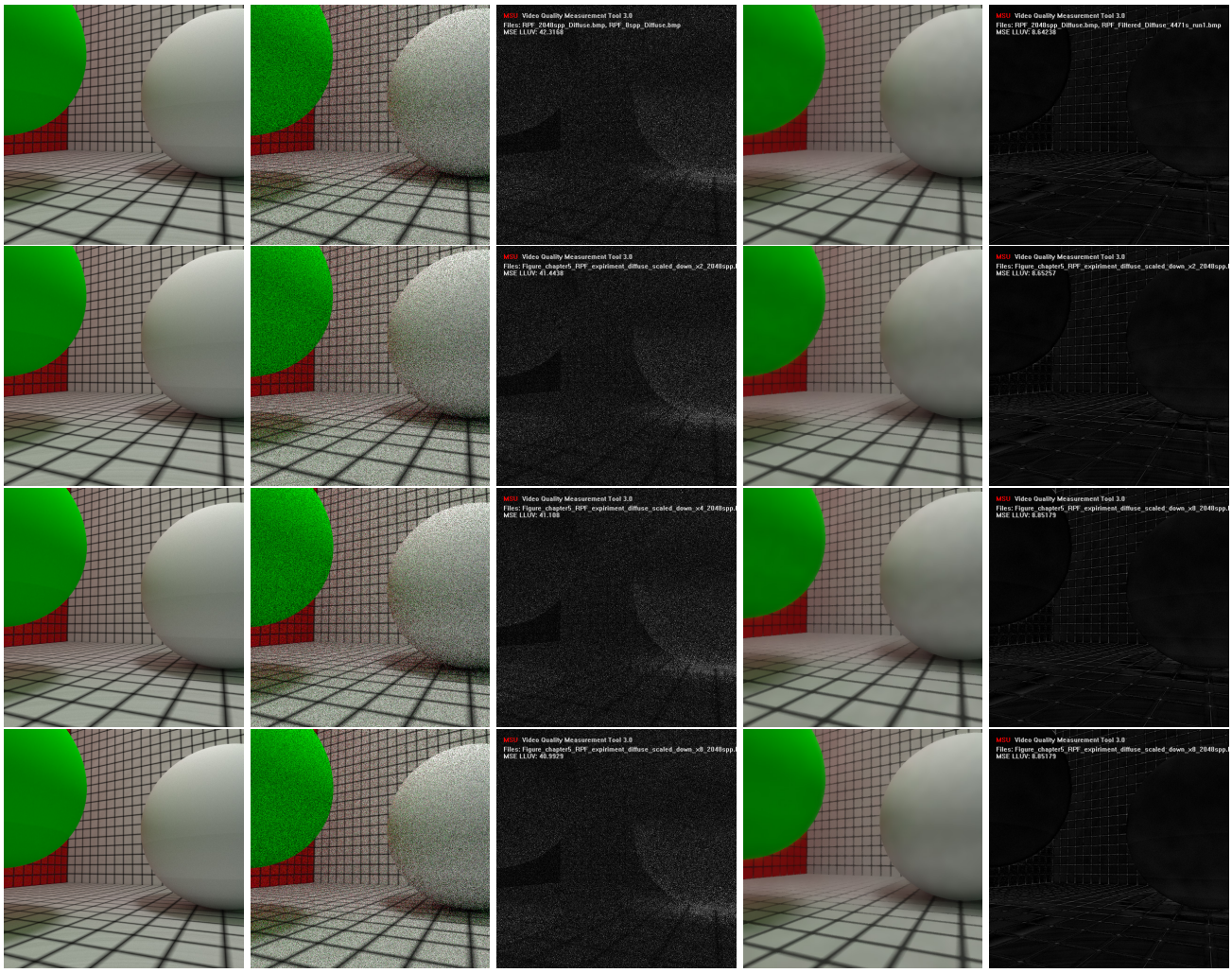


Figure 5.10: Cornell diffuse scene. From left to right: ground-truth (2048spp), unfiltered (8spp), unfiltered MSE, filtered, filtered MSE. From top to bottom: unscaled, scaled down $\times 2$, $\times 4$ and $\times 8$

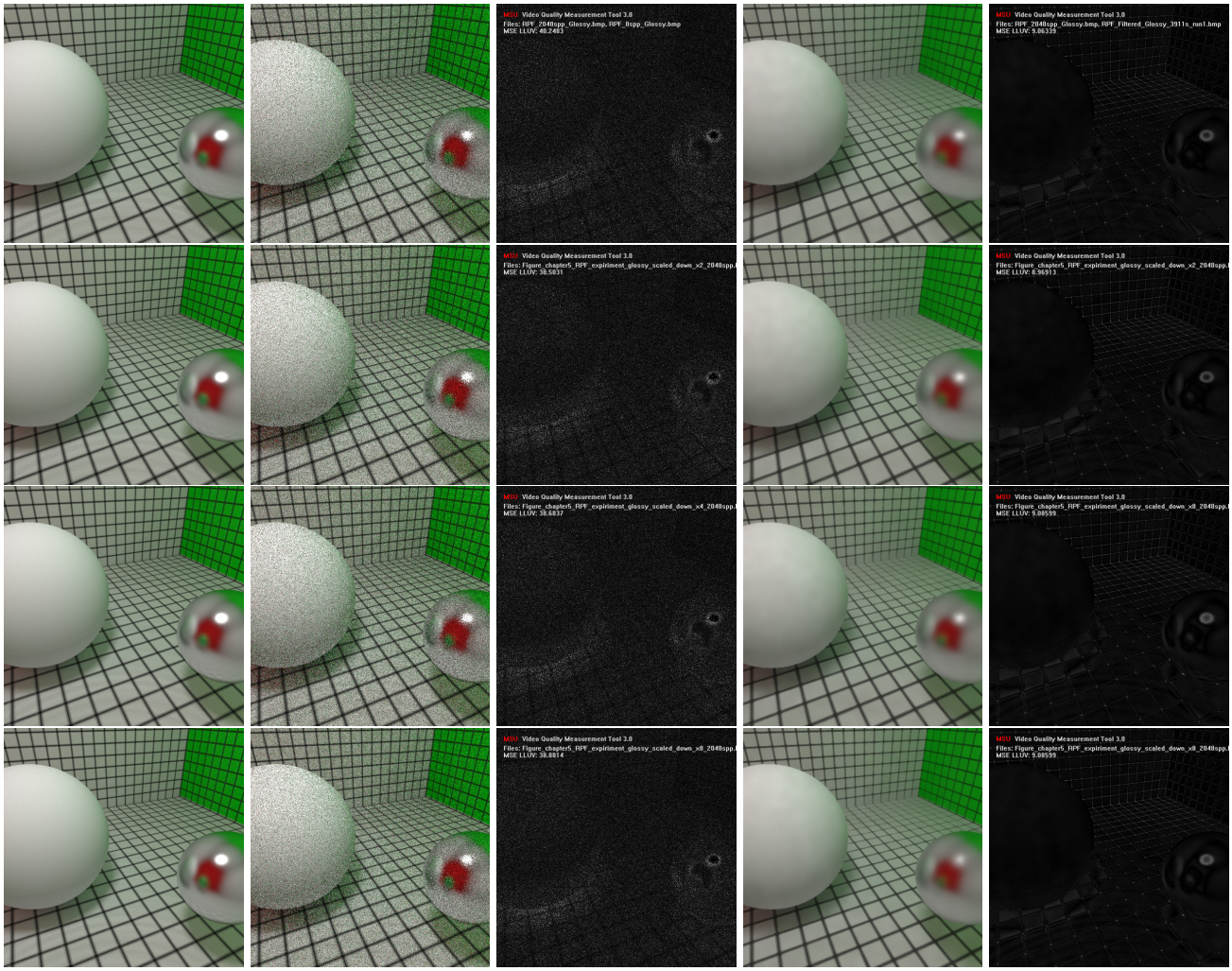


Figure 5.11: Cornell glossy scene. From left to right: ground-truth (2048spp), unfiltered (8spp), unfiltered MSE, filtered, filtered MSE. From top to bottom: unscaled, scaled down $\times 2$, $\times 4$ and $\times 8$

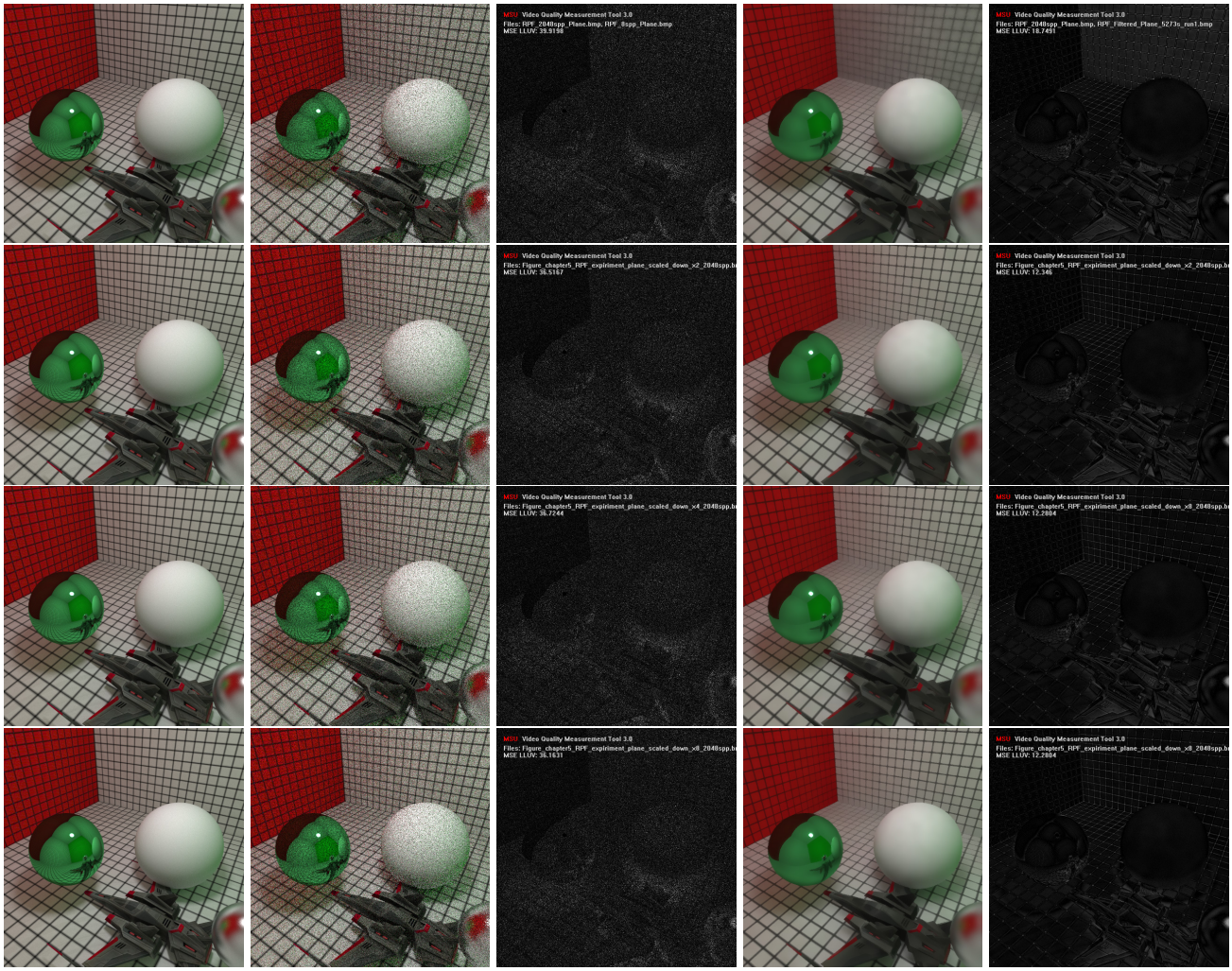


Figure 5.12: Cornell plane scene. From left to right: ground-truth (2048spp), unfiltered (8spp), unfiltered MSE, filtered, filtered MSE. From top to bottom: unscaled, scaled down $\times 2$, $\times 4$ and $\times 8$

The improvement in terms of MSE from unfiltered to filtered is significant ($\approx 4\times$ lower MSE). In table 5.4 we will show the MSE measurements for each scene and scale (unfiltered vs. filtered MSE).

Scene	MSE unfiltered	MSE filtered	μ_u	σ_u	μ_f	σ_f
Cornell Diffuse	42.317	8.642 (-0.093)				
Cornell Diffuse $\times 2$	41.444	8.653 (-0.082)	41.466	0.519	8.735	0.090
Cornell Diffuse $\times 4$	41.108	8.794 (0.059)				
Cornell Diffuse $\times 8$	40.993	8.852 (0.117)				
Cornell Glossy	40.248	9.063 (0.026)				
Cornell Glossy $\times 2$	38.503	8.969 (-0.068)	39.079	0.688	9.037	0.044
Cornell Glossy $\times 4$	38.684	9.029 (-0.008)				
Cornell Glossy $\times 8$	38.881	9.086 (0.049)				
Cornell Plane	39.920	18.749 (4.863)				
Cornell Plane $\times 2$	36.517	12.346 (-1.540)	37.331	1.508	13.886	2.808
Cornell Plane $\times 4$	36.724	12.170 (-1.716)				
Cornell Plane $\times 8$	36.163	12.280 (-1.606)				

Table 5.4: μ_u and σ_u are the mean and standard deviation of the unfiltered MSE and μ_f and σ_f are the mean and standard deviation of the filtered MSE. In the MSE filtered column the difference from the mean is depicted between ().

We compare the MSE of the filtered images from different scales with each other to get a measure of the difference in quality. Each comparison will be done against the un-scaled filtered image. Each MSE output is brightened to make the difference more visible. Figure 5.13, 5.14 and 5.15 each show different scenes.

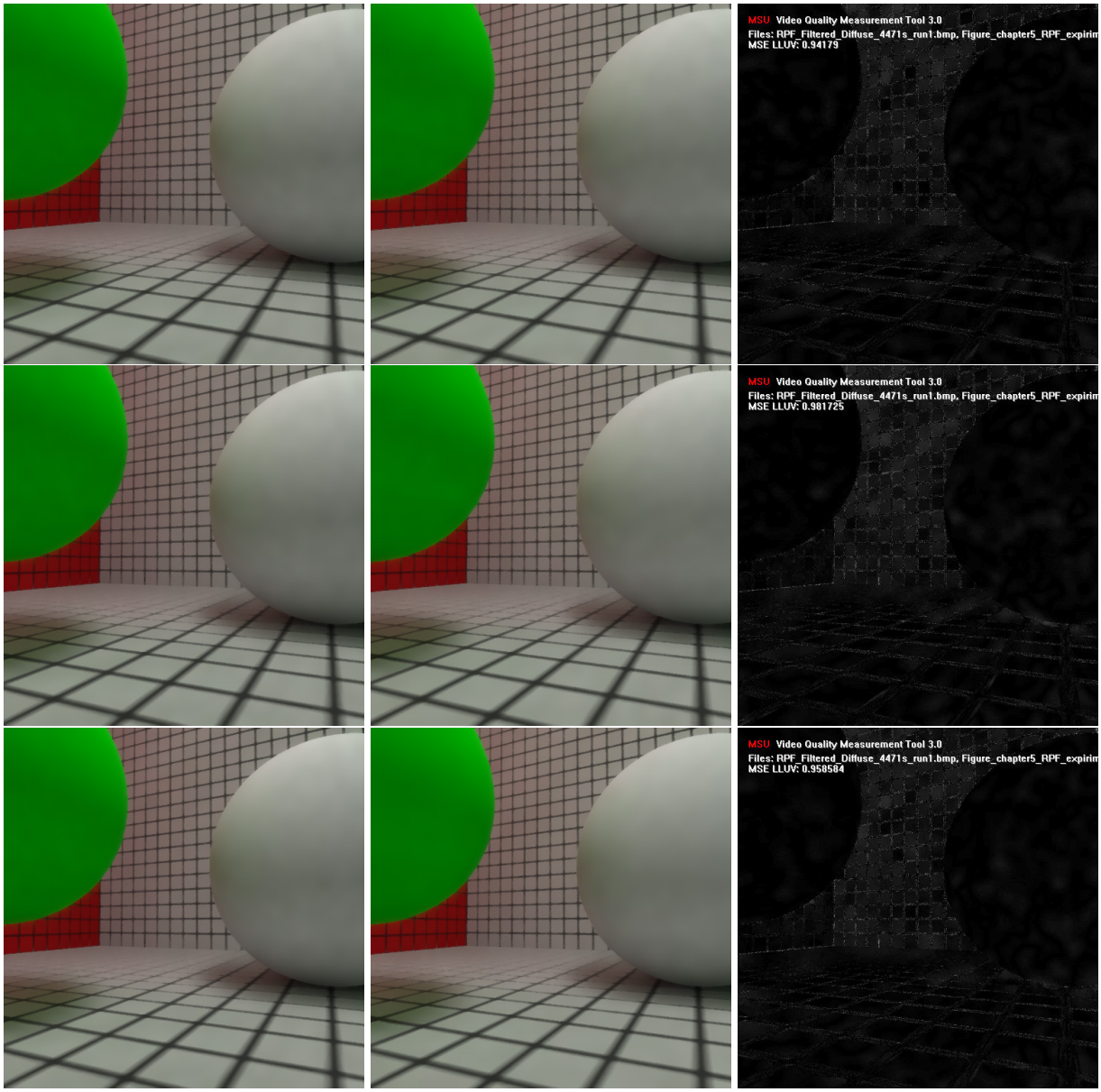


Figure 5.13: Cornell diffuse scene. From left to right: filtered unscaled, filtered scaled, filtered unscaled vs scaled MSE. From top to bottom: unscaled vs scaled down $\times 2$, unscaled vs scaled down $\times 4$ and unscaled vs scaled down $\times 8$.

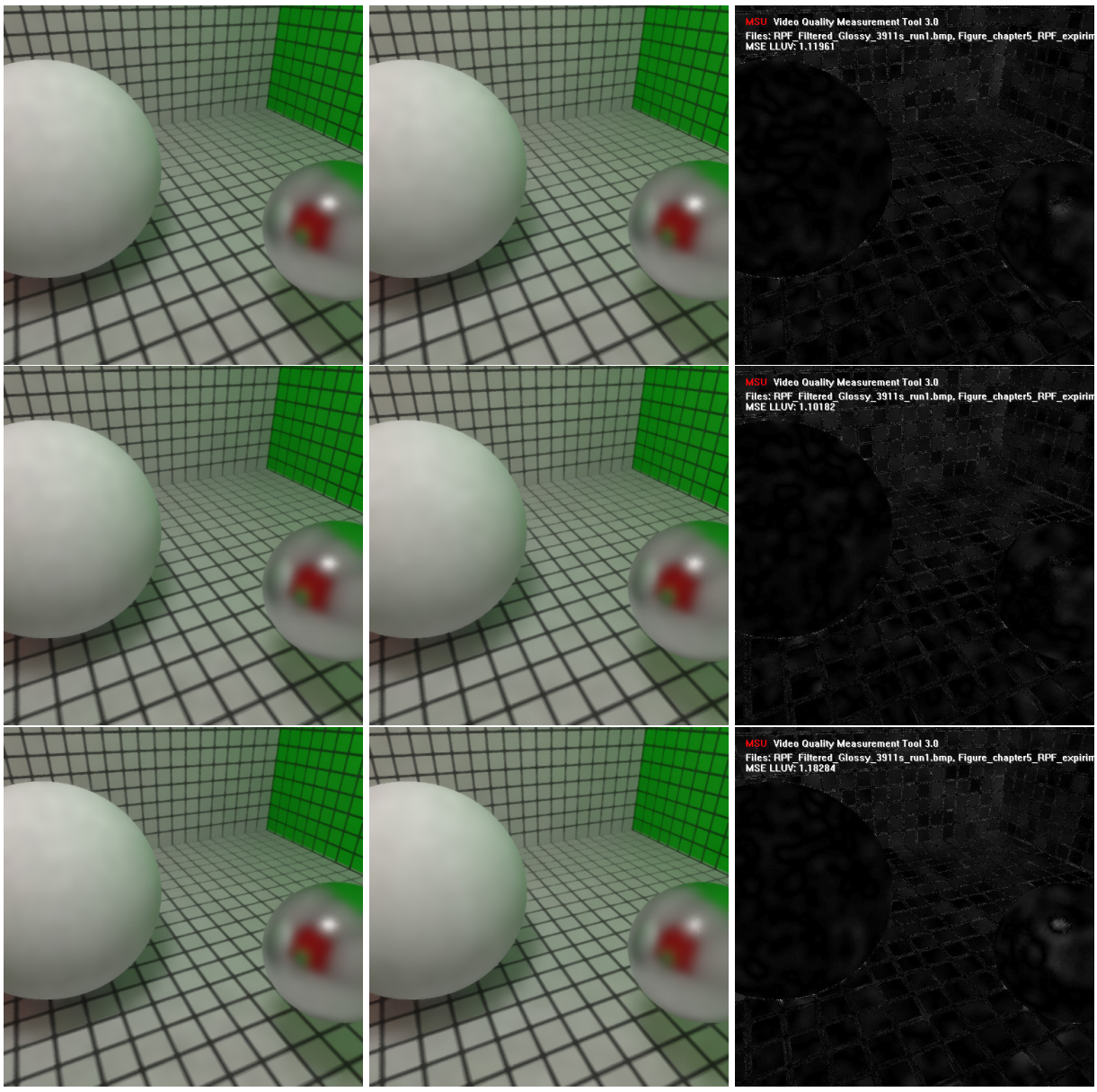


Figure 5.14: Cornell glossy scene. From left to right: filtered unscaled, filtered scaled, filtered unscaled vs scaled MSE. From top to bottom: unscaled vs scaled down $\times 2$, unscaled vs scaled down $\times 4$ and unscaled vs scaled down $\times 8$.

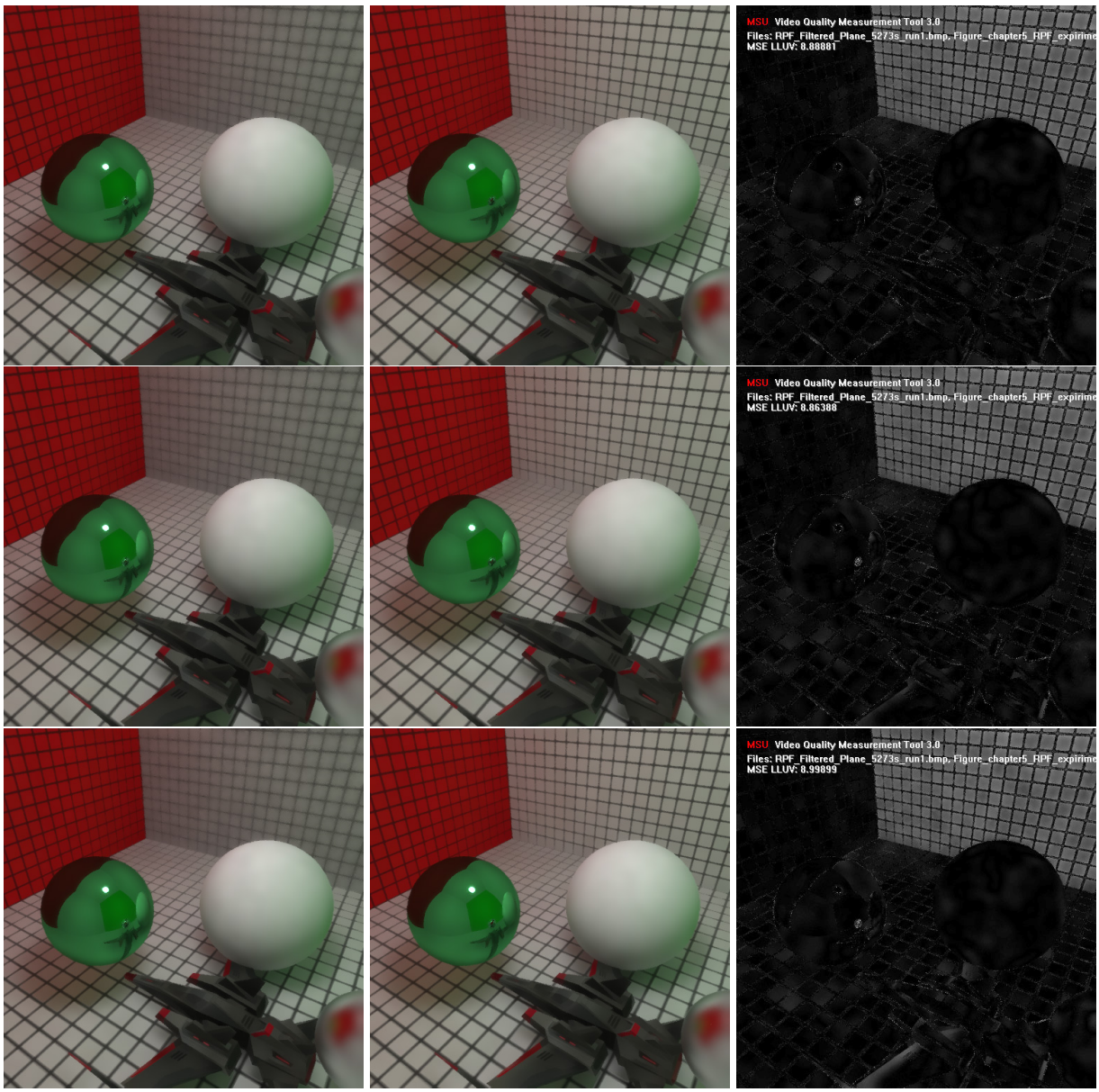


Figure 5.15: Cornell diffuse scene. From left to right: filtered unscaled, filtered scaled, filtered unscaled vs scaled MSE. From top to bottom: unscaled vs scaled down $\times 2$, unscaled vs scaled down $\times 4$ and unscaled vs scaled down $\times 8$.

We can clearly see in the filtered outputs that texture details in the reflective material are blurred out. This happens because RPF only stores texture and color information for the first bounce, but ignores multiple bounces.

5.2.3 Conclusion

The filtering capabilities of RPF are impressive, but the speed of the algorithm makes it unsuitable for a real-time implementation. The pre-process in the RPF algorithm picks random samples in the neighborhood which is bad for memory coalescence (more about this in the next chapter). Furthermore, the amount of memory needed to store N neighbors is random with a lower bound of 8 and a upper bound of 12100 sample-vectors. Since we only have to

store the index to each neighboring sample-vector the upper bound memory usage for N is $12100 \times 4Byte = 48400Byte$. The memory usage of the algorithm is high and unpredictable. Most GPU devices have enough memory to store all the sample-vectors, but not enough memory to store all the neighbors N and histograms needed to calculate the mutual information. The main bottlenecks of RPF that makes it unsuitable for a real-time GPU implementation are:

- Memory usage and memory access patterns.
- High computational cost for each pixel.
- Unpredictable running time for each frame.

Quality drawbacks of the filter include:

- Loss of texture detail in reflections
- Over blurring a sky-box or environment map (will be discussed in chapter 7)

RPF offers good insight on how noise relates to other parameters. The use of a modified cross bilateral filter with additional weights is also a good addition. We can use these ideas to create a new filter.

Chapter 6

Data Driven Filtering

In this chapter we present a filtering method based on the cross-bilateral filter that uses variance instead of mutual information to determine the noise level inside a pixel. The aim is to use more scene information for the filtering process while still focusing on speed and quality. We will also focus on the main constraints for GPU implementation i.e. memory coalescence, maximize usage of resources and optimal program flow. Instead of filtering independent samples for each pixel like RPF, we filter the average color of the pixel. The main goal is: *Create a filter that reduces the noise to acceptable levels at low sampling rates (8 spp), while preserving real-time performance*, where real-time is 24 fps. Last chapter we discussed RPF and we will use the ideas of this state of the art approach to develop a new filter. The main constraints for the filter are:

- The filter should run as a post-process next to the path tracer
- The filter has to be fast enough to allow the path tracer to keep its constraints
- The filter has to be similar in quality to recent filtering algorithms at low sampling rates i.e. 8 samples (rays) per pixel

In section 6.1 we will show the pipeline of the algorithm. Section 6.2 will discuss the implementation details including data structures, parameters and the algorithm. In the last section (6.3) we will discuss different kinds of optimizations such as memory and speed optimizations. These optimizations will take the three main constraints for optimal use of GPU hardware into account, that we discussed previously.

6.1 Algorithm pipeline

Figure 6.1 shows a simplified pipeline of the processing steps needed to go from an unfiltered to a filtered image. First the path tracer renders any amount of samples iteratively i.e. can be 1 to x samples rendered each frame depending on the target frame rate. After one sample is rendered a new average (mean) and variance Geometry Buffers (GBuffer) are calculated. Each new sample rendered is added to the mean and variance until the renderer decides it has enough samples for the current frame. The same is done for the direct and indirect color. The process of rendering samples, calculating mean and variance and filtering the pixel happens each frame. It is important to know that each new frame can contain x samples per pixel before it gets filtered. If we want a noise free image for each frame that is rendered, we need at least 8 spp before we filter the image.

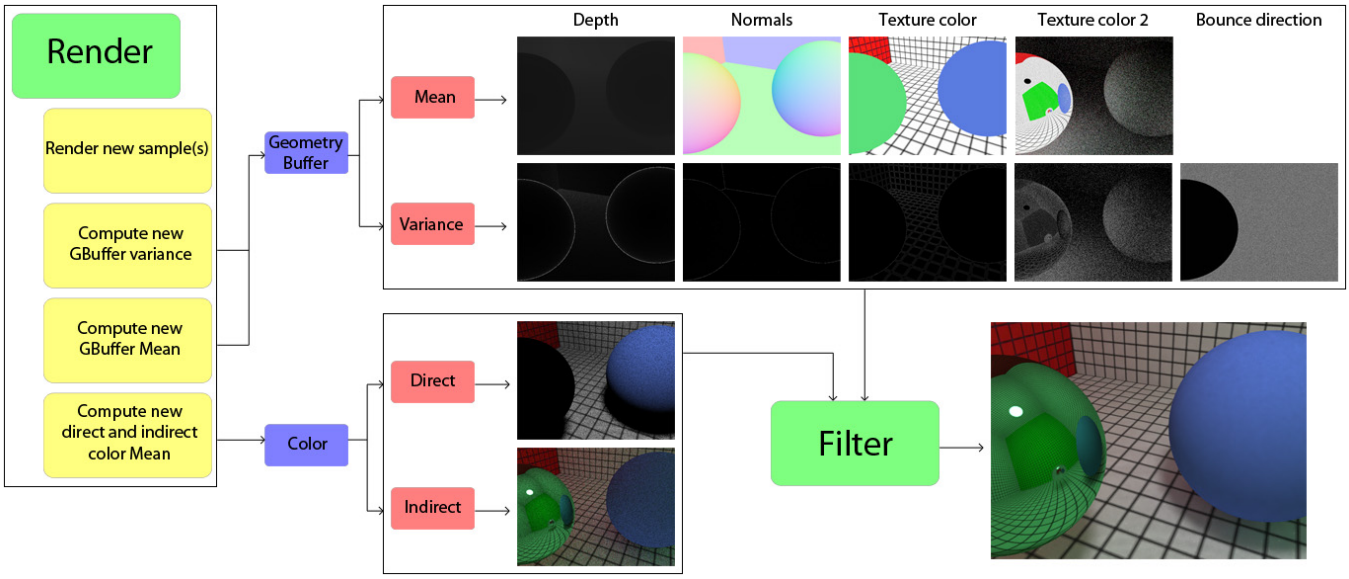
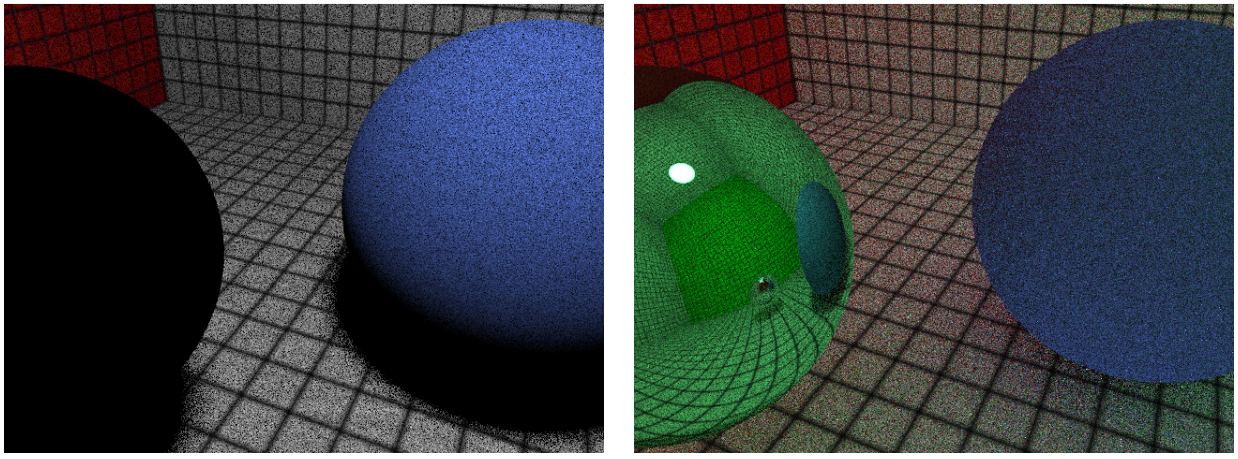


Figure 6.1

The GBuffer's depicted in figure 6.1 show what the rendered version of the data looks like. We also split the filtering of direct and indirect color which means we do separate filtering on the direct and indirect lighting. The idea to split direct and indirect lighting came from Bauszat et al. [27]. Since details like global illumination and soft shadows come from indirect illumination, we can use this separation to apply different filtering to each. Figure 6.2 shows a close-up of these two buffers.



(a) (b)

Figure 6.2: Direct and indirect illumination

6.2 Implementation detail

We start with the explanation of the data structures we use to store every buffer. Here we will also explain every element of these buffers. After this we discuss how the real-time calculation of the mean and variance is done. Finally we discuss the full algorithm with an explanation of all the parameters and calculations that need to be done.

6.2.1 Data structures

We need to keep in mind when building these data structures that the GPU likes to read/write data in chunks of 2^x Bytes starting at a minimum of 4 Bytes. Instead of one data structure for every scene feature and color, we store all the data in one data structure. We call this data structure the FilterVector and it stores information in the following way:

$$F(x, y) = \{direct, indirect, depth, normal, texture, texture2\}. \tag{6.1}$$

The size of the FilterVector is exactly 64 Bytes which makes it fit perfectly on a global memory cache line on modern GPU's[1]. The cache is used as an intermediate buffer to get global memory to private or local memory. The renderer stores the $direct$ illumination at the first bounce and the $indirect$ illumination for the rest of the bounces. The $depth$, $normal$ and $texture$ color of the first bounce are also stored. Additionally the texture color of the diffuse object that is hit after the first bounce will be stored in $texture2$. By doing this we can store details from diffuse objects that are behind refractive of reflective objects. Figure 6.3 gives an example of how all scene features (GBuffer) are stored for one pixel.

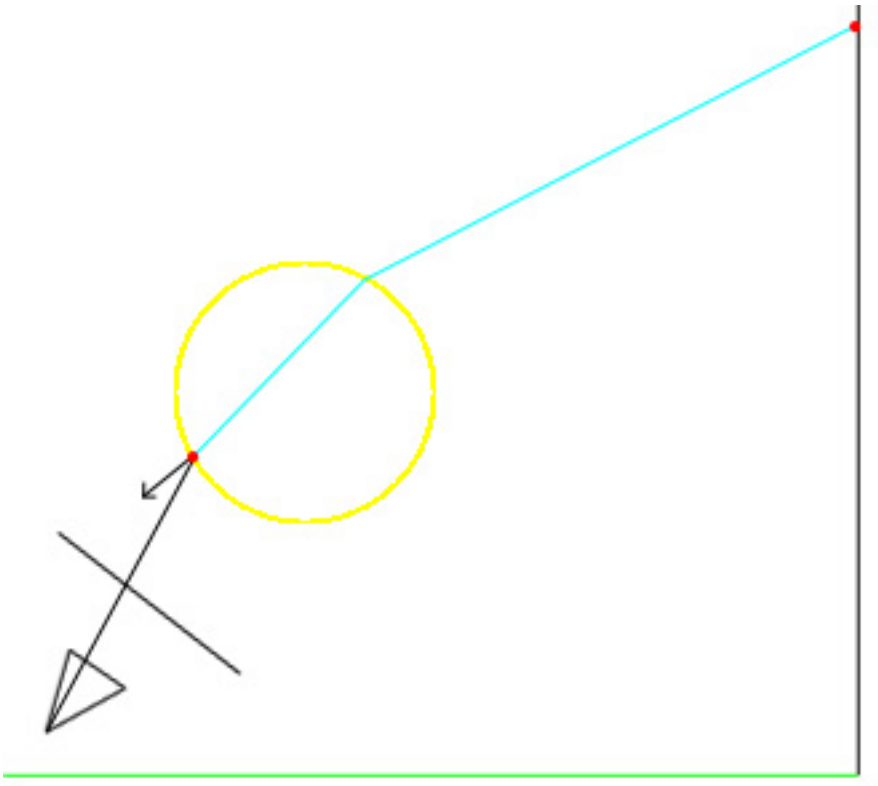


Figure 6.3: At the first bounce (left red dot) the depth, normal and texture color are stored. At the third bounce, the second texture color is stored after the ray has passed through a glass sphere (right red dot).

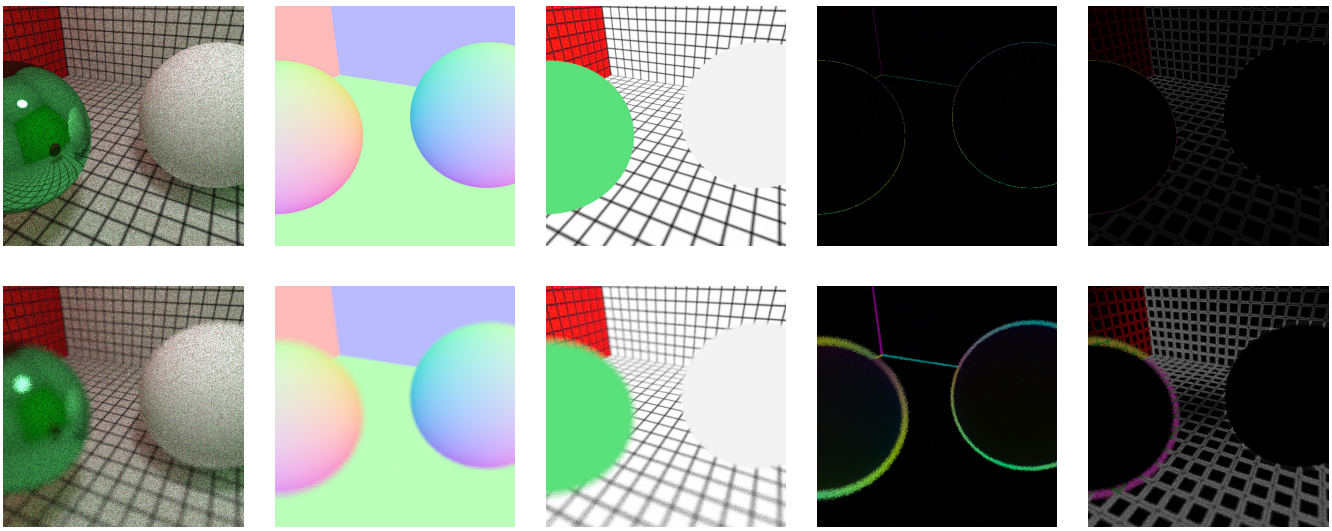
Although the FilterVector does not store the ray direction after the first bounce, it does get stored by the renderer to calculate the variance of the bounce direction. The data structure that stores the variance of the scene features is different from the FilterVector. Additionally it stores

[1] AMD Radeon 7970: Cache size $= 16384$ Bytes and cache line size $= 64$ Bytes.

the first bounce direction and drops the direct and indirect colors. This data structure is simply called the GBuffer since it stores the variance of all the available scene features i.e. geometric features. We use the variance to detect the difference inside the pixel, since we don't store every sample that is generated for the pixel like RPF. Now we only have to store the average over all the samples for each pixel. This data structure is formulated as:

$$GB(x, y) = \{depth, normal, texture, direction, texture2\}. \tag{6.2}$$

Each pixel has its own variance buffer. This variance can be used to determine how much of the pixel needs to be filtered. If the variance is high, the pixel needs more filtering and vice-versa. Figure 6.4 shows one scene from one camera viewpoint rendered without and with depth-of-field. The normal and texture variance around the edges of the geometry is higher when depth of field is used. Especially for parts of the scene that are out-of-focus, the texture variance is higher.



(a) input (b) normals (c) texture (d) normal variance (e) texture variance

Figure 6.4: Example of normal and texture variance buffers. Top row is rendered without depth-of-field and bottom row with depth-of-field

The renders clearly show more variance on geometric edges in the normal and texture buffers. The main reason is that the sphere on the left is out-of-focus and the sphere on the right is less out-of-focus making the variance on the edge lower. Figure 6.5 shows why the normal and texture variance for one pixel dramatically increase for objects out of focus. The random position inside the pixel as well as the random position on the lens (with depth-of-field), affect how much normal and texture variance there is. Highly detailed geometry with an irregular surface can also affect the variance of the normals.

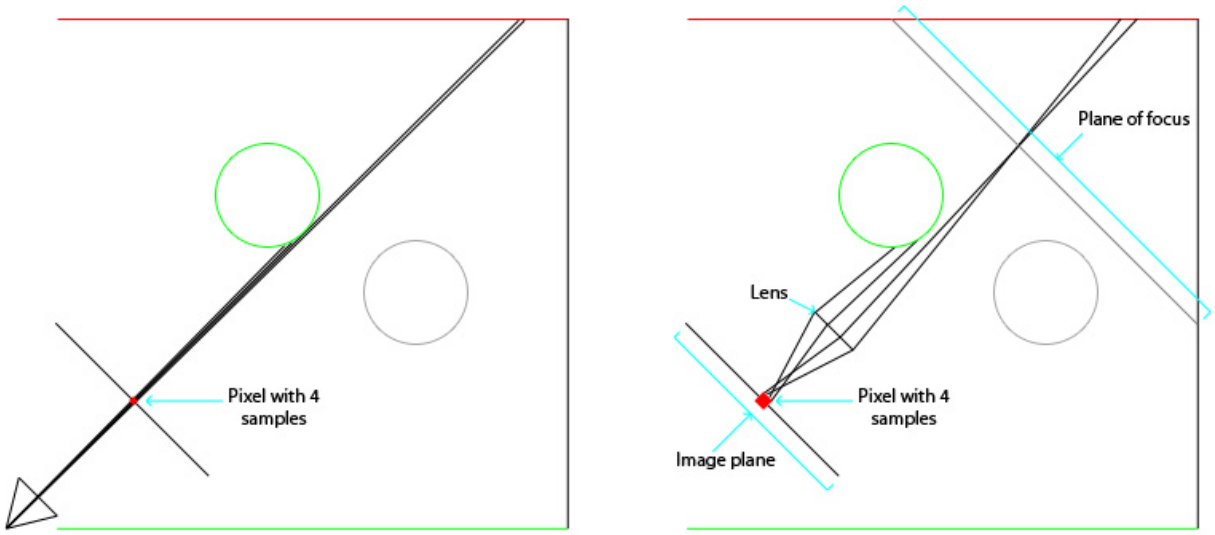
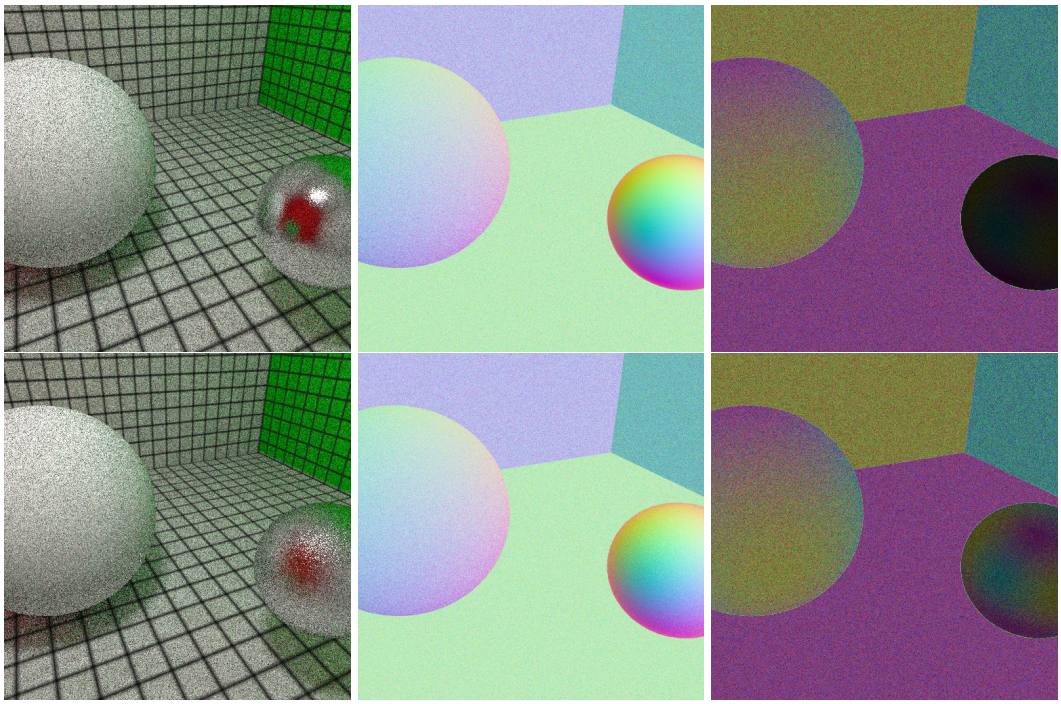


Figure 6.5: Normal and texture variance caused by pixel sampling and/or depth-of-field. The left image shows rays being shot into the scene without and right with depth-of-field

The two images clearly show that with depth-of-field, the normal and texture color will be the average of the sphere and wall in the back. This causes the normal and texture variance to go up. The average depth will be somewhere in between the sphere and the wall causing variance in the depth buffer as well. To give an intuitive example on how the ray direction after the first bounce can be used in the filter, we show an example of a glossy material in figure 6.6



(a) input (b) direction (c) direction variance

Figure 6.6: Example of direction variance. Top row is rendered with near specular object and bottom row with more diffuse object

The variance in the direction can give an estimate of the roughness of the material i.e. glass/mirror material is smooth and diffuse material is rough. Here's a short summary for each buffer that is used as an input for the data driven filter:

Buffer	Type	Stored in	Description
Direct color	μ	FilterVector	The average direct illumination for one pixel with x spp. It is used during filter time as the input to be filtered.
Indirect color	μ	FilterVector	The average indirect illumination for one pixel with x spp. At the end of the rendering process of one frame, direct and indirect illumination are summed up to create the final output.
Depth	μ	FilterVector	The average depth for one pixel with x spp. The depth is one of the scene features used for edge detection. It is especially useful when combined with depth variance to detect noisy out-of-focus edges.
Normal	μ	FilterVector	The average normal for one pixel with x spp. The normal can be used to detect sudden changes in the geometry. These changes can be caused by an edge or highly detailed surface geometry caused by normal maps or something else.
Texture	μ	FilterVector	The average texture color for one pixel with x spp. This is the texture/material color for the first bounce. This parameter is used to preserve texture and material detail.
Texture2	μ	FilterVector	The average texture color of the second diffuse bounce for one pixel with x spp. This feature is used to preserve edges and texture detail in reflective and refractive materials to prevent over blurring. These edges are present in the indirect color, but this parameter enforces preservation of detail.
Depth	σ^2	GBuffer	The variance in the depth buffer for one pixel with x spp. If there is high variance in this buffer it means the samples for this pixel all hit different geometric surfaces. This can be caused by depth-of-field, motion blur or pixel sampling.
Normal	σ^2	GBuffer	The variance in the normal buffer for one pixel with x spp. High variance in the normal buffer can mean it is highly detailed geometry or the pixel is out-of-focus in which cause it needs filtering.
Texture	σ^2	GBuffer	The variance in the texture buffer for one pixel with x spp. High variance here means a highly detailed textured surface is detected or the pixel is out-of-focus.
Direction	σ^2	GBuffer	The variance in the direction buffer for one pixel with x spp. High variance in the bounce direction says something about the material properties.
Texture2	σ^2	GBuffer	The variance in the texture color buffer for the second diffuse bounce for one pixel with x spp.

Table 6.1: Mean and variance buffers

6.2.2 Calculating mean and variance in real-time

The standard way to calculate the sample mean for x samples is:

$$\hat{\mu}_n = \frac{1}{n}\sum_{k=1}^{n} x_k, \tag{6.3}$$

where $\hat{\mu}_n$ is the unbiased estimator for the population mean μ and n is the number of samples [28]. To find an unbiased estimate for the population variance we use $\hat{\mu}_n$ in the following equation [29]:

$$\hat{\sigma}^2_{N-1} = \frac{1}{N-1}\sum_{i=1}^{N}(x_i - \hat{\mu}_n)^2. \tag{6.4}$$

We do not store all the samples in order to calculate $\hat{\mu}$ and $\hat{\sigma}^2$, so we need a way to calculate the new mean and variance every time we get a new sample. To do this we apply a method proposed by Knuth [30], to calculate the running mean and variance in real-time. We use the following recursion equations to calculate the sample mean and variance:

$$\mu_n = \mu_{n-1} + \frac{(x_n - \mu_{n-1})}{n} \tag{6.5}$$

$$S_n = \begin{cases} S_{n-1} + (x_n - \mu_{n-1})(x_n - \mu_n) & \text{if } n > 1 \\ 0 \end{cases}, \tag{6.6}$$

where n is the number of samples, μ_{n-1} is the old mean, x_n is the new sample, μ_n is the new mean and the n^{th} estimate of the variance is: $\hat{\sigma}^2 = \frac{S_n}{n-1}$. The last part where we actually obtain the sample variance is only calculated when we want to use the variance in the filter. Equations 6.5 and 6.6 are implemented in the renderer and do not affect the filter time.

6.2.3 Filter algorithm

Each pixel goes through the same filtering process. Table 6.2 gives an overview of all user adjustable parameters in the filter. These parameters are discussed in more detail in the next chapter where they will be used for experimentation. Table 6.3 gives an overview of all the other parameters including the input of the algorithm. These parameters are calculated at run-time and can depend on the user adjustable parameters. The inputs of the algorithm do not change while the pixels are being filtered. The outputs are the filtered pixels which are displayed by the renderer. It is possible to extend the filter to have more output pixelbuffers e.g. to visualize the calculated weights.

Name	Parameter	Description	range
kernel width	kw	Determines the width and height of the pixel neighborhood which is used to filter the pixel. Larger neighborhoods results in smoother filtering, but causes the filter to slow down.	$kw > 0$ and $kw =$ odd integer
direct color sigma	σ_{dc}	This parameter controls how important the direct color is.When a scene is rendered where the direct illumination has low variance, this term can be lowered to preserve as much of the original unfiltered direct color as possible.	$\sigma_{dc} > 0$
indirect color sigma	σ_{ic}	See direct color sigma description.	$\sigma_{ic} > 0$
depth sigma, normal sigma, texture sigma, direction sigma, texture2 sigma	σ_f	σ_f is a vector and is used as a scaling term for the scene features. When a component is low, the importance of that scene feature will be high.	$\sigma_{f,i} > 0$
scene feature variance scales	s_f	These scales are used as additional weights to the variance that was calculated by the renderer. A higher scale will cause heavier filtering for the i-th scene feature. Setting the scale to 0 will cause the filter to ignore the variance in the i-th scene feature, which can lead to noisy output.	$s_{f,i} \geq 0$

Table 6.2: Dynamic parameters

Name	Parameter	Description
FilterVector current /neighboring pixel (input)	F, F_N	The FilterVector for the current pixel and neighboring pixel N.
variance GBuffer (input)	GB	This vector stores the variance for every scene feature in this pixel (equation 6.2).
Neighborhood of pixels	N	The number of pixels in N is calculated by $N_{size} = kw^2$
filtered direct and indirect color	C_{fd}, C_{fi}	Filtered direct and indirect color of the current pixel
direct color variance	σ_{dc}^2	Direct illumination scaling term, where σ_{dc} is used to dynamically adjust it.
indirect color variance	σ_{ic}^2	Indirect illumination scaling term, where σ_{ic} is used to dynamically adjust it.
scene feature variance	σ_f^2	Scene feature variance vector, where $\sigma_{f,i}$ and the variance GB are used to determine the variance for each scene feature for the pixel being filtered.
local direct weight	w_d	Direct weight calculated for one neighboring pixel ranging from 0 to 1.
local indirect weight	w_i	Indirect weight calculated for one neighboring pixel ranging from 0 to 1.
total direct weight	tw_d	Total direct color weight accumulated from neighboring pixels with a range from 0 to N, where N is the size of the neighborhood.
total indirect weight	tw_i	Total indirect color weight accumulated from neighboring pixels with a range from 0 to N, where N is the size of the neighborhood.

Table 6.3: Static parameters

Although the filter is based on the cross bilateral filter, it is missing one key component which is the spatial term. We remove this term because it can cause severe artifacts on filtered uniform surfaces with one material color and no textures. By removing this term we try to prevent these artifacts as much as possible. As a reminder, the spatial term is calculated by the weighted distance between the center pixel and the neighboring pixel using a 2D-Gaussian function:

$$G_s(x, y) = \exp[-\frac{(|p.x - q.x|)^2 + (|p.y - q.y|)^2}{2\sigma^2}]. \tag{6.7}$$

The main algorithm consists of 3 main steps and can be found in appendix A.

- Initialization
- Calculating weights and accumulate neighboring pixel contribution

- Normalization and output

Initialization

First w_d, w_i, tw_d and tw_i are initialized to 0. Next we calculate σ_{dc}^2 and σ_{ic}^2 using the following equations:

$$\sigma_{dc}^2 = \frac{1}{2 \cdot (\sigma_{dc})^2} \tag{6.8}$$

$$\sigma_{ic}^2 = \frac{1}{2 \cdot (\sigma_{ic})^2} \tag{6.9}$$

In order to calculate the scene feature variance vector we use the input variance buffer GB and the scene feature scales s_f. Here we show how to calculate the scene feature variance for one component i i.e. $depth$, $normal$, $texture$, $direction$ or $texture2$:

$$\sigma_{f,i}^2 = \frac{1}{2 \cdot (\sigma_{f,i})^2} \cdot (1 - Clamp_{(0,1)}(\sum_{k=1}^{m}(GB_{i,k}) \cdot s_{f,i})), \tag{6.10}$$

where $\sum_{k=1}^{m}(GB_{i,k})$ sums the variance for every component for that scene feature e.g. the normal has 3 components (x, y, z) which will be summed to one number and multiplied by a scaling factor for that scene feature ($s_{f,i}$). We clamp the output between 0 and 1 to ensure the outcome never becomes negative. If the variance in the scene feature is high, $\sigma_{f,i}^2$ will be low or even 0. If the scene feature has 0 variance ($\sum_{k=1}^{m}(GB_{i,k}) = 0$), then the variance is determined by the user determined scaling parameter $\sigma_{f,i}$.

Calculating weights and accumulate neighboring pixel contribution

This part of the algorithm is done for each pixel in N. Since we split the color into direct and indirect, we also keep track of separate weights for both. The following equations are used to calculate w_d and w_i, where some temporary local variables are used to store separate weights for direct color (w_{dc}), indirect color (w_{ic}), direct scene features (w_{fd}) and indirect scene features (w_{fi}). The larger the distance between two colors, normals, etc., the lower the weight will be. First the local weights for the direct and indirect color are calculated which is often referred to as the range weight, which represent the difference in color between current and neighboring pixels.

$$w_{dc} = \exp[-\sigma_{dc}^2 \cdot \sum_{i=0}^{2}(|F_{0,i} - F_{N0,i}|^2)] \tag{6.11}$$

$$w_{ic} = \exp[-\sigma_{ic}^2 \cdot \sum_{i=0}^{2}(|F_{1,i} - F_{N1,i}|^2)] \tag{6.12}$$

Secondly, the scene feature weight for direct and indirect are calculated separately. The $depth$ ($F_{2,i}$), $normal$ ($F_{3,i}$) and $texture$ ($F_{4,i}$) are used to calculate the weights for both direct and indirect. The bounce direction variance $\sigma_{f,3}^2$ and $texture2$ ($F_{5,i}$) color difference and variance $\sigma_{f,4}^2$ are only used when the indirect weight is calculated. Since the direct color buffer does not

capture any information about multiple bounce, it is useless to filter it with additional multiple bounce information like direction and additional texture information.

$$\begin{aligned} w_{fd} &= \exp[-\sigma_{f,0}^2 \cdot |F_{2,i} - F_{N2,i}|^2] \cdot \\ &\quad \exp[-\sigma_{f,1}^2 \cdot \sum_{i=0}^{2}(|F_{3,i} - F_{N3,i}|^2)] \cdot \\ &\quad \exp[-\sigma_{f,2}^2 \cdot \sum_{i=0}^{2}(|F_{4,i} - F_{N4,i}|^2)] \end{aligned} \tag{6.13}$$

$$w_{fi} = w_{fd} \cdot \exp[-\sigma_{f,4}^2 \cdot \sum_{i=0}^{n-1}(|F_{5,i} - F_{N5,i}|^2)] \cdot \exp[-\sigma_{f,3}^2], \tag{6.14}$$

where $F_{x,i}$ and $F_{Nx,i}$ are the x-th components of the FilterVector for this pixel and neighboring pixel respectively (equation 6.1). Now the total weight which includes color and feature weights need to be computed before we can add the contribution of the neighboring pixel to the filtered color. We do this by multiplying range weights with the feature weights:

$$w_d = w_{dc} \cdot w_{fd} \tag{6.15}$$

$$w_i = w_{ic} \cdot w_{fi}. \tag{6.16}$$

With our final weights we can add the contribution for the neighboring pixel to the final filtered pixel with the following two equations:

$$C_{fd} = C_{fd} + w_d \cdot F_{N0} \tag{6.17}$$

$$C_{fi} = C_{fi} + w_i \cdot F_{N1} \tag{6.18}$$

Finally we need to calculate the total weight which is used to normalize the color after we are done looping over the entire neighborhood. This is a standard process also done in bilateral and cross/joint bilateral filters.

$$tw_d = tw_d + w_d \tag{6.19}$$

$$tw_i = tw_i + w_i \tag{6.20}$$

Normalization and output

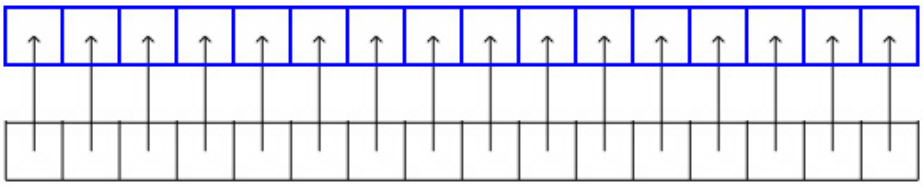
After the filtered direct and indirect colors over the whole neighborhood, we normalize it with the total weight and calculate the final filtered pixel color with the following equation:

$$C_f = \frac{C_{fd}}{tw_d} + \frac{C_{fi}}{tw_i} \tag{6.21}$$

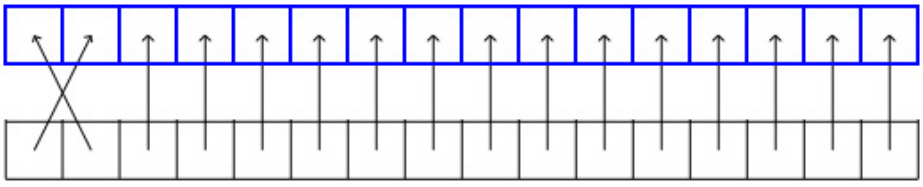
6.3 Optimizations

There are several considerations to be made when implementing an algorithm on the GPU which have to do with the GPU architecture. Modern GPU's use SIMT architecture, where multiple threads in a group execute in parallel using one set of instructions. For this reason it is important for all threads to follow a similar code-path. According to the OpenCL specification

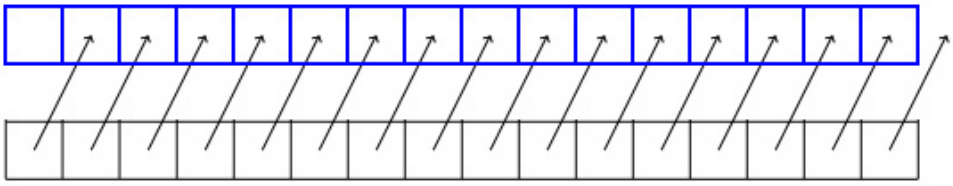
[31] it is important for any algorithm to access memory coalesced and aligned to get optimal performance. When memory access is not coalesced, OpenCL will serialize memory access for all work-items in a work-group. Figure 6.7 depicts coalesced, not coalesced and misaligned memory access patterns. Since global memory access is relatively slow, we want to reduce the number of memory transfers from global to local / private memory.



(a) Coalesced memory access causing 1 transfer of 64 Bytes



(b) Non coalesced memory access causing 16 transfers of 4 Bytes



(c) Misaligned memory access causing 16 transfers of 4 Bytes

Figure 6.7: Each blue cell is 4 Bytes and each black cell is one work-item

Random memory access patterns are bad. There are three main constraints for a GPU implementation of the algorithm to make optimal use of GPU hardware:

- Maximize number of threads used.
  To utilize as many resources as possible we want to keep every work-item busy at all times.
- Optimize memory usage and access.
  Make use of fast local memory when possible, always try to get coalesced memory access patterns and keep the memory consumption low.
- Optimize program flow.
  Always try to minimize code branching to avoid work-items from doing nothing.

A naive implementation of this filter can give bad results in terms of speed and consistency with resolution changes. The filter will work without the optimizations, but will be slow due to the way GPU architectures work. There are three main optimizations we did to increase the speed and reliability of the filter. The first set of optimizations are focused on the use of build in data structures that are already optimized to the hardware (6.3.1). In the second part we will talk about the use of local memory to reduce the amount of memory traffic from/to global

memory. In the last part I've applied a trick to speed up the filter by $3\times$. There is a special case where this filter failed which is handled separately (6.3.4). A quantitative evaluation of the speed and quality of the filter will be given in the next chapter.

6.3.1 Using OpenCL built-in vector data structures

Every main data type like int, float and double have vectorized data types to i.e. *intn, floatn and doublen*. When these types are used, the compiler and memory manager will automatically align each vector to a cache-line in the cache when transferring memory from/to global memory.

To ensure that a custom data structure like the FilterVector F and GBuffer GB align with the cache, we cast them to a vector type that fits the data structure. The largest vector can carry 16 components, so if we want to get F from global memory in one go, we need to fit it in 64 Bytes. In section 6.2.1 we already discussed that the size of F is exactly 64 Bytes. However, GB is only 52 Bytes so the closest fit is a float16. This is no big deal since we only need to load GB once for each pixel. Nevertheless we want each work-item in a work-group to coalesce its memory access to prevent the situations depicted in figure 6.7(b, c).

It is also useful to use as much read only private memory where possible since it is faster than read/write memory.

6.3.2 Global vs Local memory

As mentioned before, global memory is slower than local memory. Figure 6.8 depicts the brute force implementation using only global memory. It shows one pixel loading a neighbor every time it needs one, meaning a lot of redundant memory access. The bottleneck lies in the part of the algorithm where we loop over the neighborhood of pixels. Each neighborhood Filter-Vector F_N needs to be downloaded from global memory to private memory. If the size of the neighborhood is 25×25 pixels we need to do 625 global memory loads for every pixel in the screen.

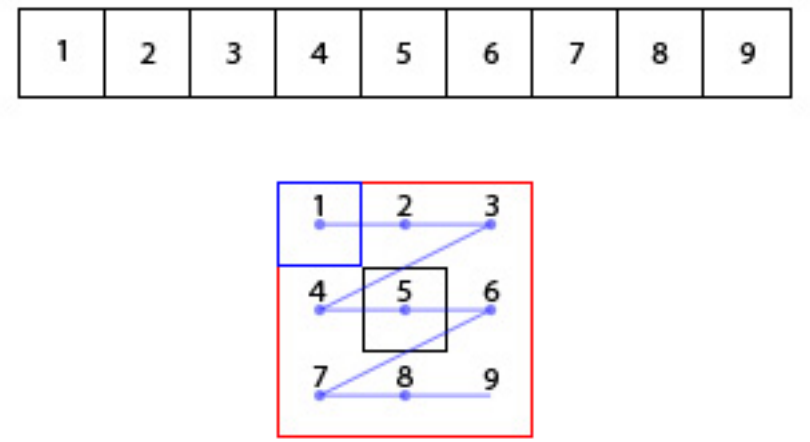


Figure 6.8: Loading F_N every iteration over the neighborhood with a total of 9 loads. The black center pixel is being filtered and the blue pixel needs to be loaded from global memory.

Important to know is that modern GPU's have around 32 kB of local memory for every work-group of work-items. work-items within one work-group can share data across local

memory. We would like to create a situation where we only need to download one big chunk of memory that contains all F_N's for every pixel in the neighborhood before looping over the neighborhood N. This way we only have to do global memory loads one time. Since all work-items in the work-group share the local memory it is useful to load every neighbor for every work-item. Let's say the work-group size is 4 and the size of N is 3×3. Each work-item has to load a chunk of memory at the same time so the memory manager can handle it as one memory load. This setup is depicted in figure 6.9.

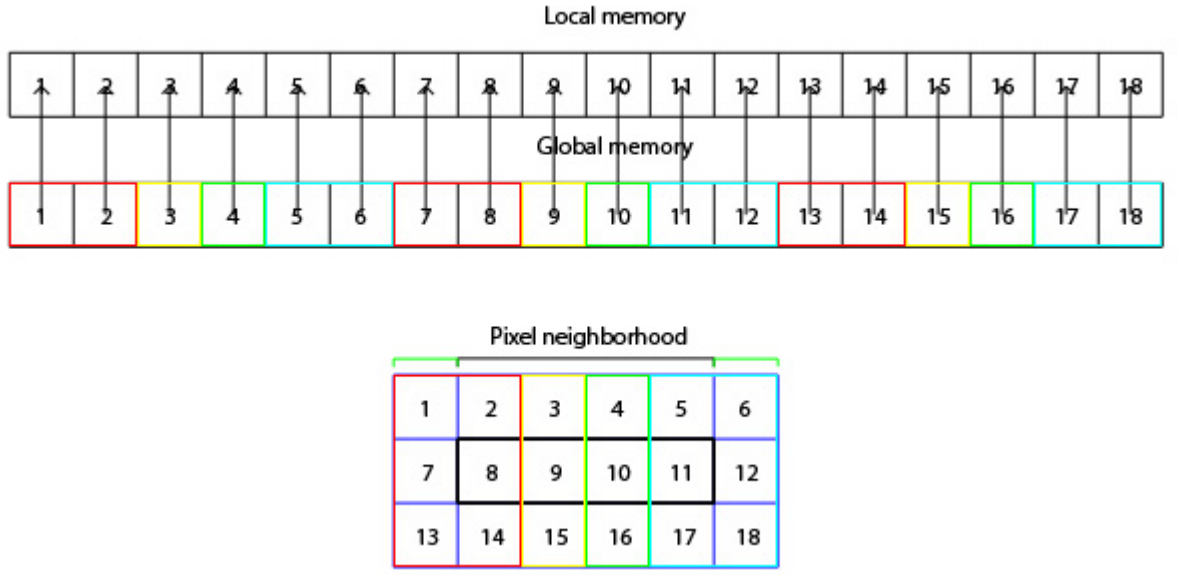


Figure 6.9: Each pixel (work-item) loads its own part from global memory. The first and last pixel (8 and 11) have to do extra loads because of the borders. The middle pixels (9 and 10) load a column.

There are three main problems with this approach. The goal was to have one memory load, which does not happen in this cause because of the non-coalesced memory access pattern for each work-item, since some work-items have to load more neighbors. Even if we can somehow get a coalesced pattern we would still have to wait for the two outside pixels to finish loading neighbors, before the work-items continue. We want all work-items to do an equal amount of work to prevent one work-item from stalling all the other work-items in the work-group. The last problem is the amount of memory that is needed to store every neighbor in local memory. In the example we only need $18 \times 64 = 1152$ Bytes, but when we have a normal sized work-group of 64 with a neighborhood size of 25×25 we need $((64 + 12 + 12) \times 25) \times 64 = 140.800$ Bytes. We only had 32.768 Bytes so we have a shortage of local memory.

Instead of loading every neighbor into local memory, we only store the neighbors of the row that's being processed. This setup is depicted in figure 6.10.

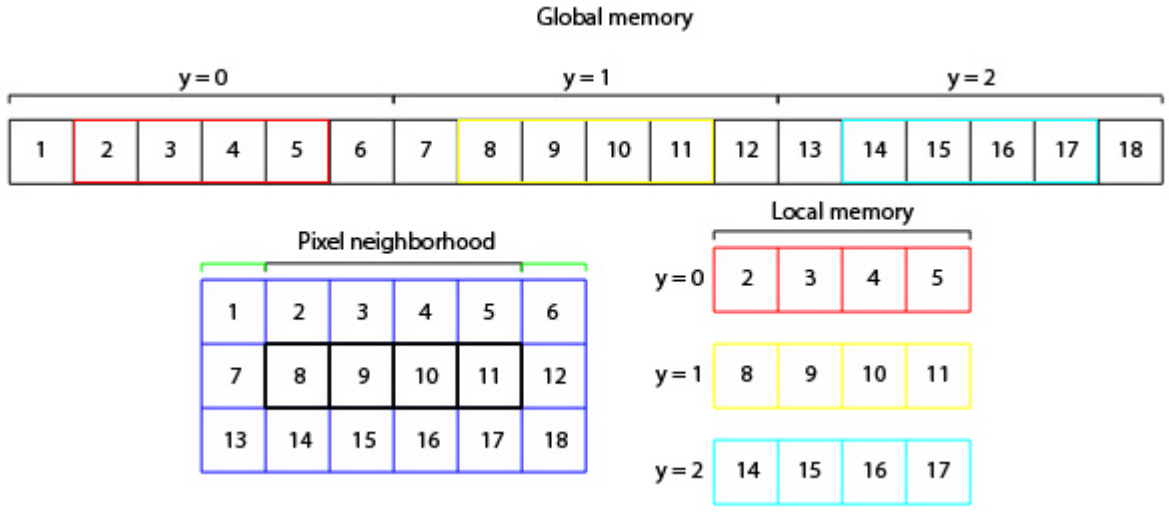
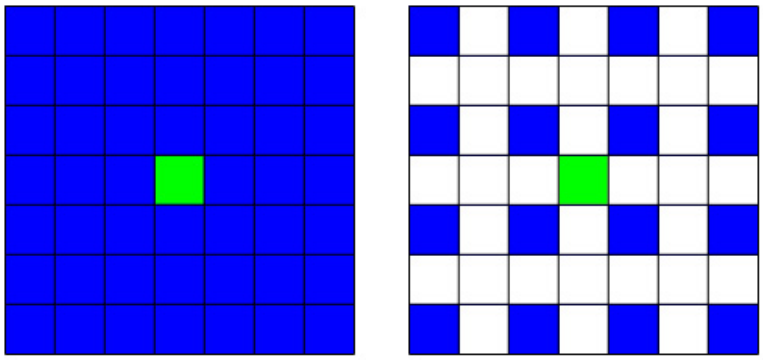


Figure 6.10: Each pixel (work-item) will load the neighbor in its own column (x) at position y into local memory. When all work-items continue to the next row (y is increased), a new row is loaded from global to local memory.

When a pixel wants to load a neighbor that falls outside the range that is loaded into local memory (1, 7, 13 or 6, 12, 18), it loads the neighbor from global memory instead. Now all loads are coalesced and don't take up as much local memory. If we take the same 64 large work-group and 25×25 neighborhood size, we end up with: $64 \times 64 = 4096$ Bytes. With this scheme the amount of local memory needed only depends on the size of the work-group. The maximum work-group size on modern GPU's is 256 which puts us on a maximum of $64 \times 256 = 16.384$ Bytes. When the neighborhood size increases, the amount of global memory loads will increase, but the local memory usage will stay the same.

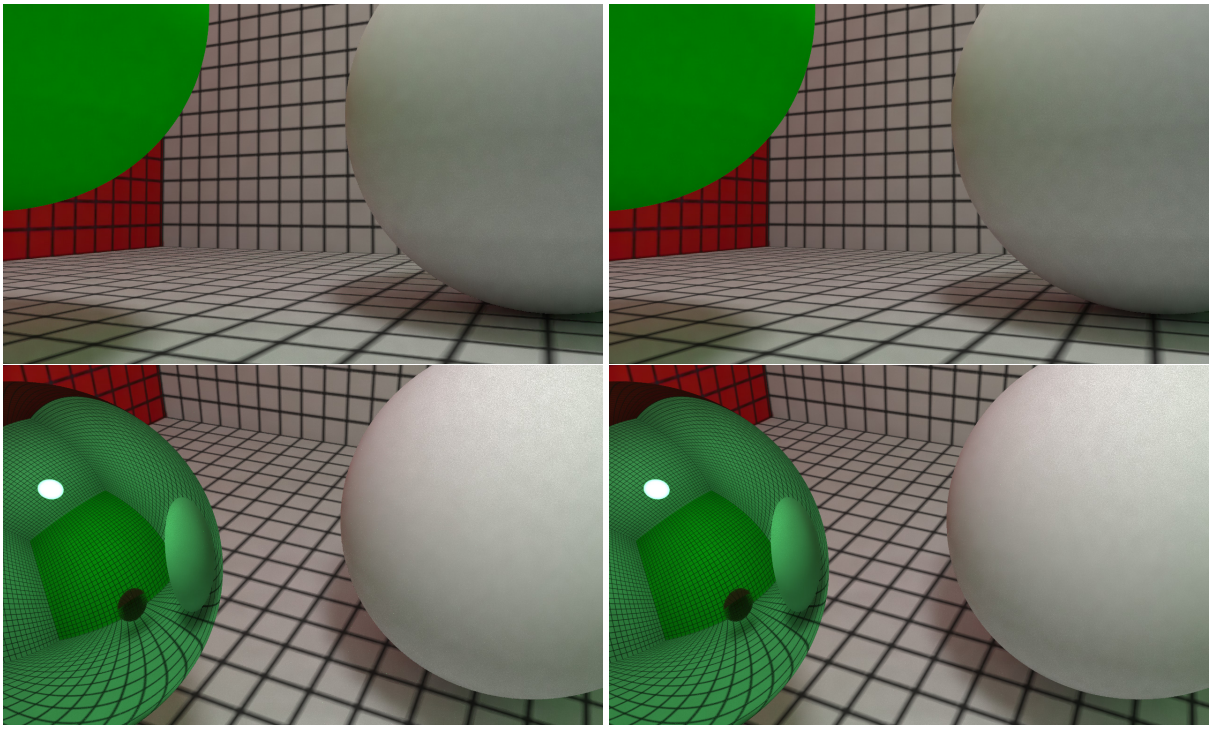
6.3.3 With holes

Instead of using all the neighbors in the neighborhood N we can skip some of them to gain some speed. A speed increase of $\approx 3\times$ is gained by skipping half of the neighbors in width and height which is depicted in figure 6.11. Figure 6.12 shows the difference with and without holes.



(a) (b)

Figure 6.11: Each blue cell is a neighboring pixel being used for filtering

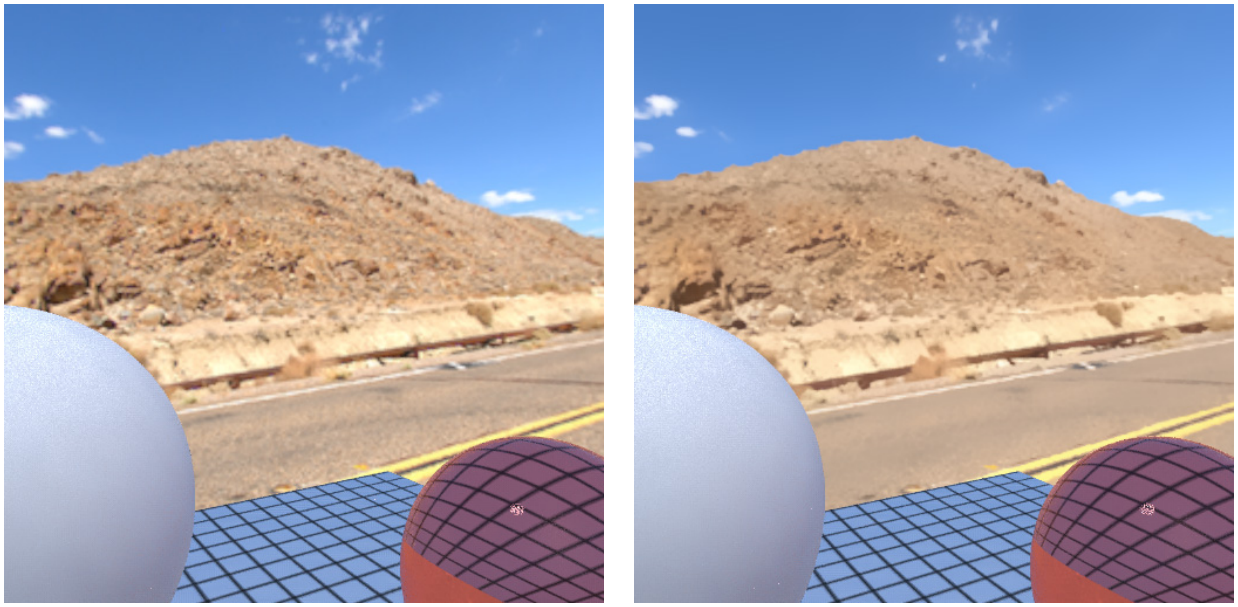


(a) With holes filtered render with 16 spp (b) Without holes filtered render with 16 spp

Figure 6.12: Filtered renderer output with and without holes

6.3.4 Special case

When rendering a scene with a static sky-box, the filter tends to over blur the detail present in the sky-box. Although color differences are used to determine neighboring pixel weights, the scene features also affect the weight (equation 6.17 and 6.18). Since the sky-box lies at infinity and has no real normals, these scene features will be the same for the filtered pixel and neighboring pixel causing the scene feature weights w_{fd} and w_{fi} to become close to one (equation 6.13 and 6.14). To take this into account the depth is checked and if it lies at infinity, the filtering is skipped and the original pixel color is used. Now over blurring is prevented. A render of this special case is depicted in figure 6.13.



(a) With sky-box filtering taken into account

(b) Without sky-box filtering taken into account

Figure 6.13

Part III

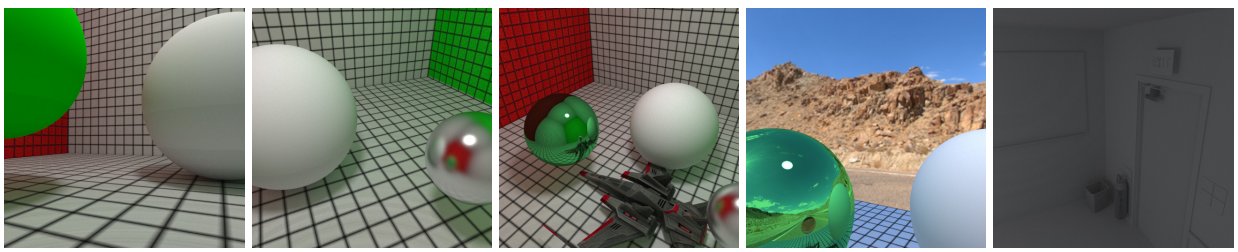
Results

Chapter 7

Evaluation and Results

In this chapter we compare our optimized GPU implementation of the data driven filter with RPF. First we will focus on the speed of the filter. Then we will do experimentation to find optimal filtering parameters for the data driven filter. After this we show the comparisons between the new method and RPF. All comparisons against RPF are done with five scenes were the ground truth for each scene is rendered with 2048 spp at a resolution of 512×512 pixels (figure 7.1).

- Cornell diffuse scene. Objects in the scene use diffuse only materials where some of them are textured. This scene has low complexity in terms of geometry and materials. Texture detail needs to be preserved by the filter.
- Cornell glossy scene. Objects in the scene use diffuse, glossy and specular materials where some of them are textured. This scene has low geometry complexity and medium/high material complexity. Reflection and texture detail needs to be preserved in the filter.
- Cornell plane scene. Objects in the scene use diffuse, refractive and specular materials where some of them are textured. This scene has medium geometry complexity and medium/high material complexity.
- Outside scene. Objects in the scene use diffuse, refractive and specular materials where some of them are textured. An environment light (sky-box) is used to light the scene. This scene has medium geometry complexity and medium/high material complexity. The filter should filter the sky-box correctly.
- Conference room. Objects in the scene use diffuse materials without textures. This scene has high geometry complexity and low material complexity. There is no edge information in the $texture$ and $texture2$ buffers, because of the lack of materials/textures.



(a) Cornell diffuse (b) Cornell glossy (c) Cornell plane (d) Outside scene (e) Conference room

Figure 7.1

We first discuss the conditions of the experiments and why these conditions are valid for the filter. After this, experiments on speed and quality are done in which we try to find parameters that give the best balance between the filters quality and speed. Finally we compare the results with RPF. For this quality comparison we use MSE as discussed in section 5.2 to grade the quality of the filter.

7.1 Conditions

We use the same system setup as in section 5.2. The speed and quality experiments are all tested under different constraints:

- All speed experiments are done with filter sizes: 7×7, 15×15, 25×25, 55×55
- All speed experiments are done with resolutions: 320×240, 512×512, 640×480, 1280×768
- All speed measurements are shown in milliseconds (ms)
- All quality comparisons between the data driven filter and RPF are done at a resolution of 512×512. The data driven filter will use the optimal parameter found through experimentation
- RPF filters with 8 spp and the data driven filter with 8, 16 and 32 spp

The MSE measures will be displayed in graphs to show the difference between the ground truth and filtered result.

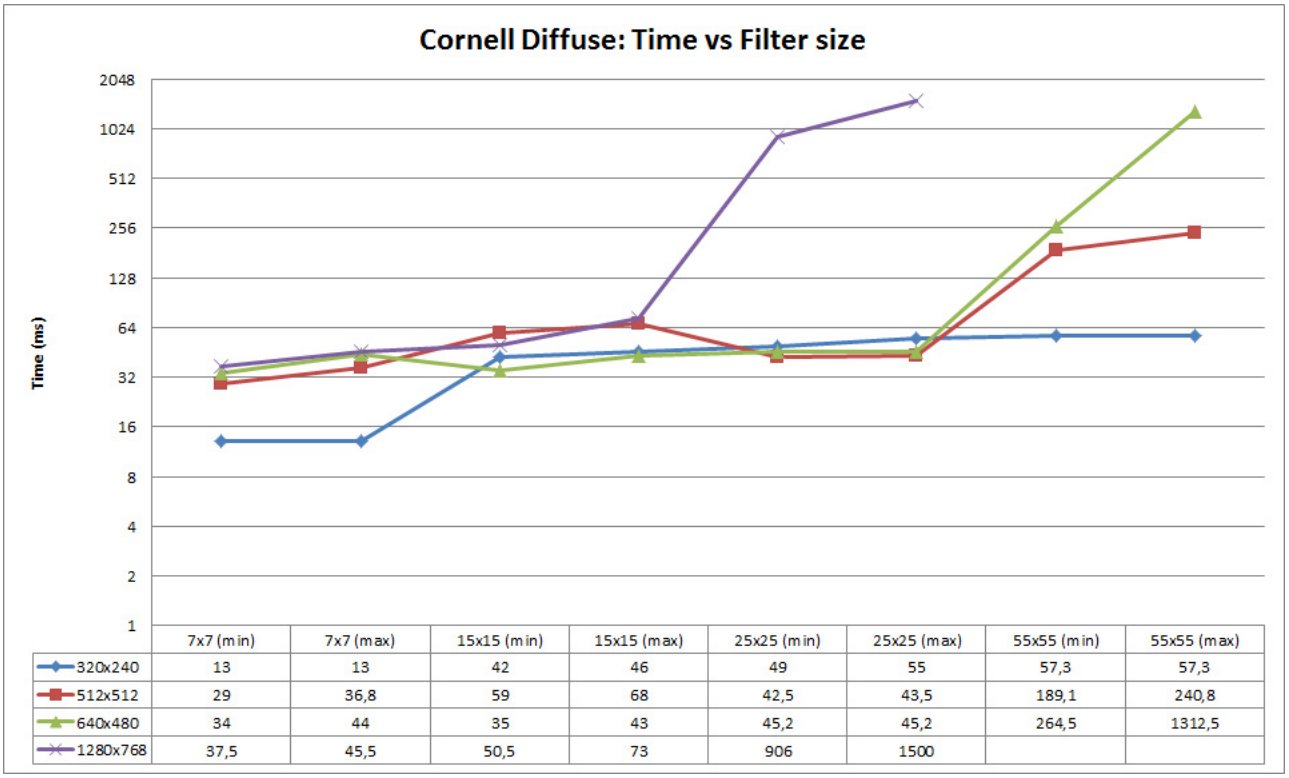
7.2 Speed results

When we do path tracing in real-time, the filter needs to have a predictable run-time i.e. we want the filter to have a lower bound and upper bound running time that are close to each other. We will first show the running times of the un-optimized filter and then show the results for the optimized filter.

The speed is measured in the renderer using the OpenCL event system to time the performance of the kernel. We use the profiling information given through the event system to measure the running time. The lowest and highest running time of the filter over a time period of 10 seconds are stored. These timings can be displayed and/or stored to provide the speed results we need.

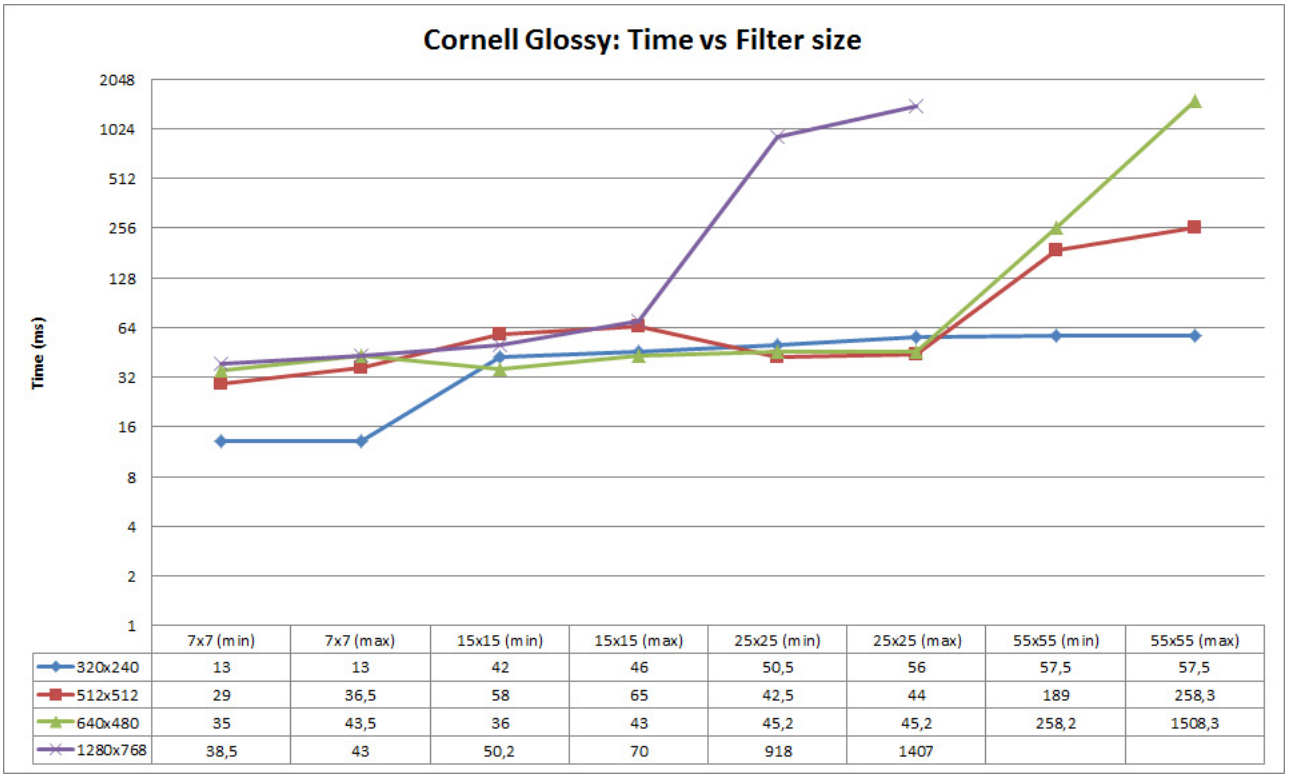
7.2.1 Un-optimized filter

The un-optimized version of the filter is very unstable in terms of filtering time. In these experiments we show the un-optimized filter times versus the filter size. Each graph shows the lower bound and upper bound running times of the filter. The y-axis is scaled logarithmically. Resolutions of 1280×768 with a filter size of 55×55 where not measurable since the program crashed each time. The reason for these crashes are the running time of the filter causing the GPU to crash, which is a built in safety protocol. The GPU hardware has a built-in timer that stops the program after a while (depends on GPU Vendor) to prevent dead-locks.



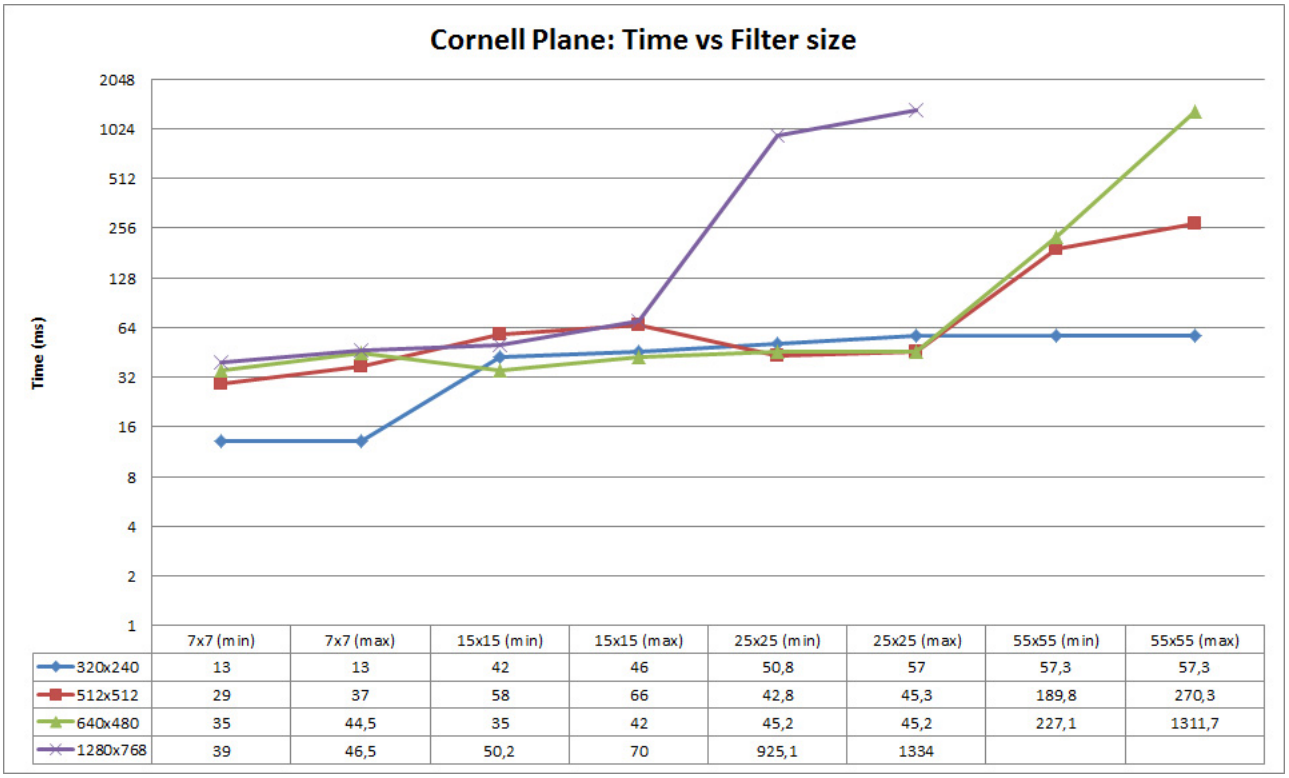
	7x7 (min)	7x7 (max)	15x15 (min)	15x15 (max)	25x25 (min)	25x25 (max)	55x55 (min)	55x55 (max)
320x240	13	13	42	46	49	55	57,3	57,3
512x512	29	36,8	59	68	42,5	43,5	189,1	240,8
640x480	34	44	35	43	45,2	45,2	264,5	1312,5
1280x768	37,5	45,5	50,5	73	906	1500		

Figure 7.2: Graph of the un-optimized filtered Cornell Diffuse scene



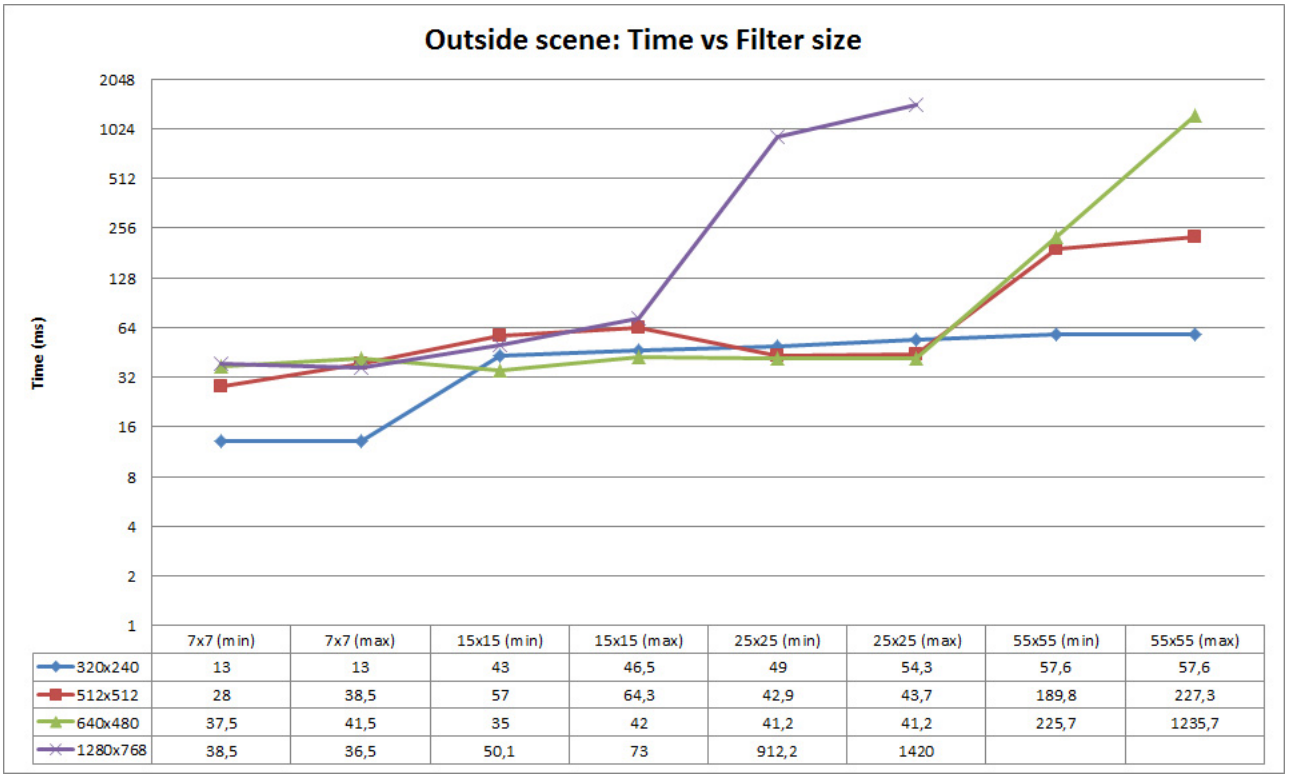
	7x7 (min)	7x7 (max)	15x15 (min)	15x15 (max)	25x25 (min)	25x25 (max)	55x55 (min)	55x55 (max)
320x240	13	13	42	46	50,5	56	57,5	57,5
512x512	29	36,5	58	65	42,5	44	189	258,3
640x480	35	43,5	36	43	45,2	45,2	258,2	1508,3
1280x768	38,5	43	50,2	70	918	1407		

Figure 7.3: Graph of the unoptimized filtered Cornell Glossy scene



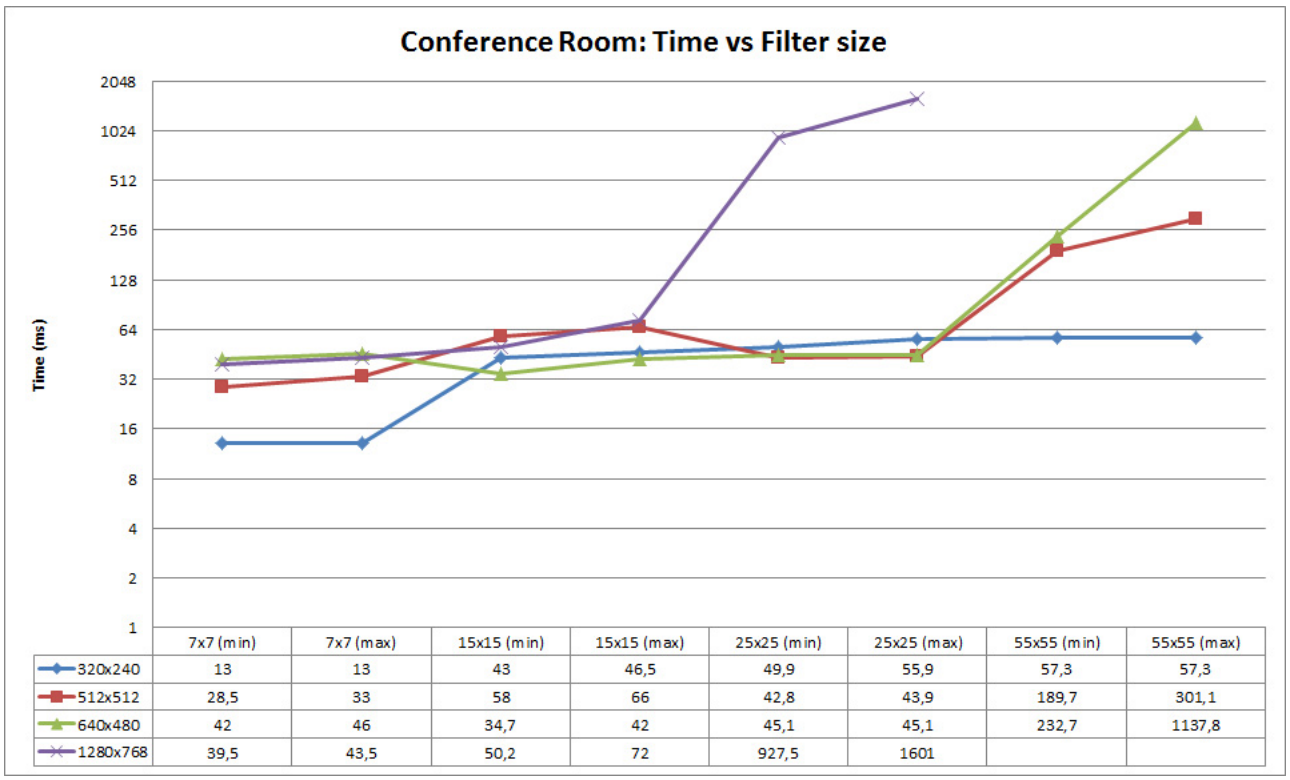
	7x7 (min)	7x7 (max)	15x15 (min)	15x15 (max)	25x25 (min)	25x25 (max)	55x55 (min)	55x55 (max)
320x240	13	13	42	46	50,8	57	57,3	57,3
512x512	29	37	58	66	42,8	45,3	189,8	270,3
640x480	35	44,5	35	42	45,2	45,2	227,1	1311,7
1280x768	39	46,5	50,2	70	925,1	1334		

Figure 7.4: Graph of the unoptimized filtered Cornell Plane scene



	7x7 (min)	7x7 (max)	15x15 (min)	15x15 (max)	25x25 (min)	25x25 (max)	55x55 (min)	55x55 (max)
320x240	13	13	43	46,5	49	54,3	57,6	57,6
512x512	28	38,5	57	64,3	42,9	43,7	189,8	227,3
640x480	37,5	41,5	35	42	41,2	41,2	225,7	1235,7
1280x768	38,5	36,5	50,1	73	912,2	1420		

Figure 7.5: Graph of the unoptimized filtered Outside scene

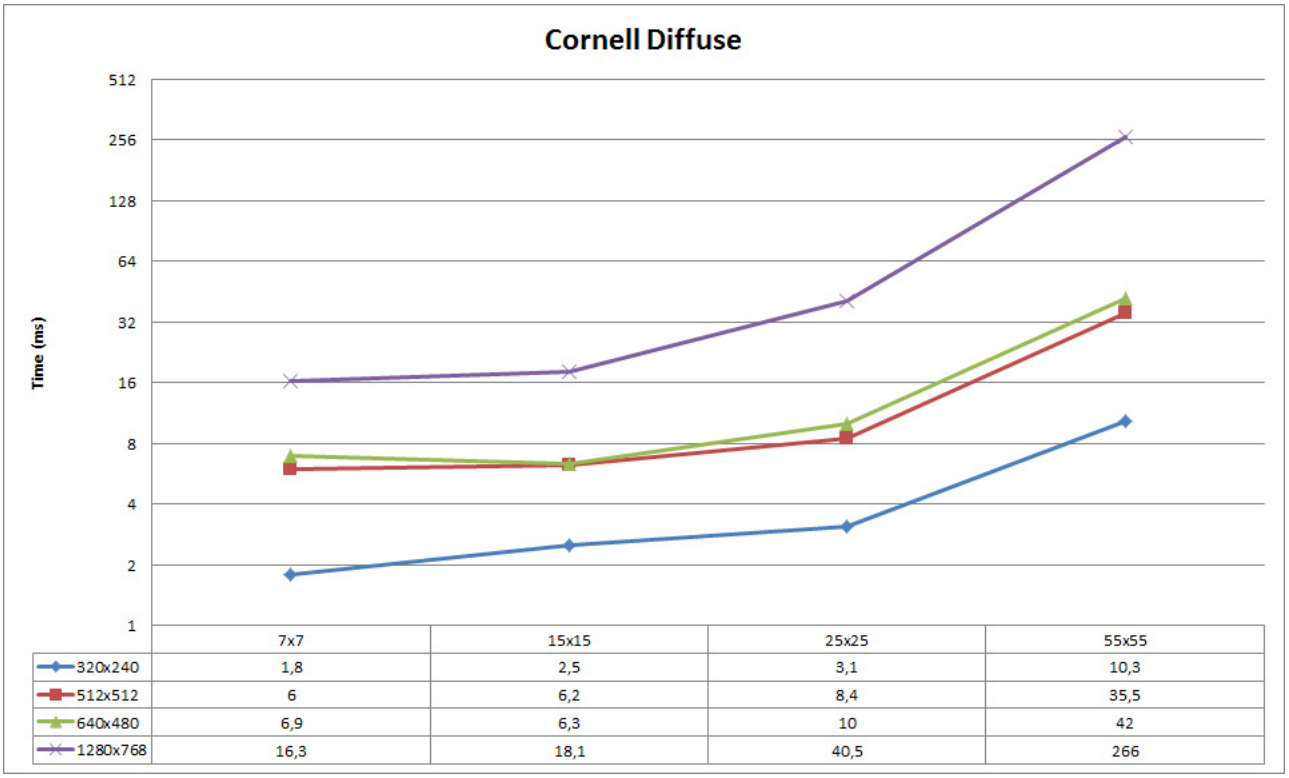


	7x7 (min)	7x7 (max)	15x15 (min)	15x15 (max)	25x25 (min)	25x25 (max)	55x55 (min)	55x55 (max)
320x240	13	13	43	46,5	49,9	55,9	57,3	57,3
512x512	28,5	33	58	66	42,8	43,9	189,7	301,1
640x480	42	46	34,7	42	45,1	45,1	232,7	1137,8
1280x768	39,5	43,5	50,2	72	927,5	1601		

Figure 7.6: Graph of the unoptimized filtered Conference room scene

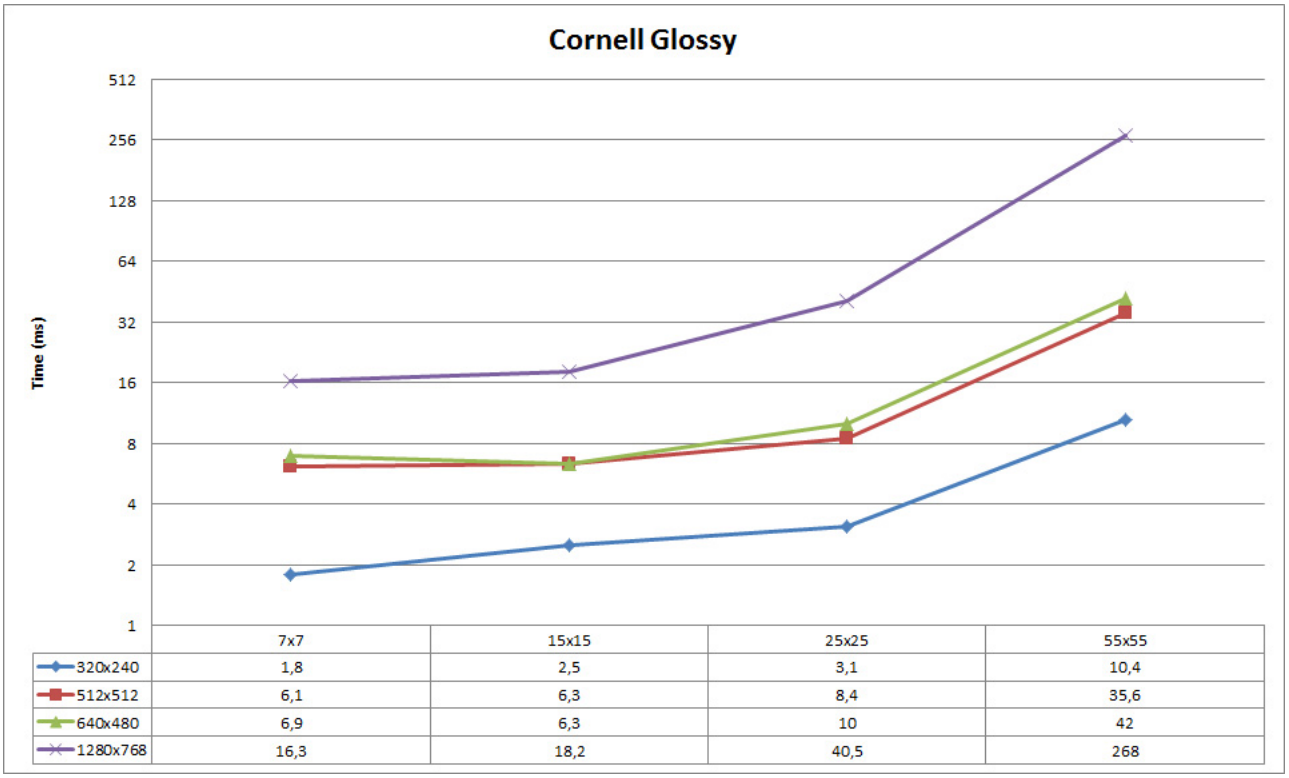
7.2.2 Optimized filter

Each speed test for the filter shows the upper bound time of the filter. The reason for this is that the lower and upper bound time are within $0.5\ ms$ from each other. Figures 7.7, 7.8, 7.9, 7.10 and 7.12 show the filter time versus the filter size for each scene. When the cube map is in full view of the screen the filtering times are constant over all filter sizes which is shown in figure 7.11. This happens because it is handled as a special case as described in the previous chapter.



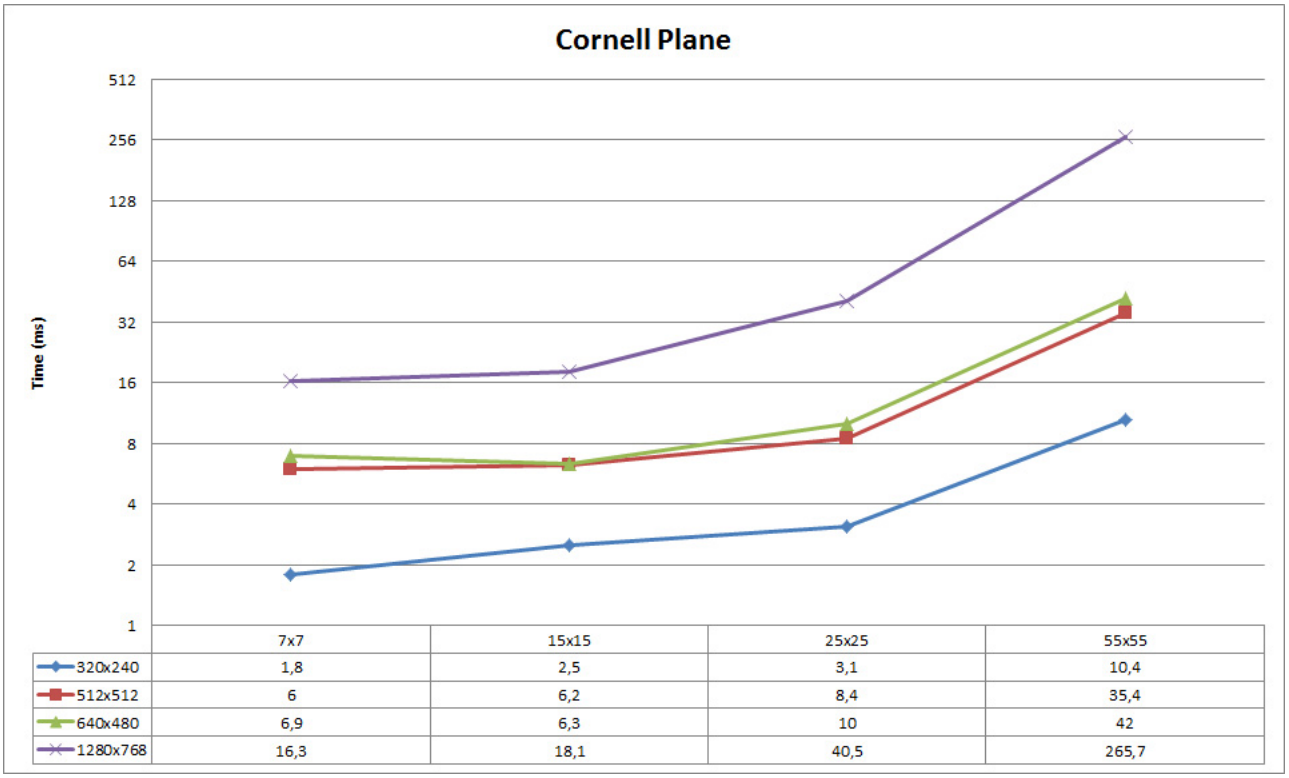
	7x7	15x15	25x25	55x55
320x240	1,8	2,5	3,1	10,3
512x512	6	6,2	8,4	35,5
640x480	6,9	6,3	10	42
1280x768	16,3	18,1	40,5	266

Figure 7.7: Graph of the filtered Cornell Diffuse scene



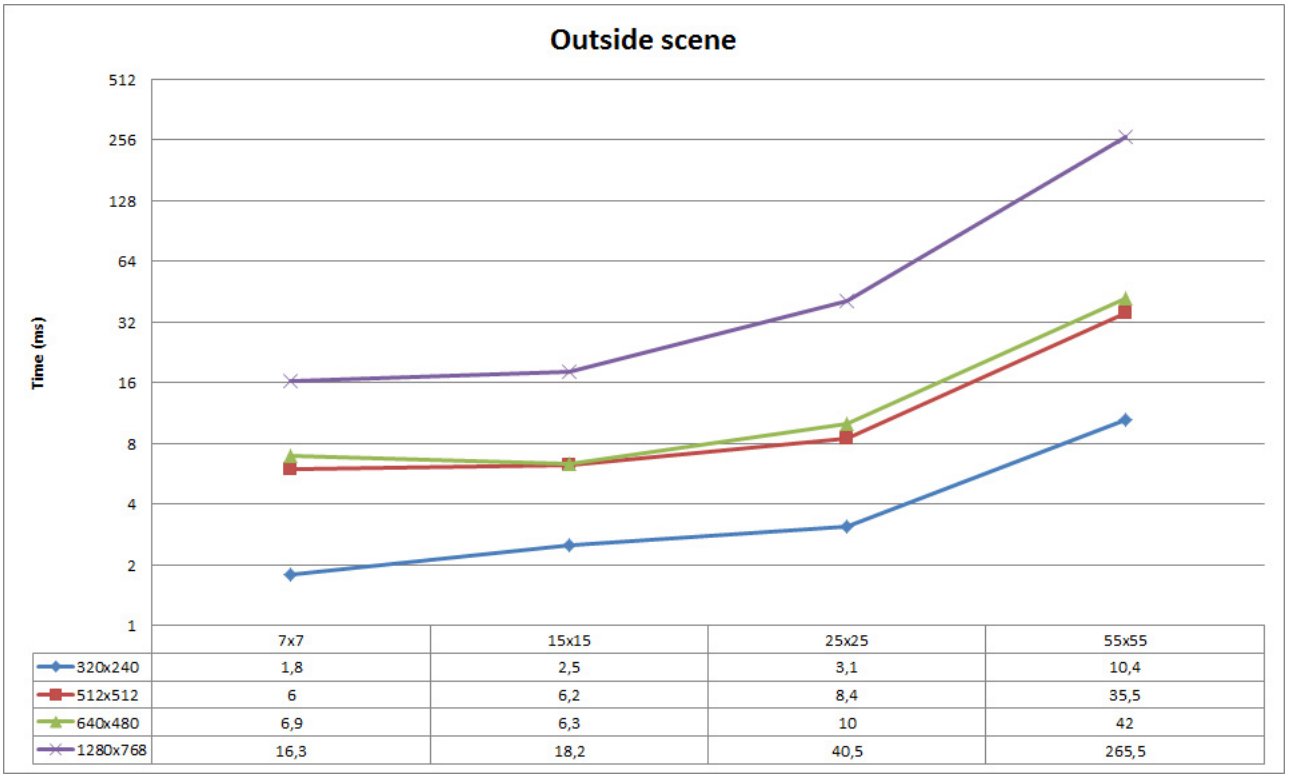
	7x7	15x15	25x25	55x55
320x240	1,8	2,5	3,1	10,4
512x512	6,1	6,3	8,4	35,6
640x480	6,9	6,3	10	42
1280x768	16,3	18,2	40,5	268

Figure 7.8: Graph of the filtered Cornell Glossy scene



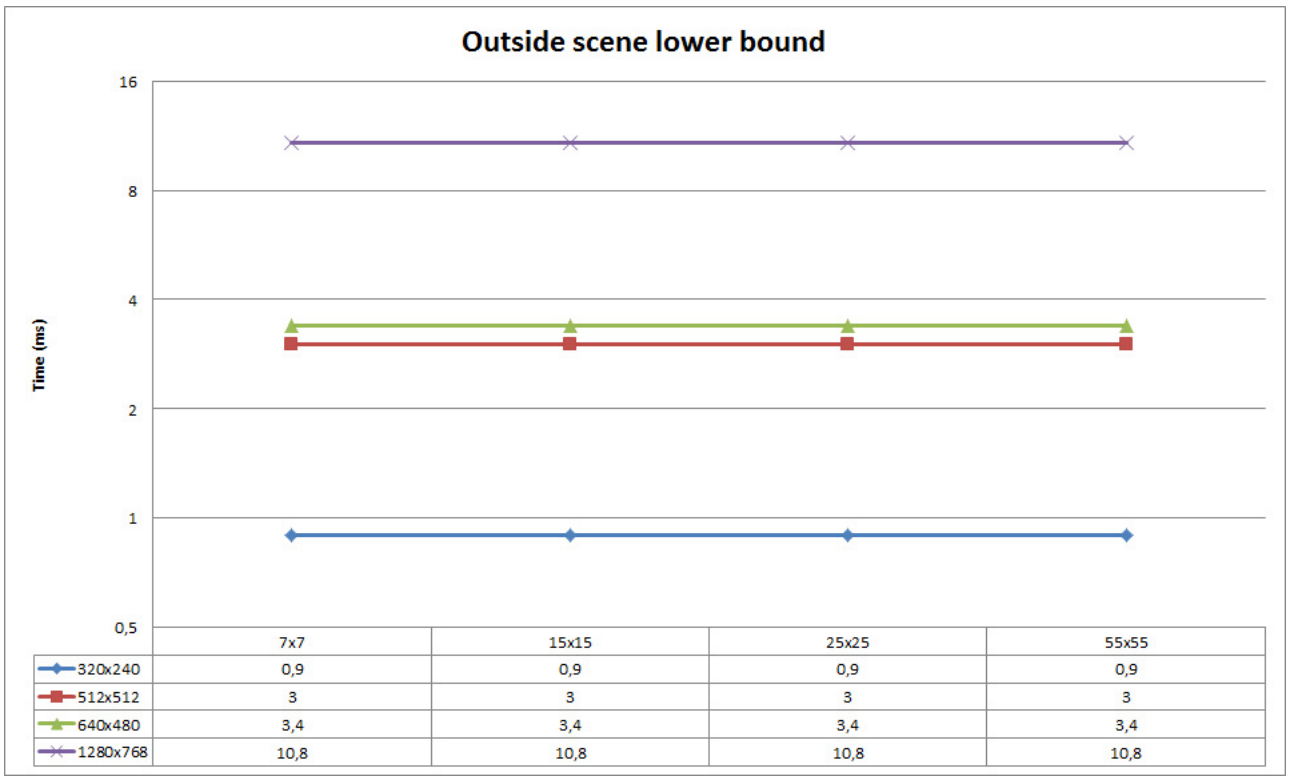
	7x7	15x15	25x25	55x55
320x240	1,8	2,5	3,1	10,4
512x512	6	6,2	8,4	35,4
640x480	6,9	6,3	10	42
1280x768	16,3	18,1	40,5	265,7

Figure 7.9: Graph of the filtered Cornell Plane scene



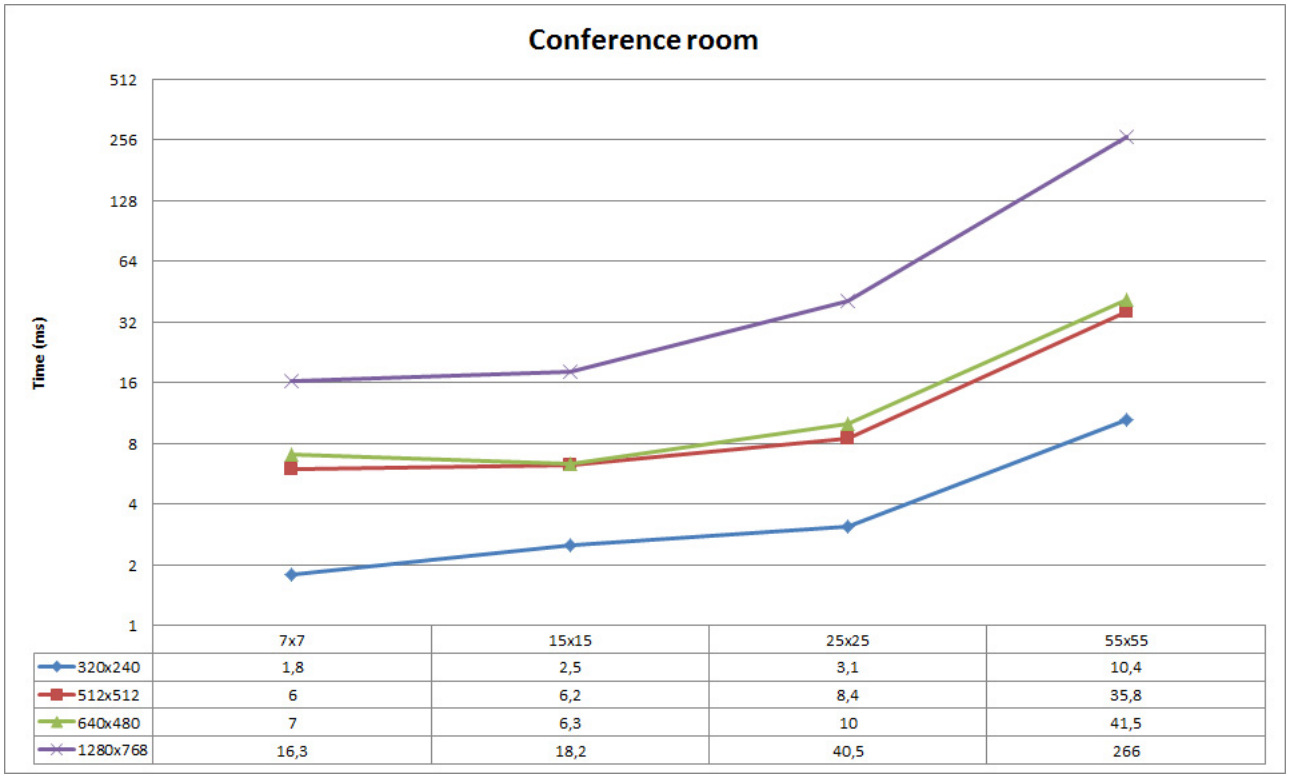
	7x7	15x15	25x25	55x55
320x240	1,8	2,5	3,1	10,4
512x512	6	6,2	8,4	35,5
640x480	6,9	6,3	10	42
1280x768	16,3	18,2	40,5	265,5

Figure 7.10: Graph of the filtered Outside scene



	7x7	15x15	25x25	55x55
320x240	0,9	0,9	0,9	0,9
512x512	3	3	3	3
640x480	3,4	3,4	3,4	3,4
1280x768	10,8	10,8	10,8	10,8

Figure 7.11: Graph of the filtered Outside scene with full view of the sky-box



	7x7	15x15	25x25	55x55
320x240	1,8	2,5	3,1	10,4
512x512	6	6,2	8,4	35,8
640x480	7	6,3	10	41,5
1280x768	16,3	18,2	40,5	266

Figure 7.12: Graph of the filtered Conference room scene

As shown in the figures, the speed decrease scales almost linearly with the screen width and height. This means when the resolution is increased $\times 4$ from 320×240 to 640×480 the speed reduces by $\approx 4\times$. However, there is one anomaly when the resolution is 1280×768 with a filter size of 55×55. This anomaly is due to the fact that the filter size is large causing more global memory transactions. If we let more work-items work in parallel at one time i.e. increasing the work-group size from 64 to 256 the speed will increase.

The filtering time for the case where the filter size and resolution were highest was improved by doing the work-group size adjustment, from $\approx 266\ ms$ to $\approx 160\ ms$. However, this adjustment only works well with high resolutions in combination with a large filter size. As we will see in the quality comparisons, a filter size of 55×55 will not be better than 21×21 or 25×25. The speed of the filter depends on the following attributes:

- Total amount of compute units i.e. processing elements.
- The core clock speed i.e. the speed of one processing element / work-item
- The memory clock speed i.e. the memory transfer speed.
- Size and speed of local memory. Local memory should be at least 16 kB for each work-group.
- Cache size and speed i.e. Best performance with cache-line size of 64 Bytes or greater

7.3 Quality results

Before we compare our new filter against RPF, we want to find optimal parameters that are a good trade-off between quality and speed. We are trying to find optimal settings for the following parameters: kw, σ_{dc}, σ_{ic} and σ_f. For each filter size we will tweak one parameter at a time until we find the settings with low MSE and low filter time. The scene that is used to find the parameters is Cornell Diffuse. Since some scene features have mostly high variance when depth of field is on, we will also test how the parameters affect the filtering quality with depth of field on. The filtering quality of scenes with reflective/refractive materials are mostly affected by the bounce direction and texture2 parameter. For this reason we will take the Cornell Glossy and Outside scenes as additional test scenes to tweak the direction and texture sigma parameters. To have a good starting point for each parameter we do all renders with a resolution of 512×512 pixels using the following starting values:

Parameter	Base Value	Experimentation values			
kw	21×21	11×11, 15×15, 21×21			
		1	2	3	4
σ_{dc}	0.5	0.10	0.25	1.00	2.50
σ_{ic}	0.3	0.10	0.25	1.00	2.50
σ_f depth,	1.0	0.25	0.50	2.50	5.00
σ_f normal,	0.1	0.05	0.25	0.50	1.00
σ_f texture,	0.1	0.05	0.25	0.50	1.00
σ_f direction,	0.5	0.10	0.25	1.00	2.50
σ_f texture2	0.1	0.05	0.25	0.50	1.00

Table 7.1: Data driven filter experimentation parameters

By changing the parameters we want to see how they affect the filtering quality. As we have seen in the speed results, for filter sizes larger than 25×25 the filter time becomes too high so we leave everything above this out of the experiments. We will try to find settings that will give us better MSE and filter times compared to the starting values. The filtered image for each set of parameters will be shown in appendix B, in combination with the ground truth, unfiltered image, MSE of the unfiltered image, the filtered image and the MSE of the filtered image. The MSE results for each set of parameters will be depicted in graphs here. Since the depth of field scenes are rendered at 16 spp, we will show those results in different graphs. We expect that certain parameters will give more change than others in terms of the given test scenes. Given this test setup we can test the following hypotheses:

- Hypothesis 1: Increasing the σ_{dc} and σ_{ic} parameters will reduce the noise level.
- Hypothesis 2: Increasing any σ_f will reduce the noise level.
- Hypothesis 3: Increasing the filter size will always decrease the noise level.

To test the first two hypotheses we will perform the parameter experiments and then evaluate the results. Additionally we will do one more experiment to test the third hypothesis, with the optimal filter parameters and filter size of 55×55 to test if these give better or worse results compared to the optimal filter size we will find.

7.3.1 Optimal filter size and parameters

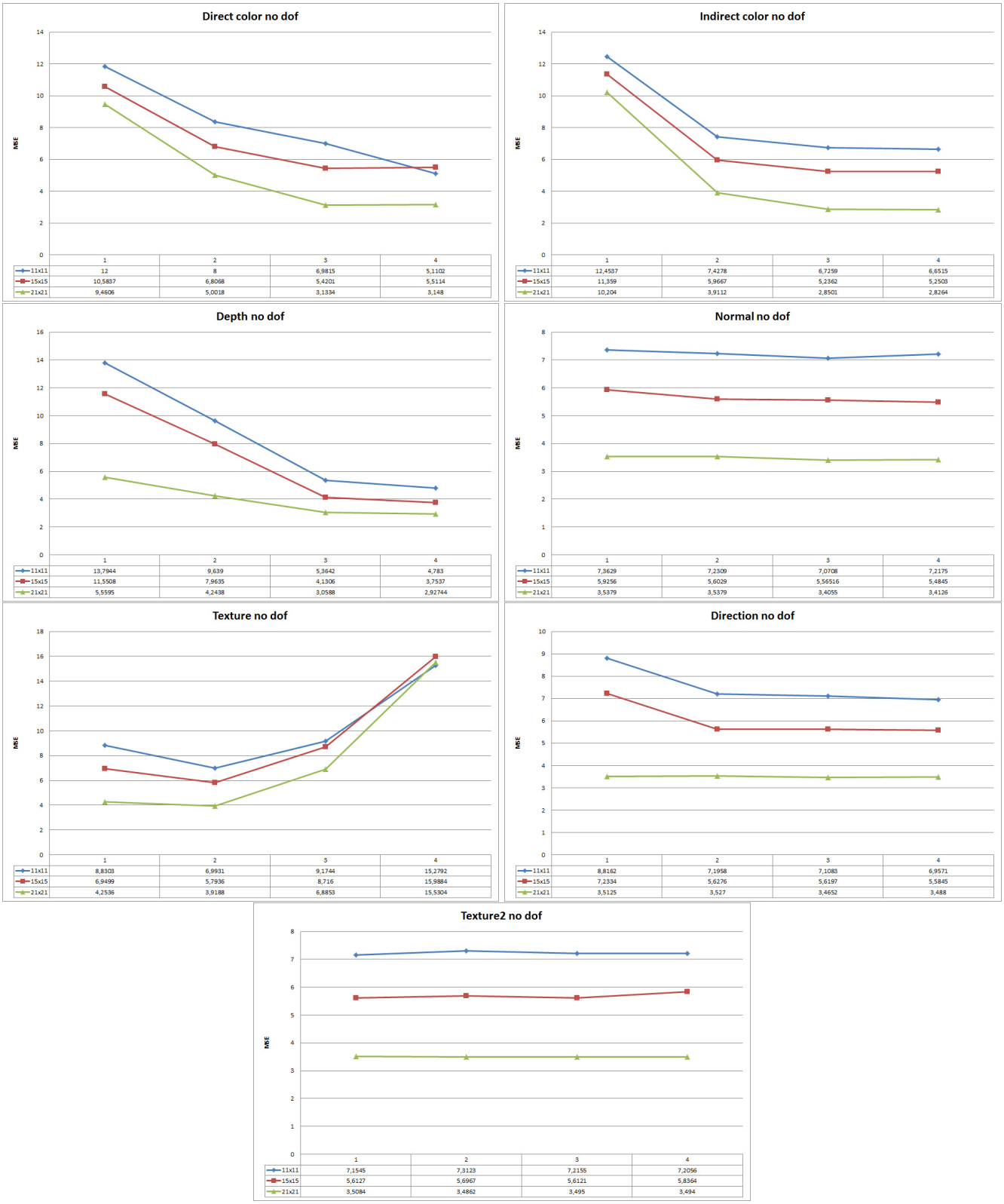


Figure 7.13: Parameter graphs for the Diffuse scene without depth of field. Each parameter changes the MSE of the output image depending on their value.

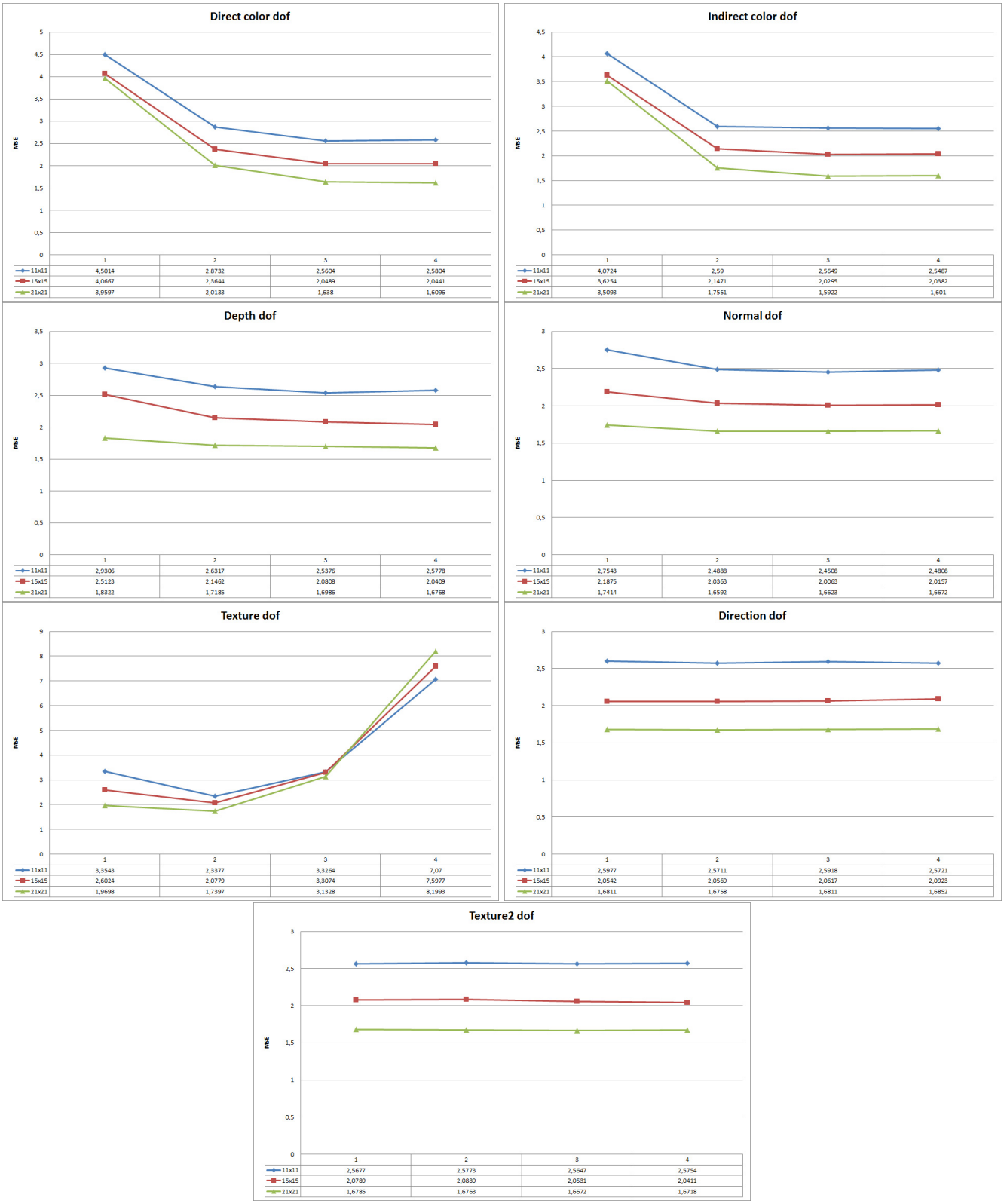


Figure 7.14: Parameter graphs for the Diffuse scene with depth of field. Each parameter changes the MSE of the output image depending on their value.

Now we will show the comparison between the σ_f direction and σ_f texture2 for the Glossy scene and Outside scene. These comparisons are done with a filter size of 21×21 since they gave the best MSE results at reasonable filter times. It is clear that the texture2 parameter is useful when there is a reflective surface in the scene, which is also shown and confirmed in the graphs.

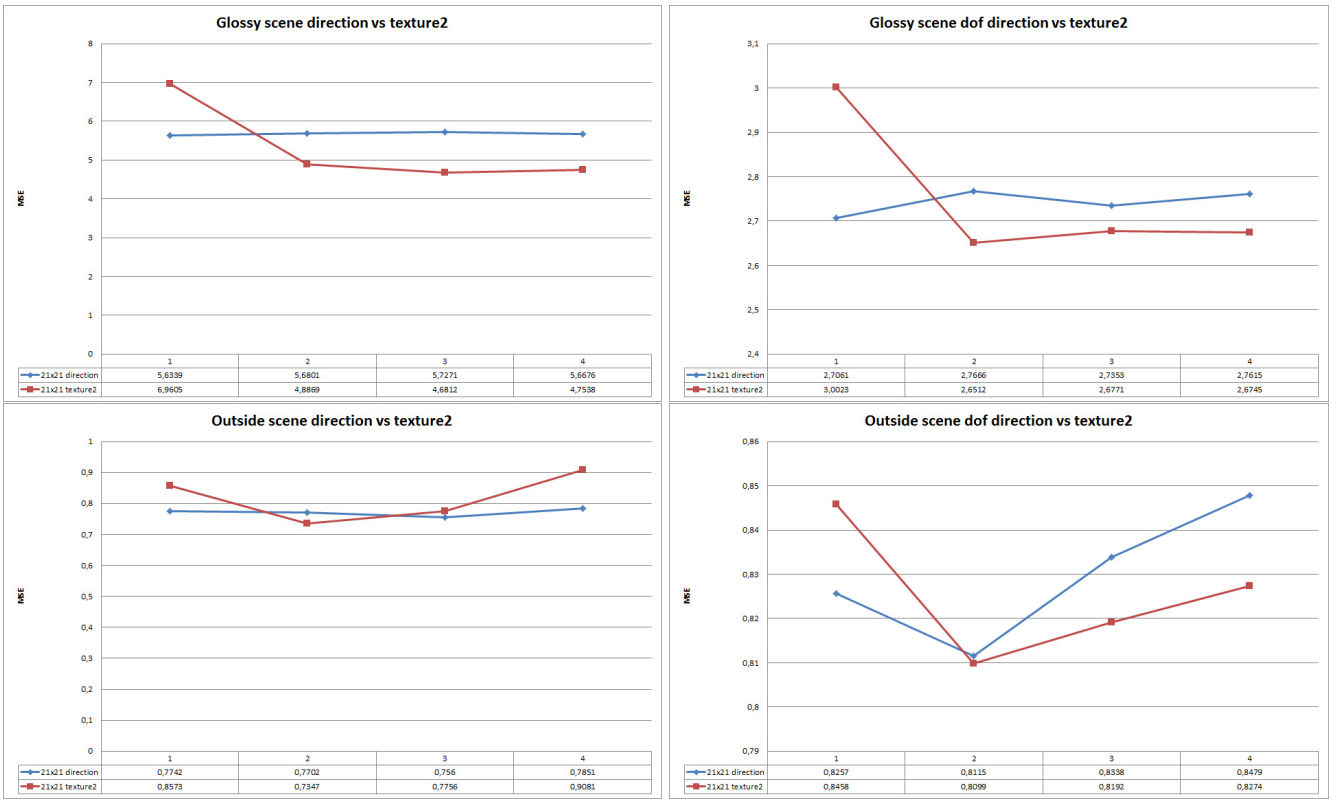


Figure 7.15: σ_f direction and σ_f texture2 parameter graphs for the Glossy and Outside scenes.

These graphs clearly show how changing the parameters can change the noise level (MSE) in the image. With this information we pick a set of parameters that will be used in the comparisons against RPF. These parameters are:

Parameter	Chosen values
kw	21×21
σ_{dc}	0.75
σ_{ic}	0.5
σ_f depth	1.75
σ_f normal	0.25
σ_f texture	0.1
σ_f direction	0.75
σ_f texture2	0.1

Table 7.2: Data driven filter experimentation final parameters.

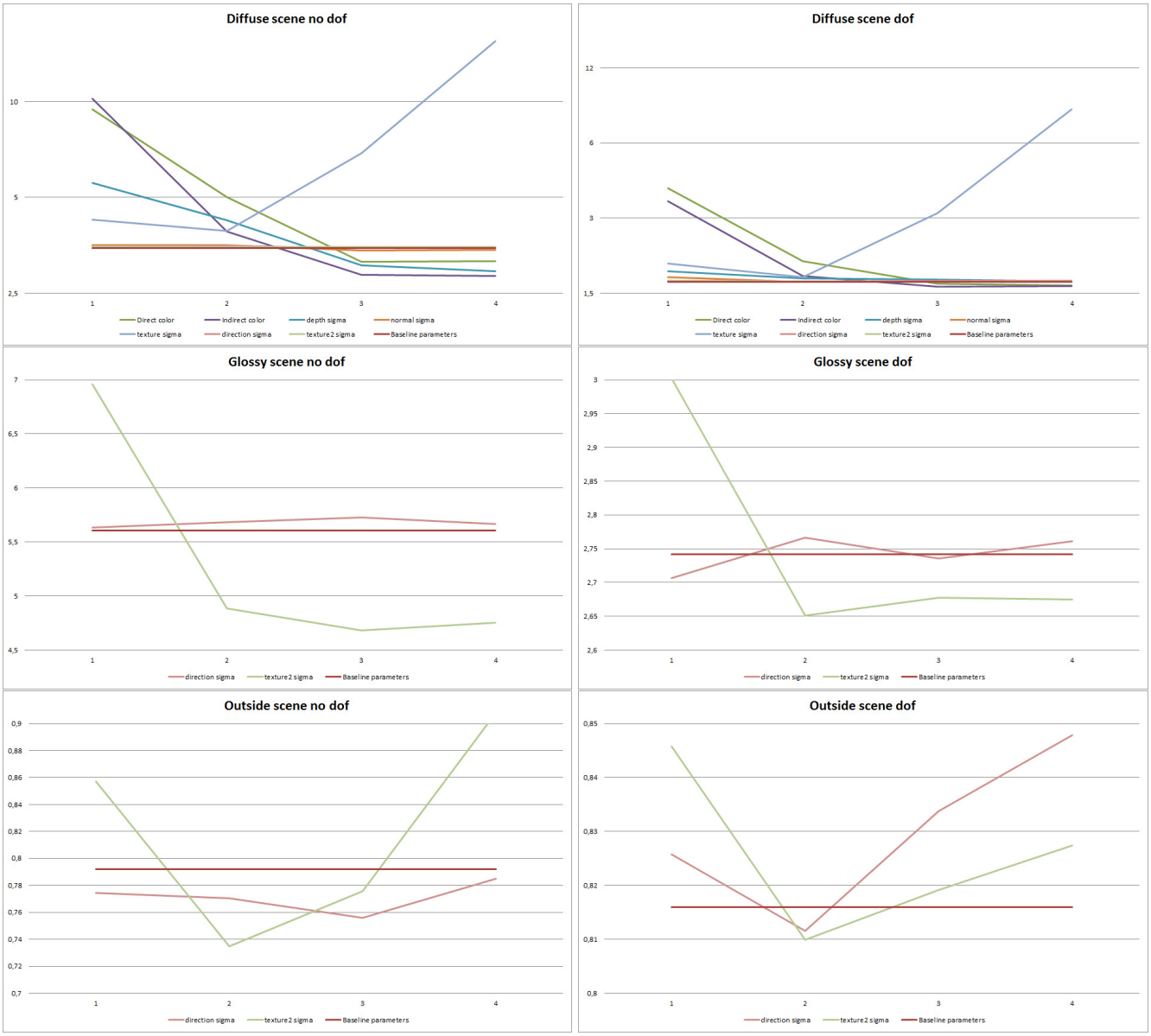
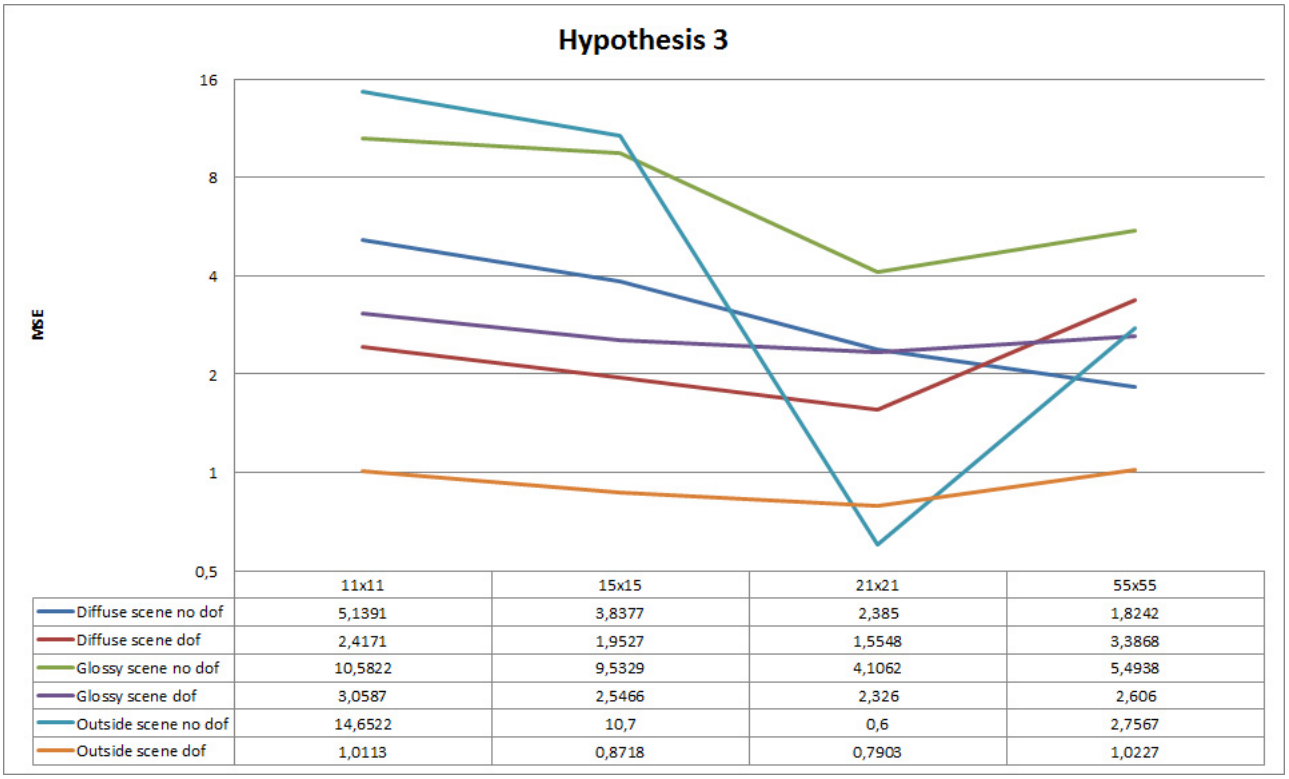


Figure 7.16: Increasing parameter values vs MSE.

Figure 7.16 confirms hypothesis 1. When σ_{dc} and σ_{ic} increase, the MSE decreases. Although the MSE is lower, increasing σ_{dc} and σ_{ic} causes details like soft shadows to over blur making the image different from the ground truth (visually), while at the same time the noise level decreased (measured) i.e. over blurring increases MSE less than noise. At the same time we see that increasing other parameters does not directly yield better quality. This is especially true for the σ_f texture parameter. When we increase this parameter we tell the filter that texture detail is not so important i.e. neighboring pixels get a high weight causing the pixel to over blur. The σ_f direction parameter seems to behave rather random, but in most cases a low value works best. Since increasing the σ_f parameters does not always decrease MSE, hypothesis 2 is incorrect. Most significant changes in MSE occur when σ_f texture is changed and the least significant changes are seen when we change the σ_f direction parameter. With the optimal parameters we will test the three scenes used in the previous experiments with filter sizes of 11×11, 15×15,21×21 and 55×55 to test our third hypothesis. The resulting filtered outputs

are shown in appendix C. Figure 7.17 shows the results of these experiments.



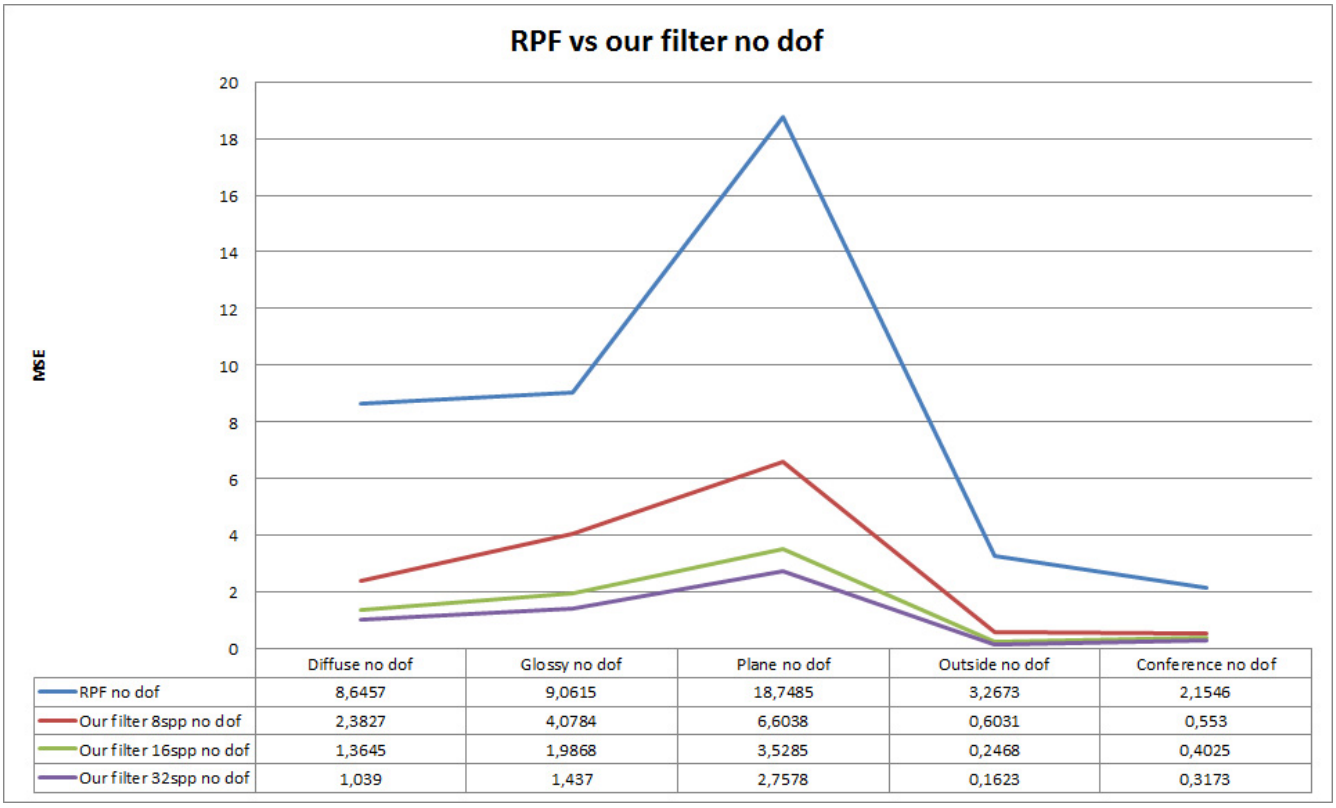
	11x11	15x15	21x21	55x55
Diffuse scene no dof	5,1391	3,8377	2,385	1,8242
Diffuse scene dof	2,4171	1,9527	1,5548	3,3868
Glossy scene no dof	10,5822	9,5329	4,1062	5,4938
Glossy scene dof	3,0587	2,5466	2,326	2,606
Outside scene no dof	14,6522	10,7	0,6	2,7567
Outside scene dof	1,0113	0,8718	0,7903	1,0227

Figure 7.17: Hypothesis 3

The graph clearly shows that there is an optimal filter size (around 21×21). We claimed that by increasing the filter size the noise level (MSE) would become lower, but from the results we can see this is not the case. Although there is visually less noise with a larger filter size, the soft shadows and reflections are clearly over blurred causing the MSE to increase. The cause of this can be the lack of the spatial term in the filter. In fact, every time the filter size increases, the reflections blur more.

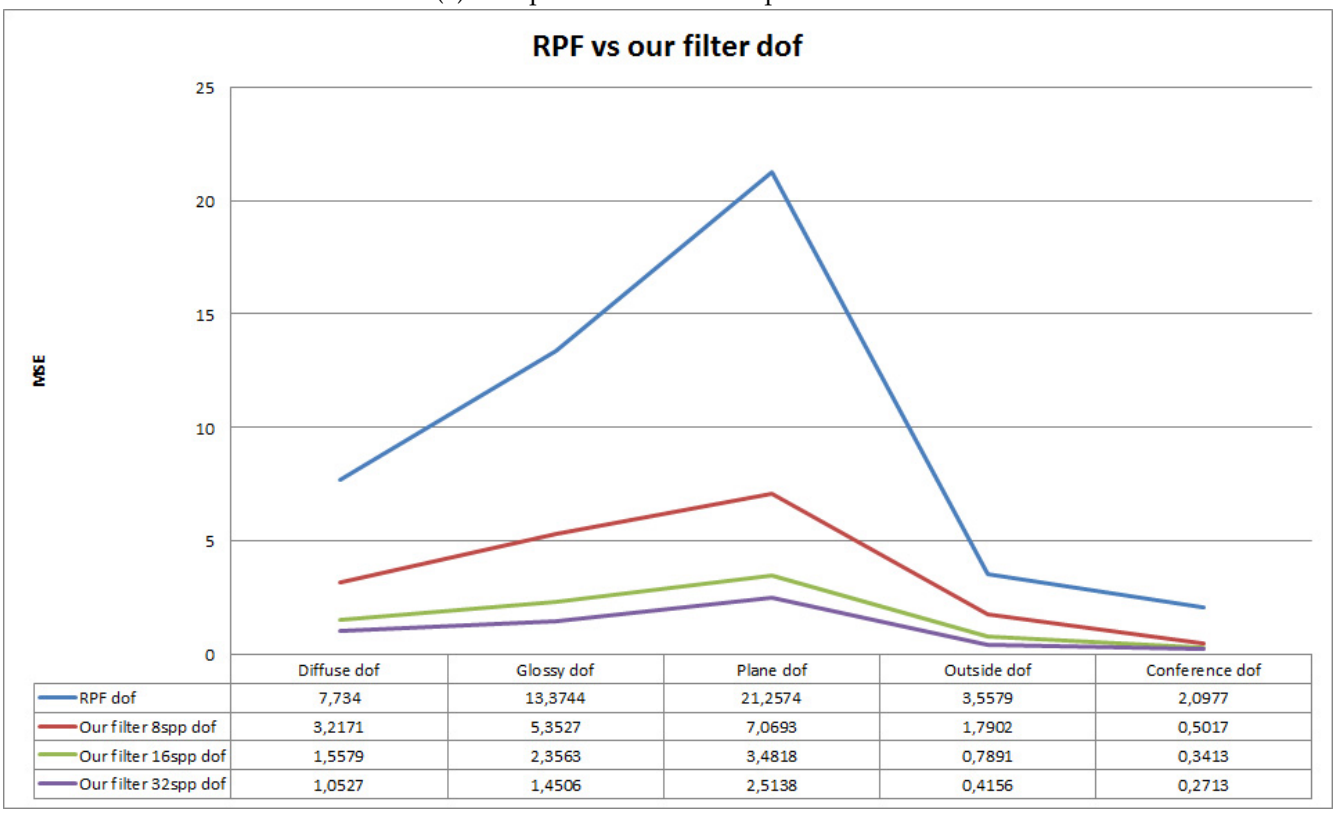
7.3.2 Data driven filter vs RPF

With the optimal parameters we will compare the data driven filter directly to RPF. We will use the five test scenes with and without depth of field. We only look at MSE quality (quantitative) and visual quality i.e. how it looks compared to the ground truth (qualitative). Each output is shown in appendix D and figure 7.18 shows a graph comparing RPF MSE with our filters MSE for each scene without and with depth of field.



	Diffuse no dof	Glossy no dof	Plane no dof	Outside no dof	Conference no dof
RPF no dof	8,6457	9,0615	18,7485	3,2673	2,1546
Our filter 8spp no dof	2,3827	4,0784	6,6038	0,6031	0,553
Our filter 16spp no dof	1,3645	1,9868	3,5285	0,2468	0,4025
Our filter 32spp no dof	1,039	1,437	2,7578	0,1623	0,3173

(a) Comparisons without depth of field.

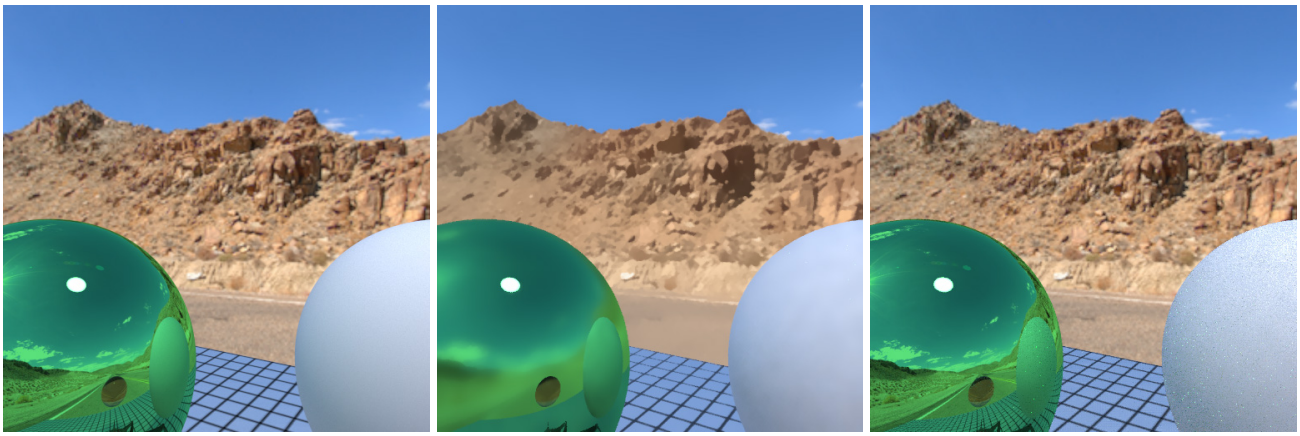


	Diffuse dof	Glossy dof	Plane dof	Outside dof	Conference dof
RPF dof	7,734	13,3744	21,2574	3,5579	2,0977
Our filter 8spp dof	3,2171	5,3527	7,0693	1,7902	0,5017
Our filter 16spp dof	1,5579	2,3563	3,4818	0,7891	0,3413
Our filter 32spp dof	1,0527	1,4506	2,5138	0,4156	0,2713

(b) Comparisons with depth of field.

Figure 7.18

Visually RPF has little or no noise in the image. However, when compared to the ground truth it is clear that some parts of the scenes are over blurred causing the MSE to go up. Because we tread the sky-box as a special case (figure 7.19), we get better results in the outside scene. Reflections are also over-blurred in the RPF filter outputs which is not the case in our filter. RPF is superior when it comes to removing noise, but it comes at a cost.



(a) Ground truth. (b) RPF filter output. (c) Our filter output (8spp).

Figure 7.19: Reflection and sky-box filtering

Chapter 8

Conclusion and future work

In this thesis we set out to achieve real-time performance in noise free path traced scenes. The focus on physically based path tracing comes from the fact that it can produce photo-realistic images with complex effects like depth of field, soft shadows and motion blur. Recently hardware became fast enough to enable real-time path tracing at low sampling rates. These low sampling rates cause the output image to look noisy. This is why we proposed a fast post-process filter that produces near noise free outputs given a low sample count in real-time. We use as much information from the path tracer as possible to detect noise in the image as opposed to only using color information.

8.1 Conclusion

In the first part of this thesis we discussed theory and techniques that help understand the background theory of light transport, path tracing, GPGPU and filtering techniques.

In the second part of the thesis we focused on path tracing, and the implementation of a path tracer on the GPU using OpenCL. This path tracer was able to render scenes with the complexity we need to test filtering techniques. It is also fast enough to allow real-time path tracing, which is what we need as a starting point for our filter. After this we discussed the implementation of Random Parameter Filtering which is a technique that uses the statistical dependencies between the random parameters and renderer output to detect and remove noise. This technique was implemented on the c++ (host) side of the path tracer on the CPU. The authors used a multi-threaded implementation whereas we used a single threaded implementation. Lastly we discuss the implementation of the data driven filter. This discussion includes the data structures, theoretical explanation and optimizations aimed on filtering speed.

In the last part of the thesis we evaluate the data driven filter on speed and quality. First we evaluate the filtering speed using pre-set parameters. Secondly we try to find the optimal filter parameters that give the best speed/quality balance. We do this by carefully changing parameters, evaluate the filtered outputs and pick optimal parameters at the end. Finally we compare the quality of our filter with RPF.

Since the RPF filter is focused on quality instead of speed, we do not compare the two methods in terms of speed. Considering the difference in filtering speed we are still able to achieve comparable quality with a significantly lower filter time i.e. over $10.000\times$ filtering speed increase.

Contributions

We show the speed and quality of the filter and we compared it to RPF to find out how well it performs compared to a state of the art approach. The contributions are:

- The filter uses the bounce direction and multiple bounce texture information to preserve texture detail in reflecting and refracting objects. By doing this we are able to prevent over-blurring in reflecting objects, which causes loss of realism. We also preserve the refraction and reflection in glass materials caused by Fresnel equations.
- We compute the variance of all scene features in real-time and use this to detect the noise level in each pixel. Using the mean μ and variance σ^2 from these scene features, in combination with separate filtering for direct and indirect lighting, we achieve better quality. This allows us to preserve detail that is present in one buffer (hard shadows are not present in the indirect lighting buffer) and not in the other e.g. global illumination is not present in the direct lighting buffer.
- We take sky-boxes into account by treating them as a special case in our filter to prevent over-blurring. In our comparisons we show this has a significant effect on the quality of the filter.
- Using the variance we can filter the depth of field effect by detecting how much variance there is in the geometric information of one pixel. When a pixel is out of focus the variance tends to be high, which causes more filtering to occur in that pixel.
- We optimized our data structures and code structure to make optimal use of modern GPU hardware. We show that these optimization make our filter faster and more stable than without optimizations. We make optimal use of local memory and take the GPU streaming architecture into account to gain a significant speed-up compared to the non-optimized filter. We also apply a trick that causes a speed-up of $\approx 3\times$ by leaving holes in the filter neighborhood. This way we can skip neighboring pixels which will save computational time.
- We provide analysis of the filter in terms of speed and quality. Then we show the comparison between RPF and our filter to see how well our filter performs in terms of quality. Five test scenes are used which all vary in complexity to show the diversity of our filter.
- We have created a renderer that supports our filter implementations and is able to achieve real-time performance.

8.2 Drawbacks and future work

This filter is built to work in real-time situations. It can be useful to use this filter in production renderers but it will never converge to a correct result when more samples are taken. It is possible to take the change in variance for each pixel into account to detect how much noise there is left in the unfiltered image, but this is left open for future work. When only 8 samples per pixel are taken, the filter under-filters glossy reflections and the depth of field effect. It is possible to apply adaptive sampling using the weights that are calculated by the filter. After looping over the neighborhood of pixels we are left with a total weight which is used for normalization. The total weight for each pixel can be used as a importance map. This can be used to do adaptive sampling in the renderer which is possibility for future work i.e. take more samples where the

filter does not benifit from the information given by neighboring pixels. It is also possible to use additional information from the renderer to improve the filter. Some examples are surface roughness and the world position of the second bounce. In the future hardware will continue to become better which can lead to path traced games in the near future. Until that time more research is needed to improve the speed of rendering. The next step is to apply path tracing and filtering techniques into modern game engines to support or replace rasterisation.

Bibliography

[1] Quake wars, ray traced, 2010. URL http://en.wikipedia.org/wiki/Quake_Wars:_Ray_Traced.

[2] J. Bikker. Arauna real-time ray tracing and brigade real-time path tracing, 2012. URL http://igad.nhtv.nl/~bikker/.

[3] Guerrilla. Killzone 4, real-time reflections, 2013. URL http://www.killzone.com.

[4] Timo Aila and Samuli Laine. Understanding the efficiency of ray traversal on gpus. In *Proceedings of the Conference on High Performance Graphics 2009*, pages 145–149, New York, NY, USA, 2009. ACM.

[5] J. Bikker. Ray tracing in real-time games, november 2012.

[6] Samuli Laine, Tero Karras, and Timo Aila. Megakernels considered harmful: Wavefront path tracing on gpus. In *Proc. High-Performance Graphics*, 2013.

[7] Dietger van Antwerpen. Unbiased physically based rendering on the gpu, 2010.

[8] James T. Kajiya. The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4):143–150, August 1986. ISSN 0097-8930. doi: 10.1145/15886.15902. URL http://doi.acm.org/10.1145/15886.15902.

[9] Otoy Inc. Brigade renderer, 2013. URL http://raytracey.blogspot.co.nz/.

[10] Johannes Hanika Hendrik P.A. Lensch Holger Dammertz, Daniel Sewtz. Edge-avoiding Á-trous wavelet transform for fast global illumination filtering. In *In Proceeding of High Performance Graphics*, pages 67–75, 2010.

[11] Dieter W. Fellner Karsten Schwenk, Johannes Behr. Practical noise reduction for progressive stocastic ray tracing with perceptual control. *Computer Graphics and Applications, IEEE*, 32(6):46–55, November-December 2012.

[12] Pradeep Sen and Soheil Darabi. On filtering the noise from the random parameters in monte carlo rendering. *ACM Trans. Graph.*, 31(3):18:1–18:15, june 2012.

[13] Ben Weiss. Fast median and bilateral filtering. *ACM Transactions on Graphics*, 25(3):519–526, 2006.

[14] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.

[15] J. van de Weijer and R. van den Boomgaard. Local mode filtering. In *in Proceedings of the conference on IEEE Computer Vision and Pattern Recognition*, pages 428–433, 2001.

[16] Frédo Durand and Julie Dorsey. Fast bilateral filtering for the display of high-dynamic-range images. *ACM Trans. Graph.*, 21(3):257–266, July 2002. ISSN 0730-0301. doi: 10.1145/566654.566574. URL http://doi.acm.org/10.1145/566654.566574.

[17] Georg Petschnigg, Richard Szeliski, Maneesh Agrawala, Michael Cohen, Hugues Hoppe, and Kentaro Toyama. Digital photography with flash and no-flash image pairs. *ACM Trans. Graph.*, 23(3):664–672, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015777. URL http://doi.acm.org/10.1145/1015706.1015777.

[18] Elmar Eisemann and Frédo Durand. Flash photography enhancement via intrinsic relighting. *ACM Trans. Graph.*, 23(3):673–678, August 2004. ISSN 0730-0301. doi: 10.1145/1015706.1015778. URL http://doi.acm.org/10.1145/1015706.1015778.

[19] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan. Ray tracing on programmable graphics hardware. *ACM Trans. Graph.*, 21(3):703–712, July 2002. ISSN 0730-0301.

[20] Amy Williams, Steve Barrus, R. Keith, and Morley Peter Shirley. An efficient and robust ray-box intersection algorithm. *Journal of Graphics Tools*, 10:54, 2003.

[21] Tomas Möller and Ben Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, October 1997. ISSN 1086-7651. doi: 10.1080/10867651.1997.10487468. URL http://dx.doi.org/10.1080/10867651.1997.10487468.

[22] Christophe Schlick. An inexpensive brdf model for physically-based rendering. *Computer Graphics Forum*, 13(3):233–246, 1994.

[23] Hanchuan Peng. Matlab package for mutual information computation, 2009. URL http://www.mathworks.com/matlabcentral/fileexchange/14888.

[24] Songfeng Zheng. Math 541: Statistical theory ii. methods of evaluating estimators, 2012. URL http://people.missouristate.edu/songfengzheng/Teaching/MTH541/Lecture%20notes/evaluation.pdf.

[25] Dr. Dmitriy Vatolin, Alexey Moskvin, Oleg Petrov, Sergey Putilin, Sergey Grishin, Arsaev Marat. Msu video quality measurement tool, 2013. URL http://compression.ru/video/quality_measure/video_measurement_tool_en.html.

[26] CIELUV, 2013. URL http://en.wikipedia.org/wiki/CIELUV.

[27] Pablo Bauszat, Martin Eisemann, and Marcus Magnor. Guided image filtering for interactive high-quality global illumination. In *Proceedings of the Twenty-second Eurographics conference on Rendering*, EGSR'11, pages 1361–1368, Aire-la-Ville, Switzerland, Switzerland, 2011. Eurographics Association. doi: 10.1111/j.1467-8659.2011.01996.x. URL http://dx.doi.org/10.1111/j.1467-8659.2011.01996.x.

[28] Eric Weisstein. Sample mean, 2013. URL http://mathworld.wolfram.com/SampleMean.html.

[29] Eric Weisstein. Sample variance, 2013. URL http://mathworld.wolfram.com/SampleVariance.html.

[30] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN 0201896842.

[31] Khronos Group. Opencl specifications, 2011. URL http://www.khronos.org/registry/cl/specs/opencl-1.1.pdf.

[32] Chuang Yung-Yu Li Tzu-Mao, Wu Yu-Ting. Sure-based optimization for adaptive sampling and reconstruction. *ACM Transactions on Graphics (TOG), SIGGRAPH Asia*, 31(6), November 2012.

[33] Johannes Gunther, Stefan Popov, Hans-Peter Seidel, and Philipp Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 978-1-4244-1629-5. doi: 10.1109/RT.2007.4342598. URL http://dx.doi.org/10.1109/RT.2007.4342598.

[34] Fatih Porikli. Constant time o(1) bilateral filtering. In *Computer Vision and Pattern Recognition*, pages 1–8, Anchorage, AK, June 2008.

[35] Yang Qingxiong, Tan Kar-Han, Ahuja Narendra. Real-time o(1) bilateral filtering. In *Computer Vision and Pattern Recognition*, pages 557–564, Miami, F, June 2009.

[36] Otoy Inc. Octane renderer, 2013. URL http://render.otoy.com.

Appendices

Appendix A

Data driven filter algorithm

Algorithm 8 Data driven filter

function FILTER(F, GB, kw, σ_f, s_f)
  $half_{kw} \leftarrow (kw - 1)/2$
  $w_d, w_i, tw_d, tw_i \leftarrow 0$
  $C_{fd}, C_{fi} \leftarrow black$
  Calculate direct and indirect color variance:
  $\sigma_{dc}^2 \leftarrow \frac{1}{2 \cdot (\sigma_{dc})2}$, $\sigma_{ic}^2 \leftarrow \frac{1}{2 \cdot (\sigma_{ic})2}$
  Calculate σ_f^2 with equation 6.10
  Loop over the neighborhood of pixel. x and y are the center pixel position.
  for $y \leftarrow y - half_{kw}$ to $y + half_{kw} - 1$ **do**
    for $x \leftarrow x - half_{kw}$ to $x + half_{kw} - 1$ **do**
      Get neighborhood FilterVector F_N from memory.
      Calculate local color weights w_{dc} and w_{ic} with equation 6.11 and 6.12
      Calculate local scene feature weights w_{fd} and w_{fi} with equation 6.13 and 6.14
      Calculate final direct and indirect weight for this neighbor
      $w_d \leftarrow w_{dc} \cdot w_{fd}$,
      $w_i \leftarrow w_{ic} \cdot w_{fi}$
      Add weighted color contribution of neighboring pixel to this pixel
      $C_{fd} \leftarrow C_{fd} + w_d \cdot F_{N0}$
      $C_{fi} \leftarrow C_{fi} + w_i \cdot F_{N1}$
      Add direct and indirect neighbor weight to direct and indirect total weight
      $tw_d \leftarrow tw_d + w_d$
      $tw_i \leftarrow tw_i + w_i$
    end for
  end for
  Normalize direct and indirect color and calculate final filtered pixel color
  $C_f \leftarrow \frac{C_{fd}}{tw_d} + \frac{C_{fi}}{tw_i}$
  return C_f
end function

Appendix B

Data driven filter MSE for parameter experiments

This appendix shows the result of every parameter experiment that is done. It is structured as follows:

- The results of all experiments done with a filter size of 11×11

  Figures B.1, B.2, B.3 and B.4 show the experiments without depth of field.

  Figures B.5, B.6, B.7 and B.8 show the experiments with depth of field.
- The results of all experiments done with a filter size of 15×15

  Figures B.9, B.10, B.11 and B.12 show the experiments without depth of field.

  Figures B.13, B.14, B.15 and B.16 show the experiments with depth of field.
- The results of all experiments done with a filter size of 21×21

  Figures B.17, B.18, B.19 and B.20 show the experiments without depth of field.

  Figures B.21, B.22, B.23 and B.24 show the experiments with depth of field.
- Additional experiments done with the σ_f direction and σ_f texture2 parameters on the Cornell glossy and Outside scenes

  Figures B.25 and B.26 show experiments without depth of field.

  Figures B.27 and B.28 show experiments with depth of field.

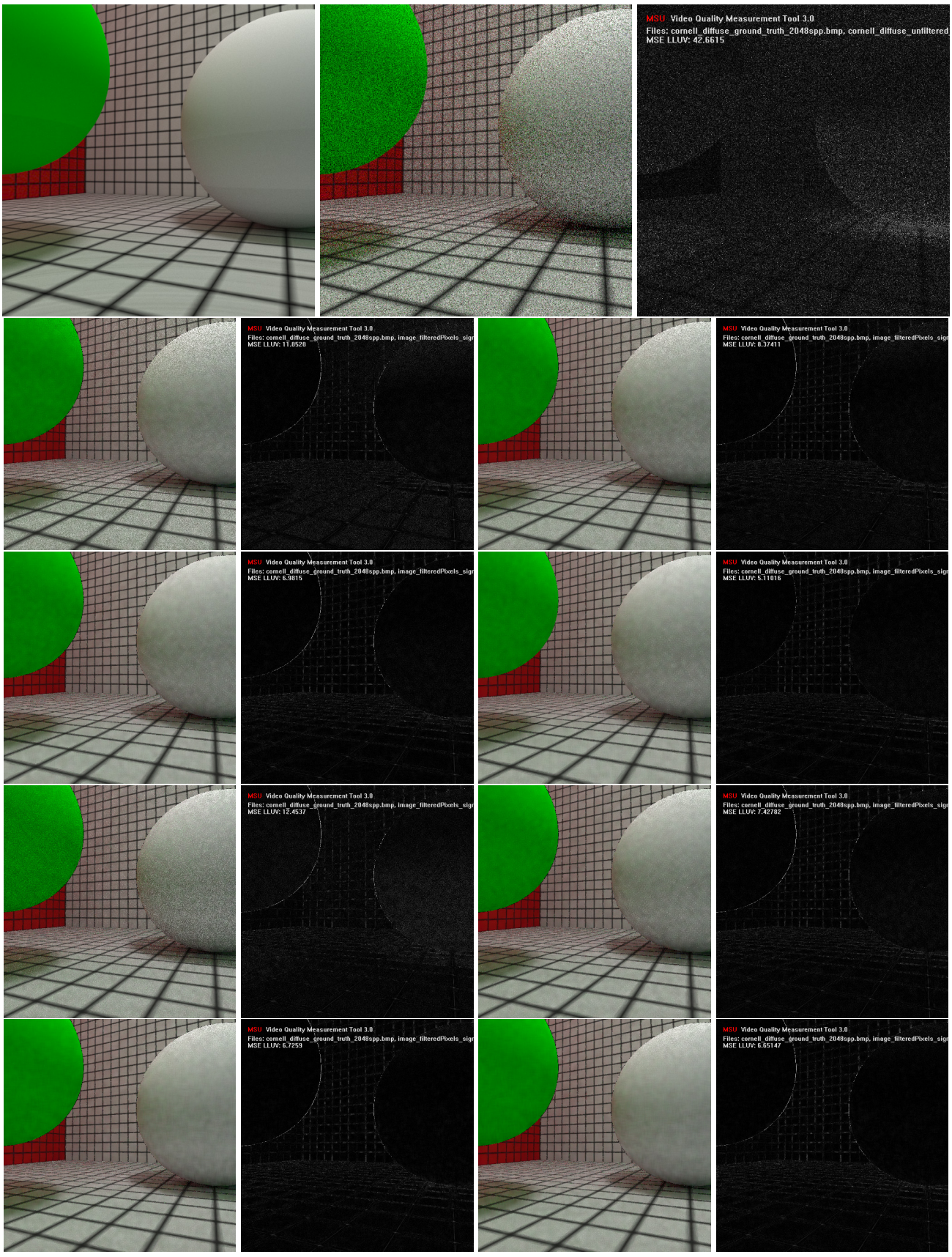


Figure B.1: Experiment 11×11, σ_{dc} and σ_{ic}. Row 1: Groundtruth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

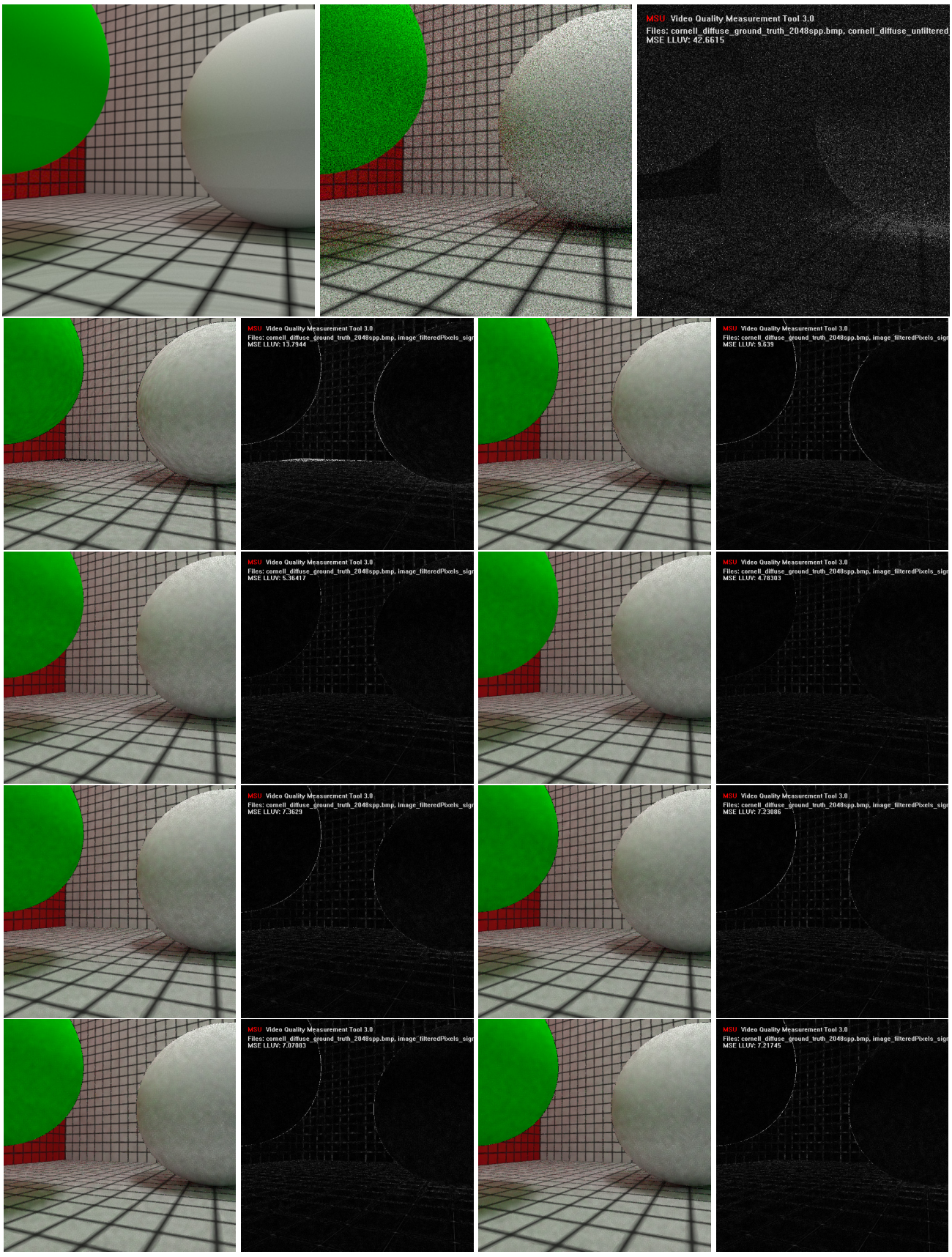


Figure B.2: Experiment 11×11, σ_f depth and σ_f normal. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

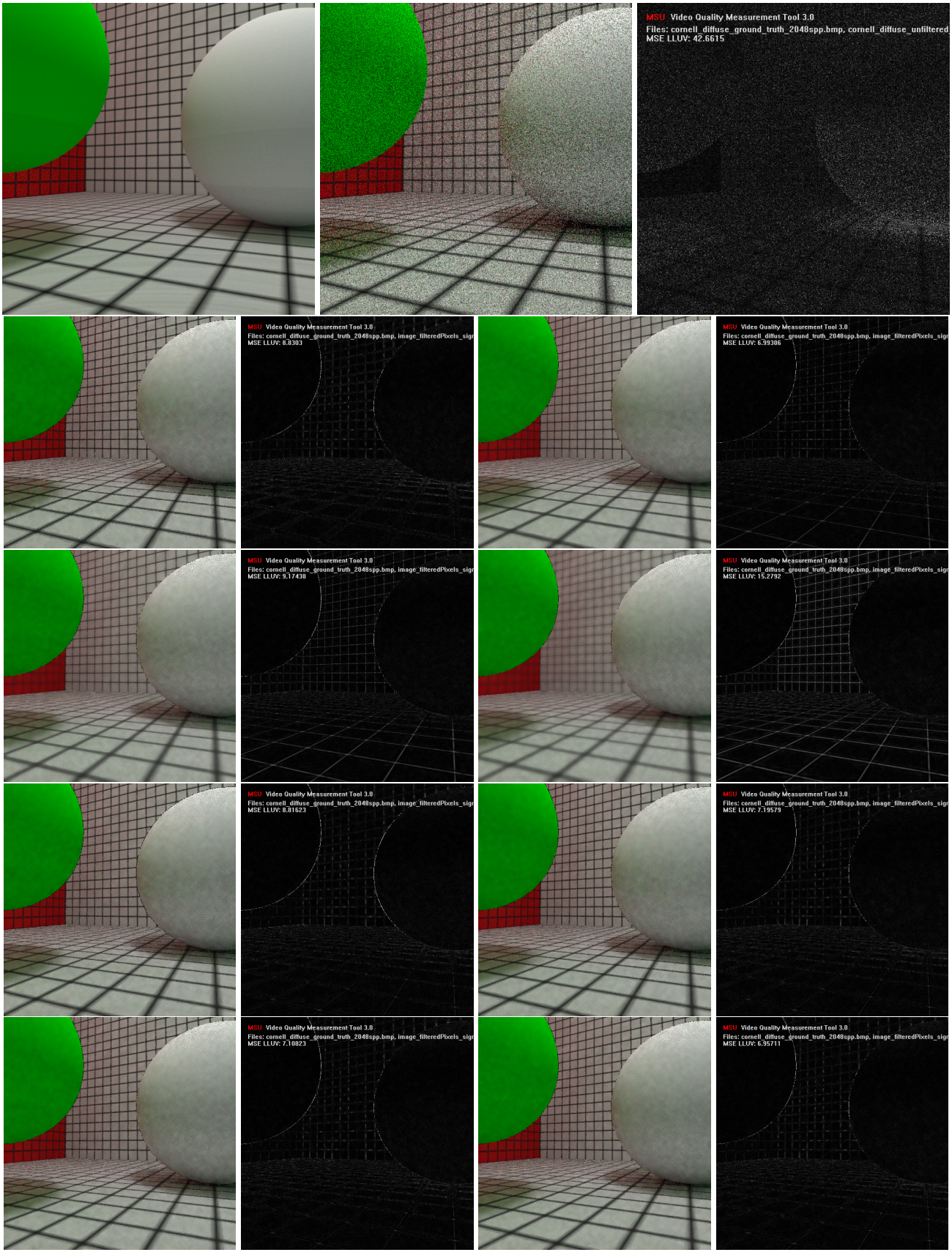


Figure B.3: Experiment 11×11, σ_f texture and σ_f direction. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

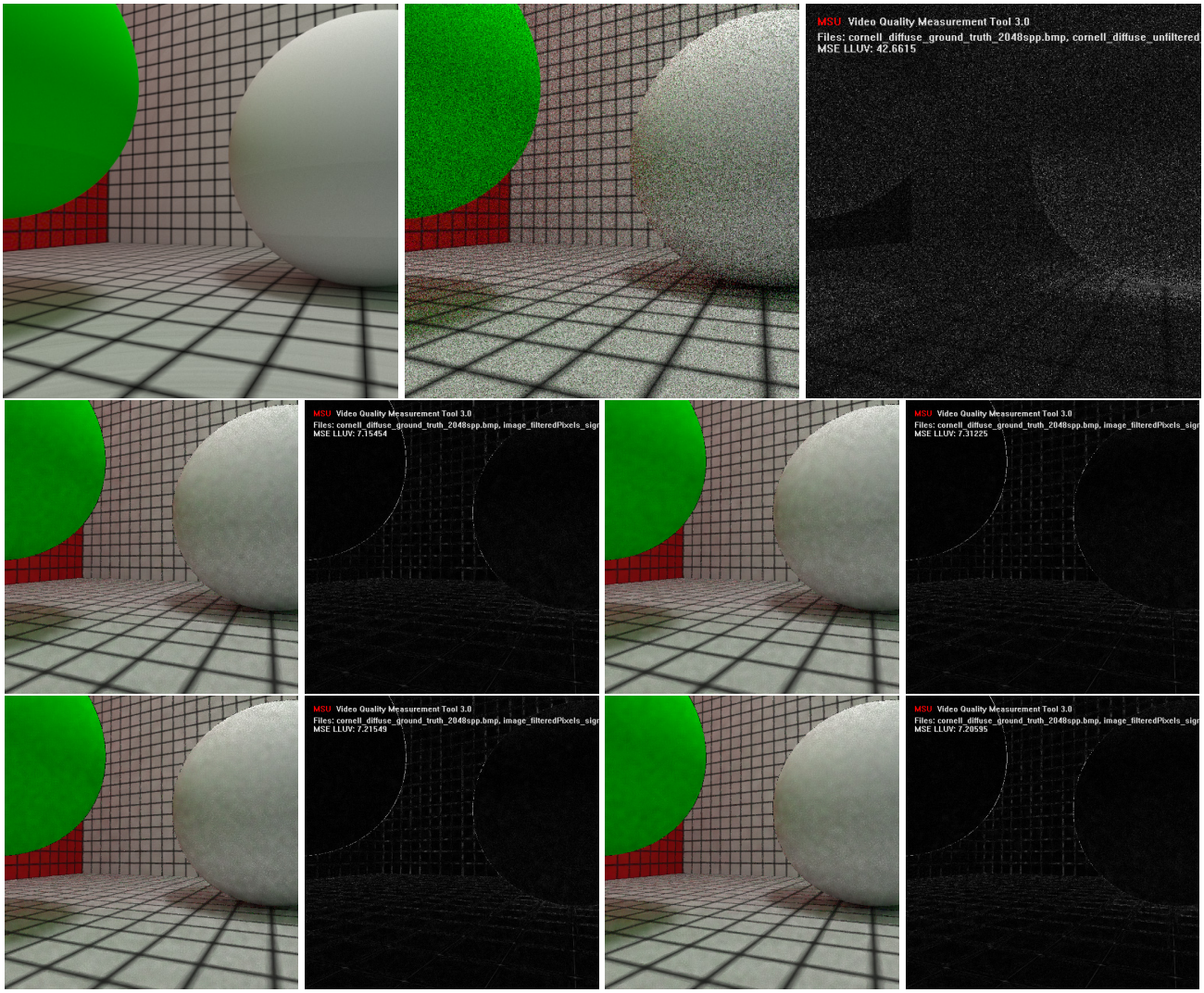


Figure B.4: Experiment 11×11, σ_f texture2. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

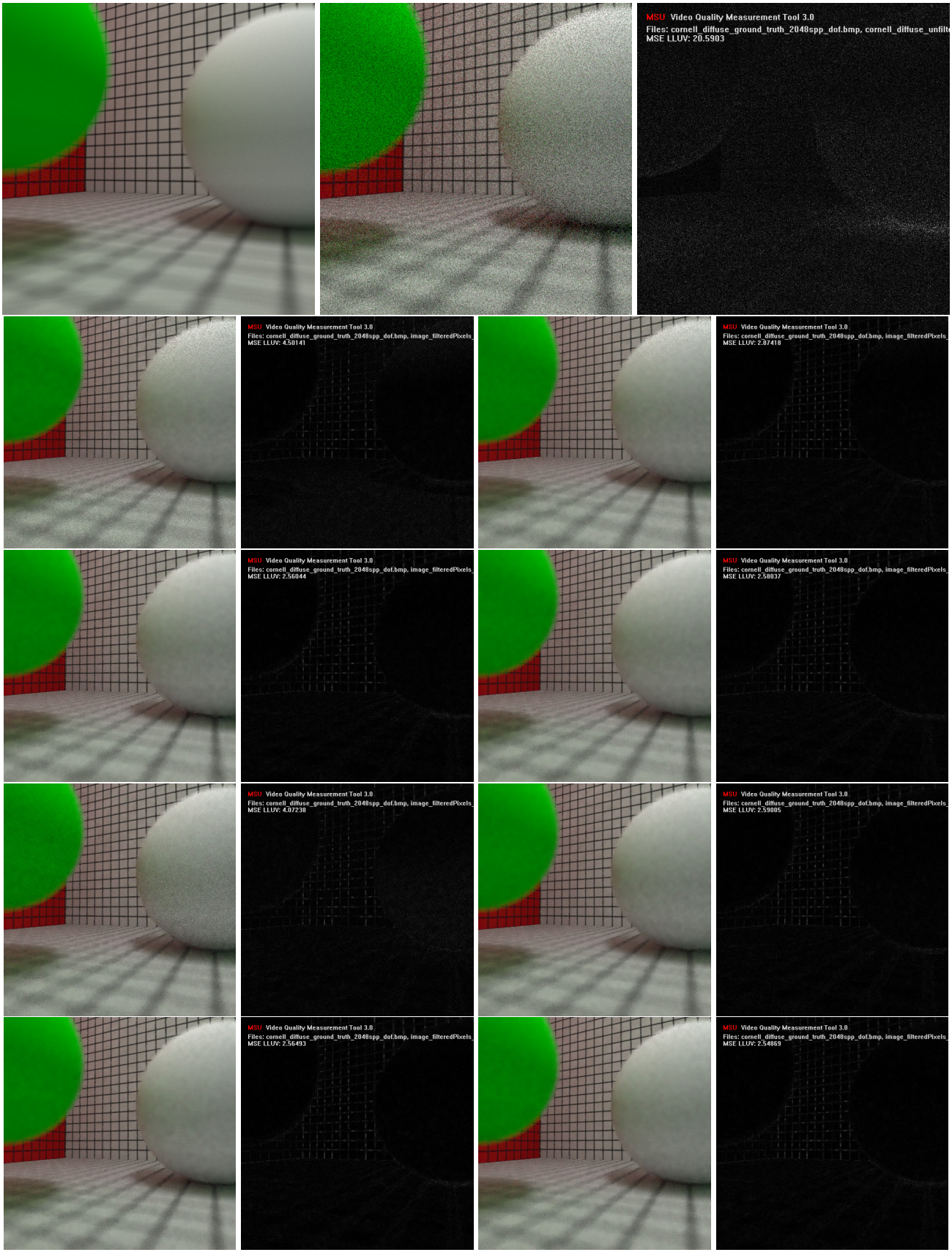


Figure B.5: Experiment 11×11, σ_{dc} and σ_{ic} with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

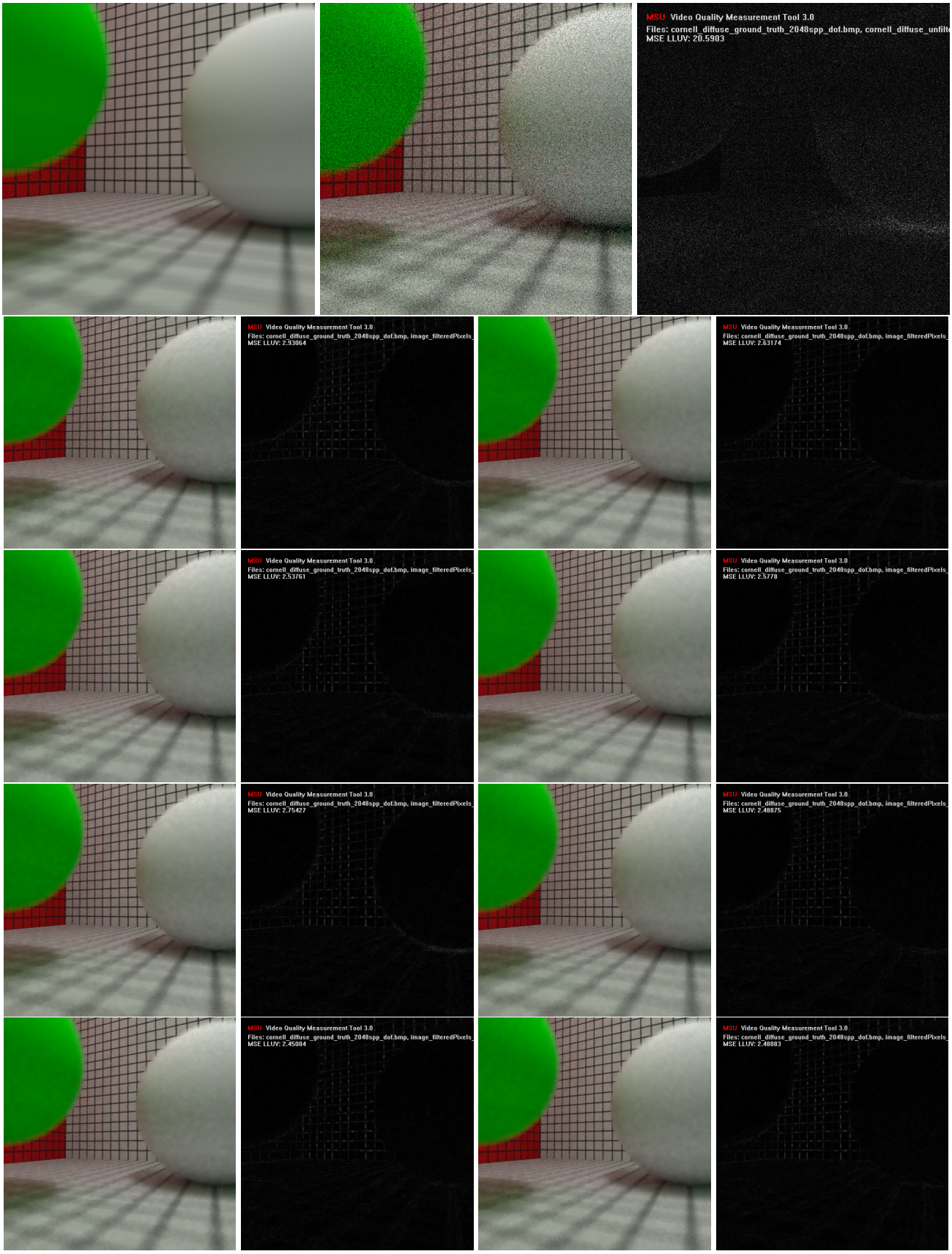


Figure B.6: Experiment 11×11, σ_f depth and σ_f normal with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

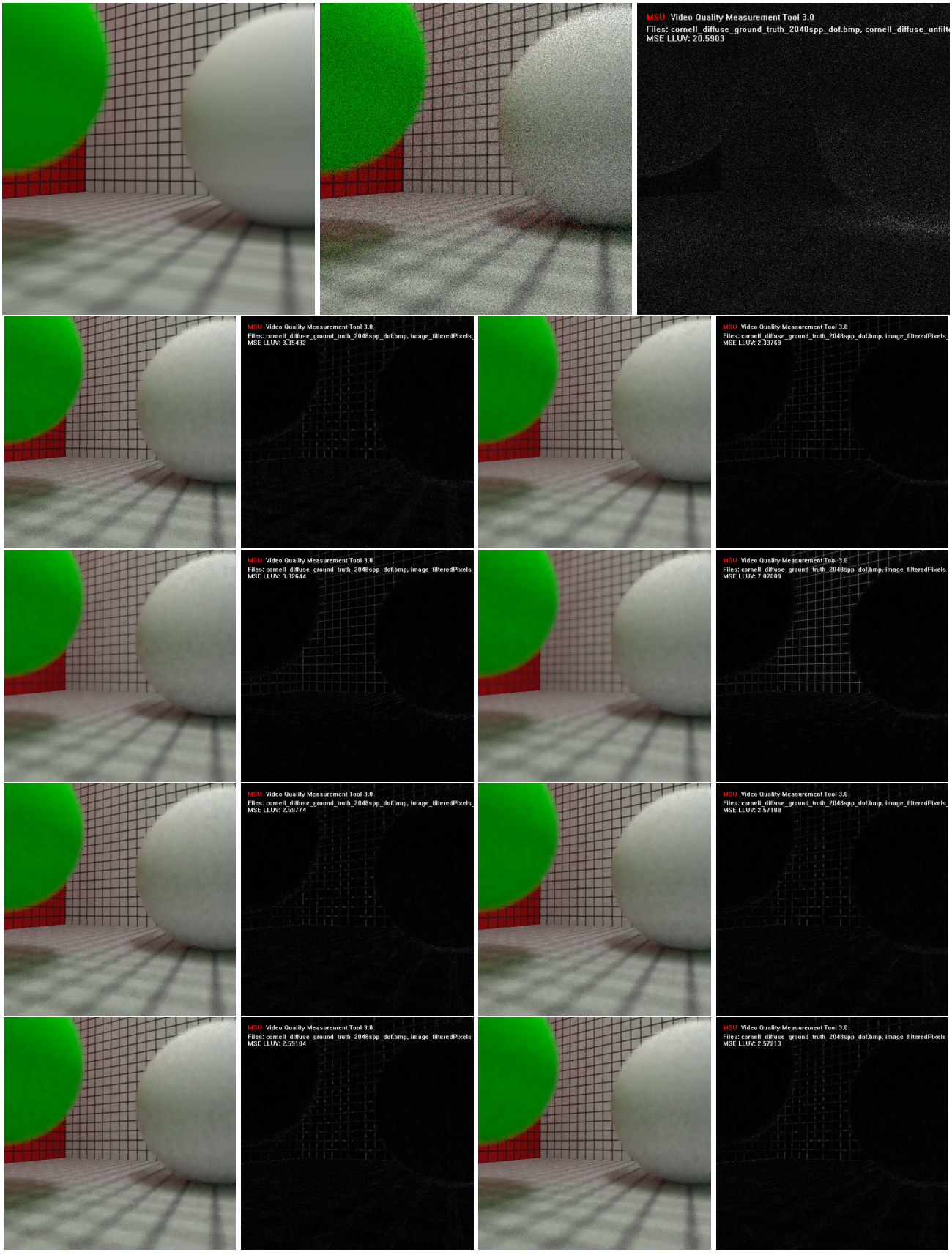


Figure B.7: Experiment 11×11, σ_f texture and σ_f direction with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

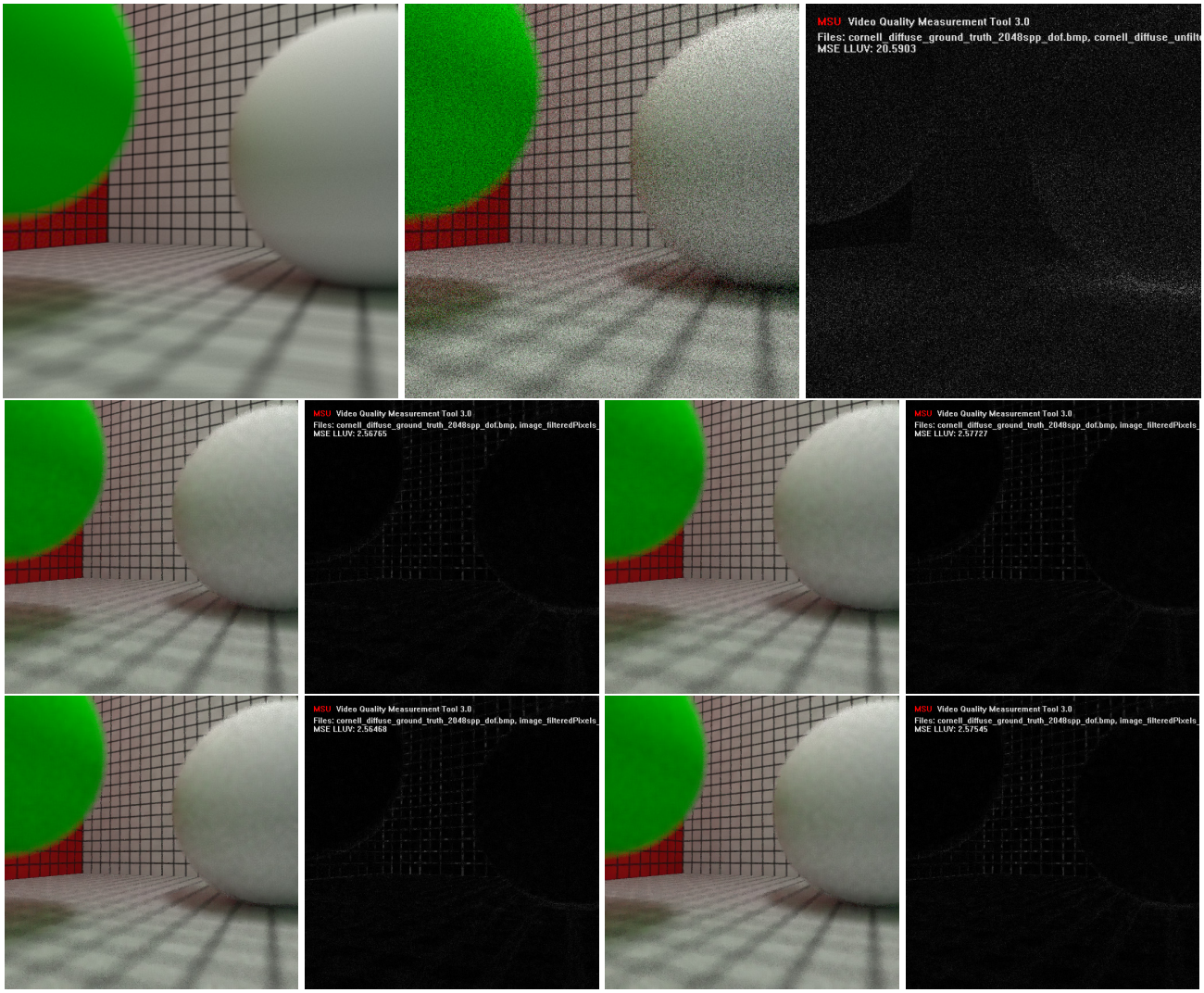


Figure B.8: Experiment 11×11, σ_f texture2 with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

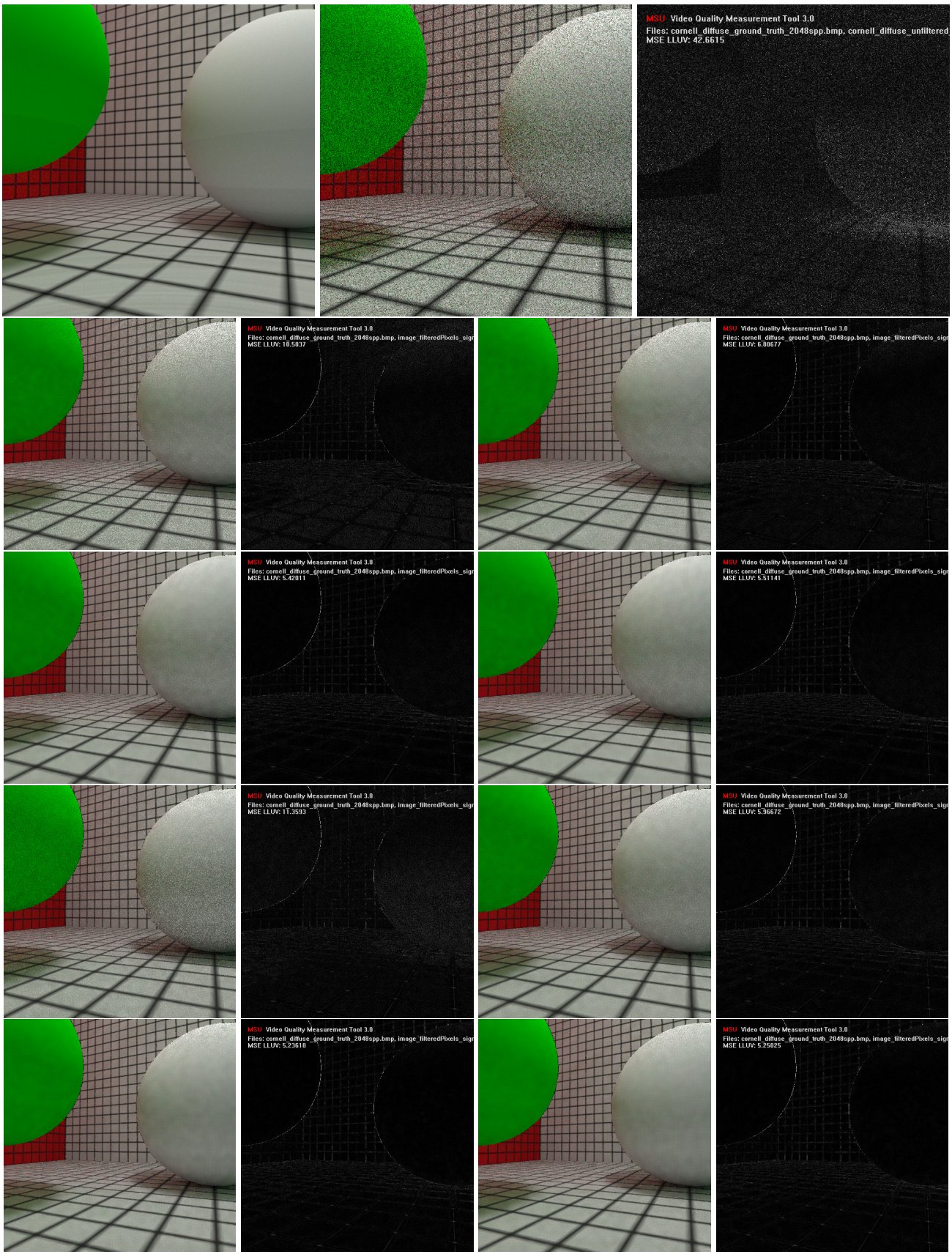


Figure B.9: Experiment 15×15, σ_{dc} and σ_{ic}. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

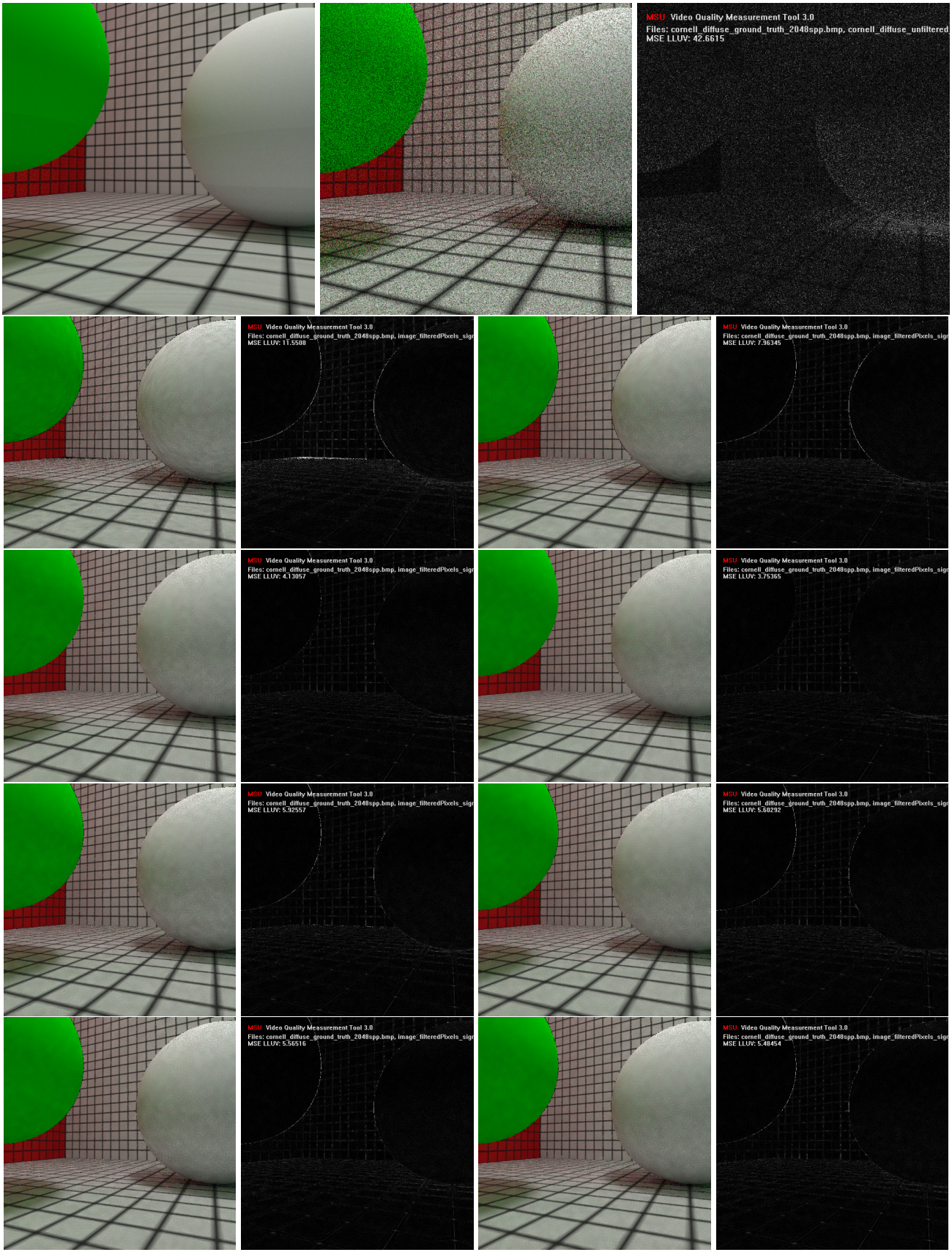


Figure B.10: Experiment 15×15, σ_f depth and σ_f normal. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

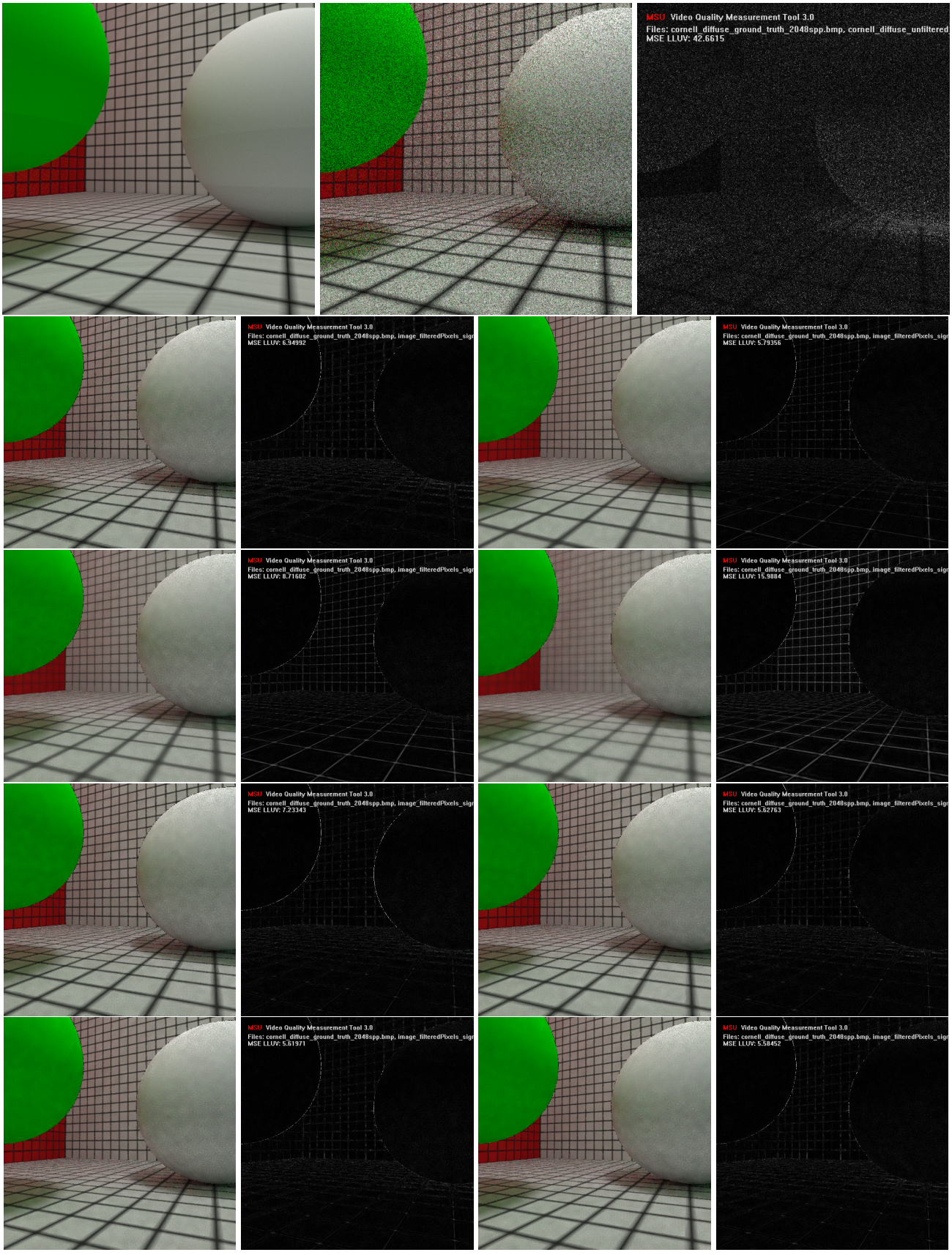


Figure B.11: Experiment 15×15, σ_f texture and σ_f direction. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

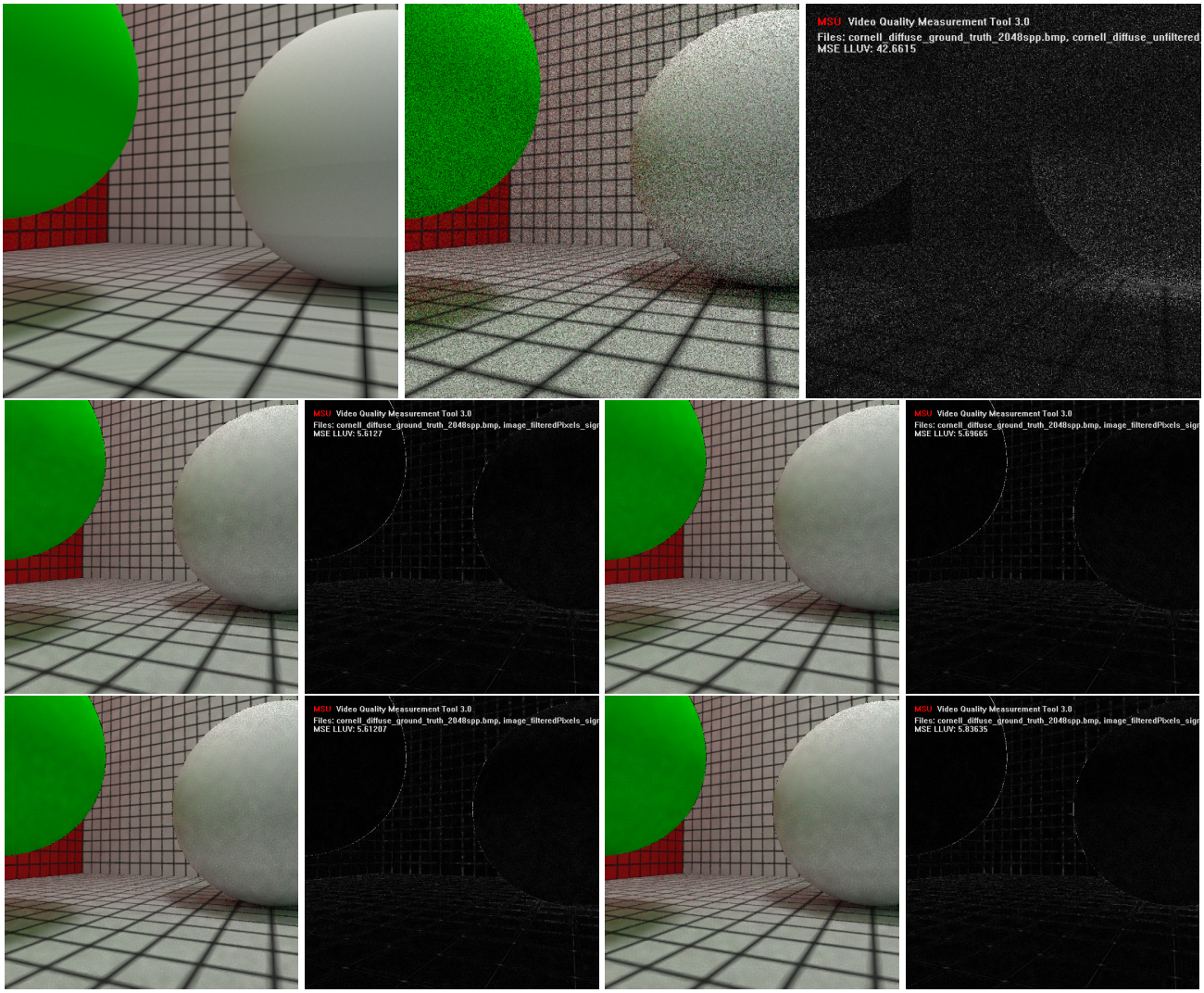


Figure B.12: Experiment 15×15, σ_f texture2. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

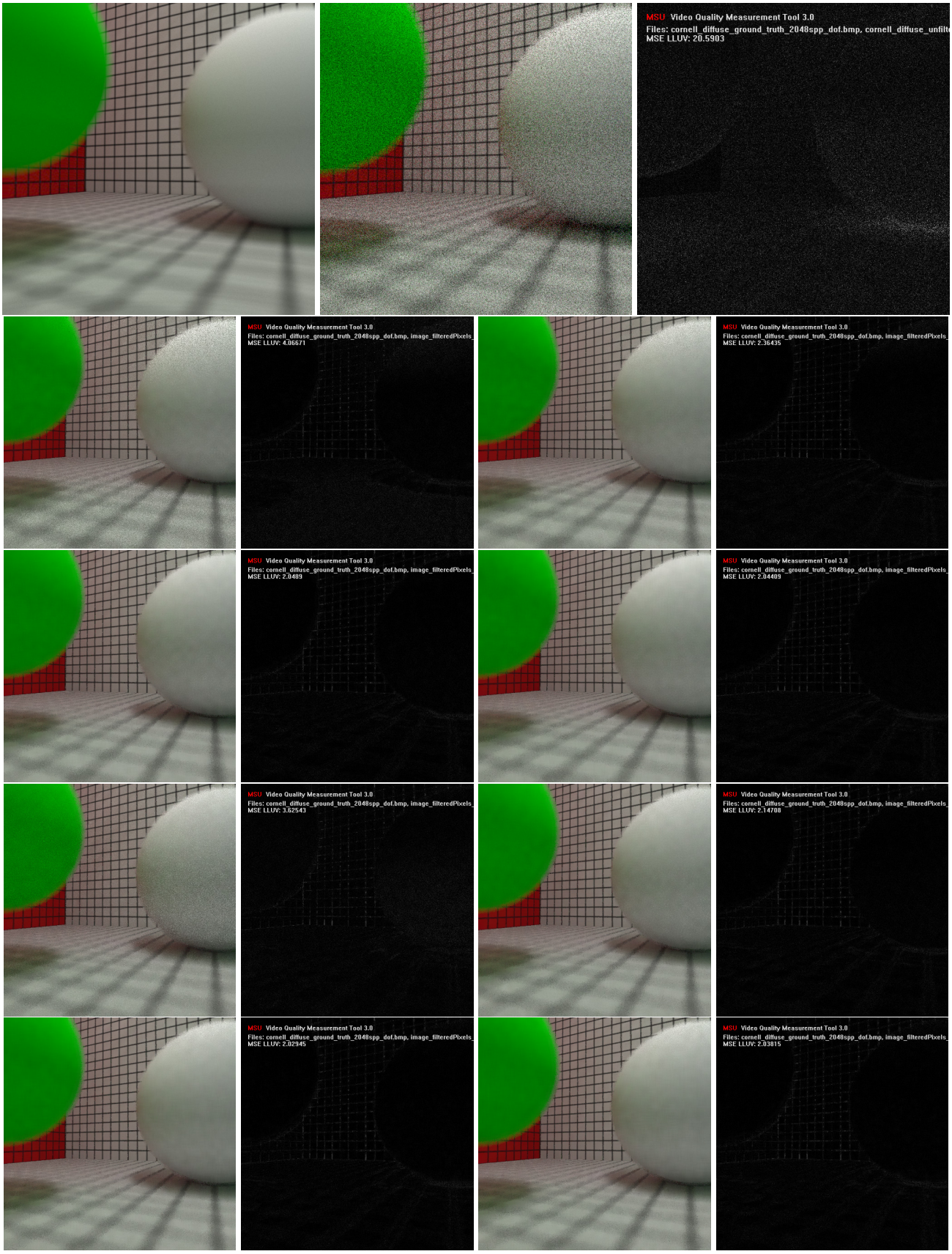


Figure B.13: Experiment 15×15, σ_{dc} and σ_{ic} with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

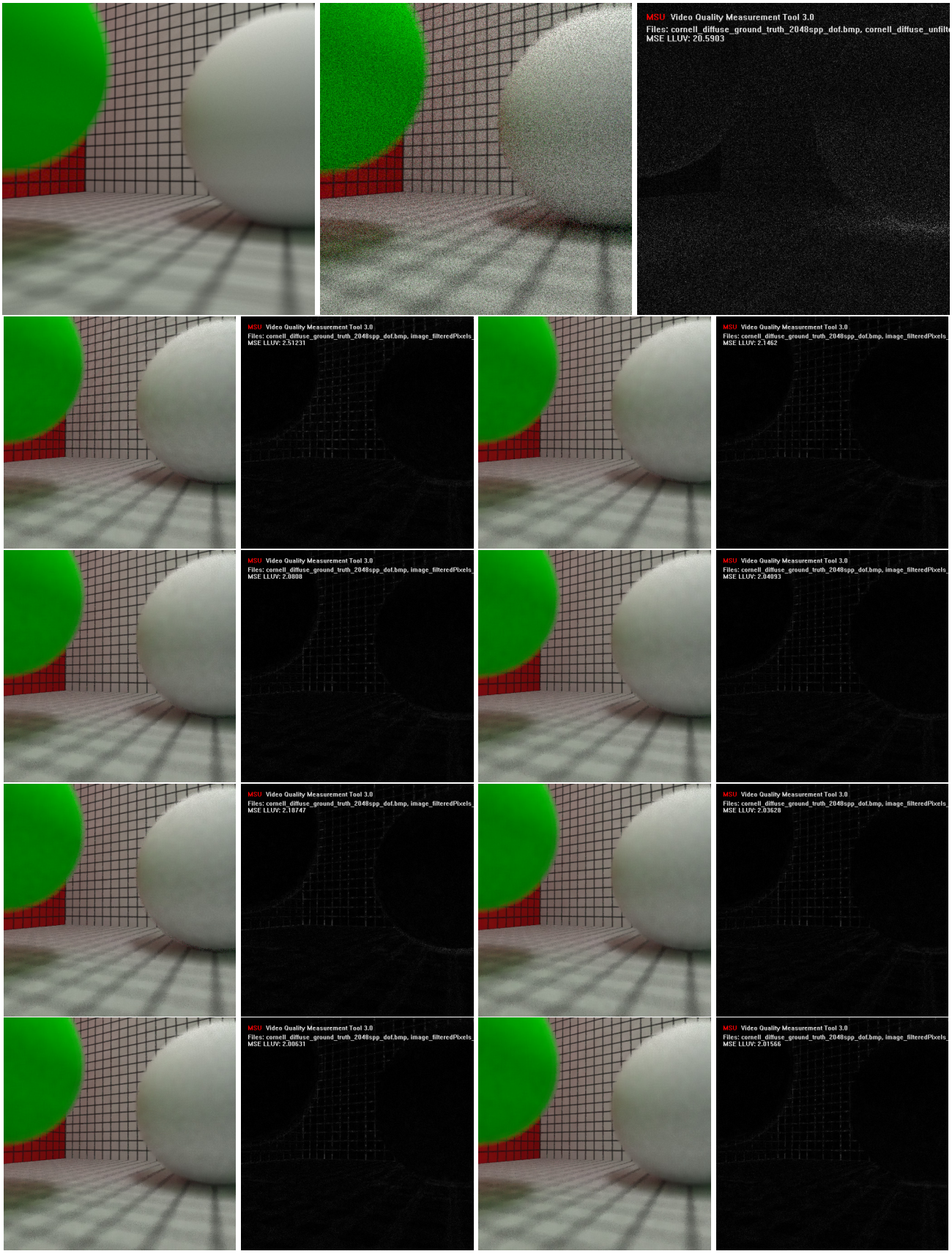


Figure B.14: Experiment 15×15, σ_f depth and σ_f normal with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

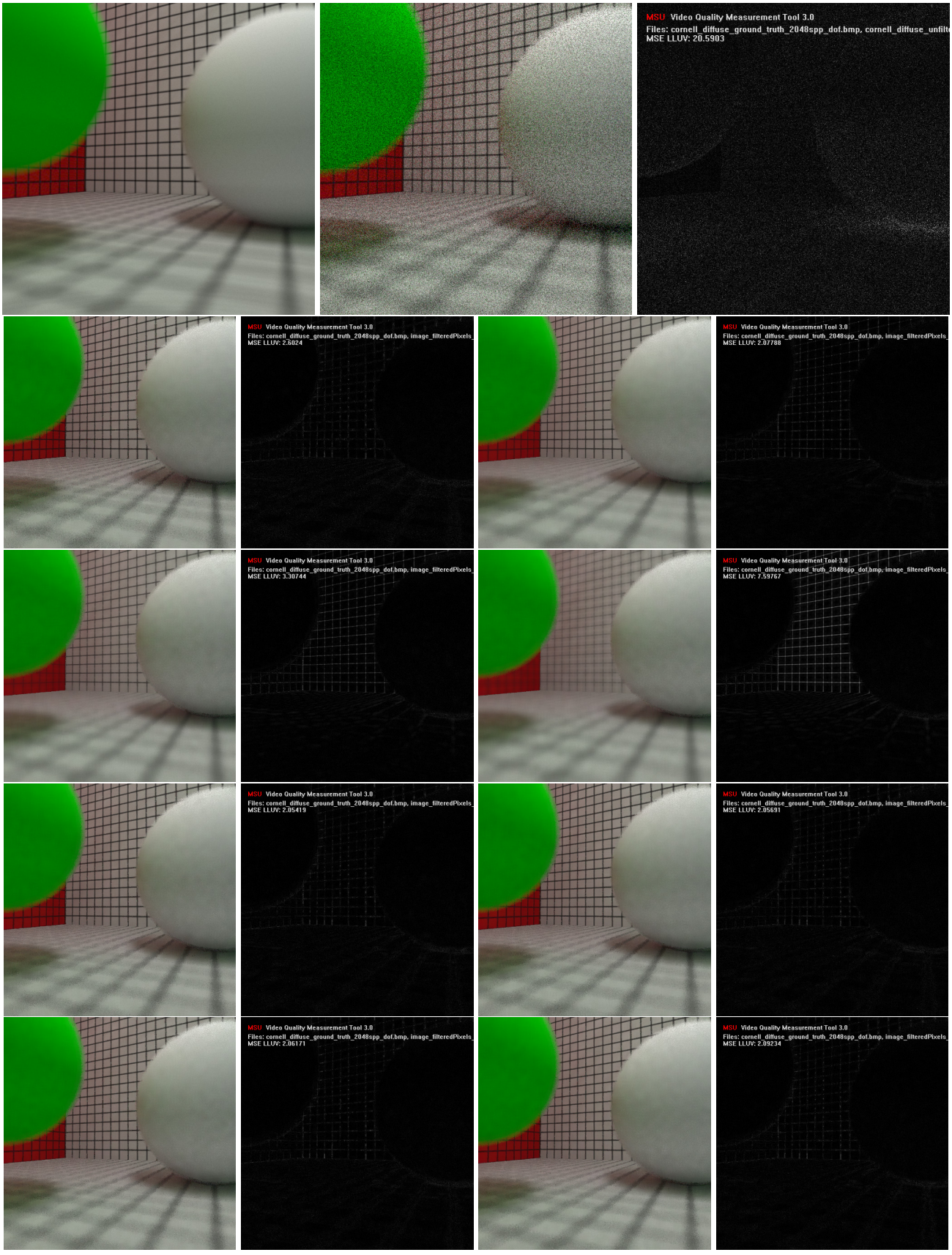


Figure B.15: Experiment 15×15, σ_f texture and σ_f direction with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

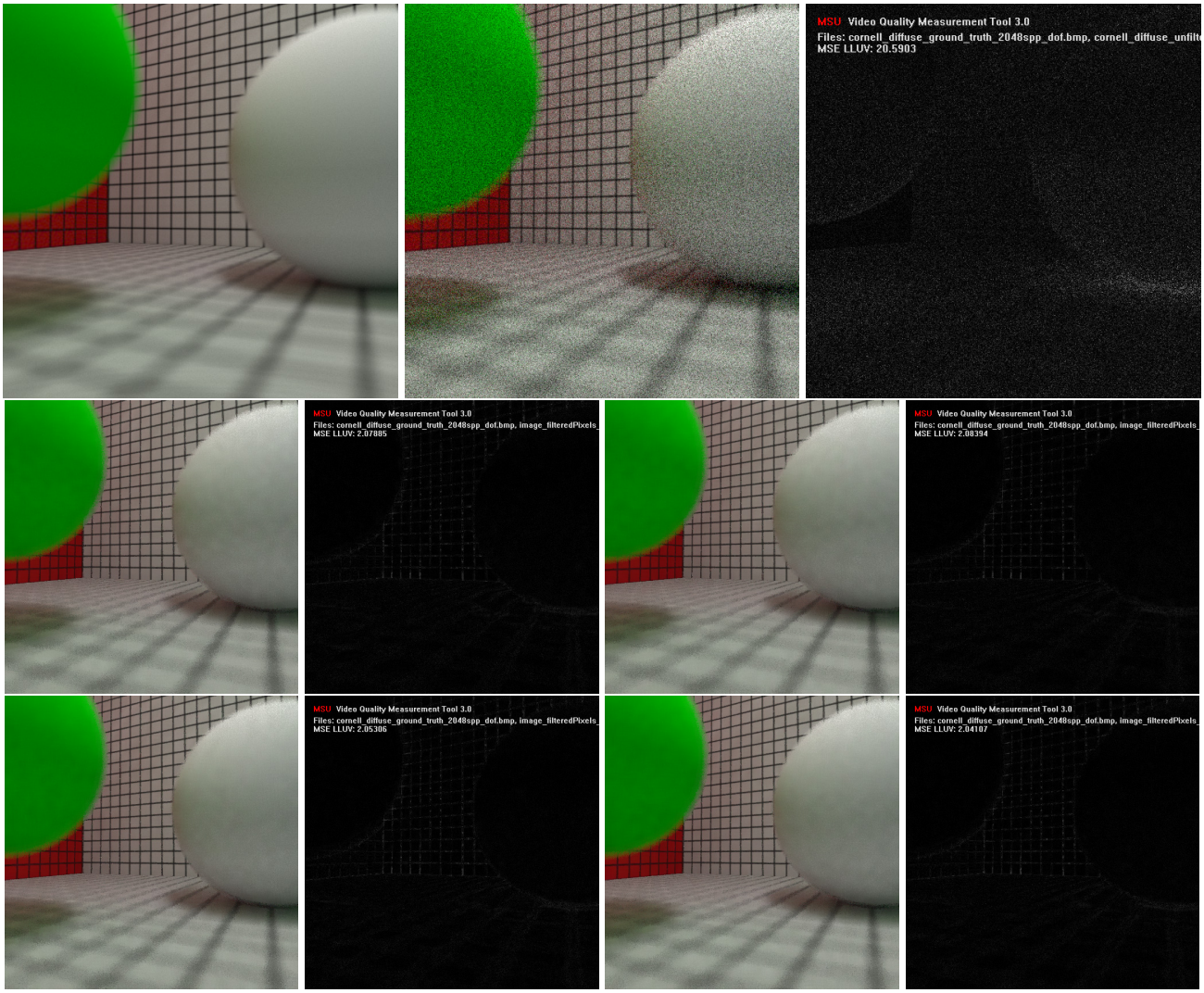


Figure B.16: Experiment 15×15, σ_f texture2 with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

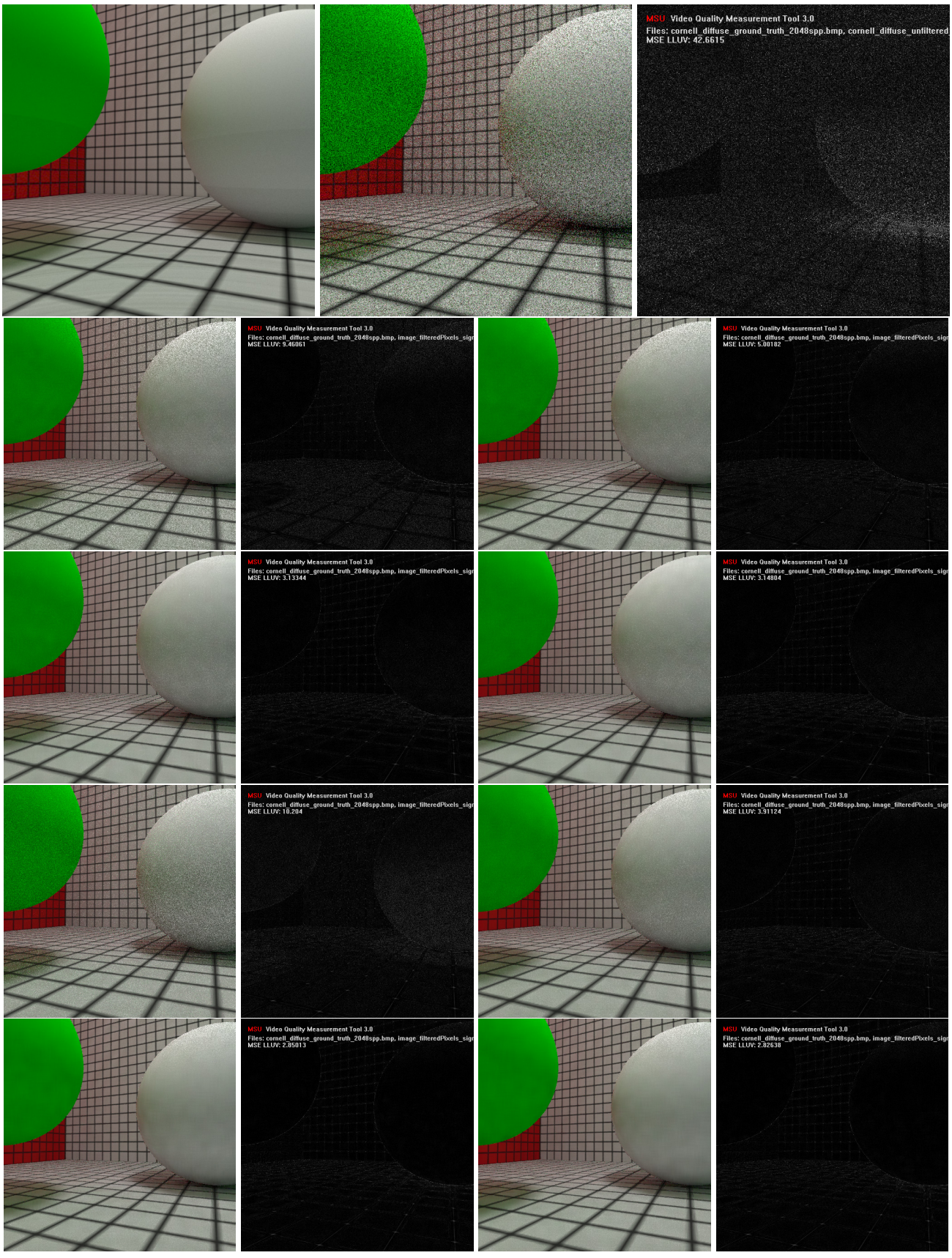


Figure B.17: Experiment 21×21, σ_{dc} and σ_{ic}. Experiment 21×21, σ_{dc} and σ_{ic}. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

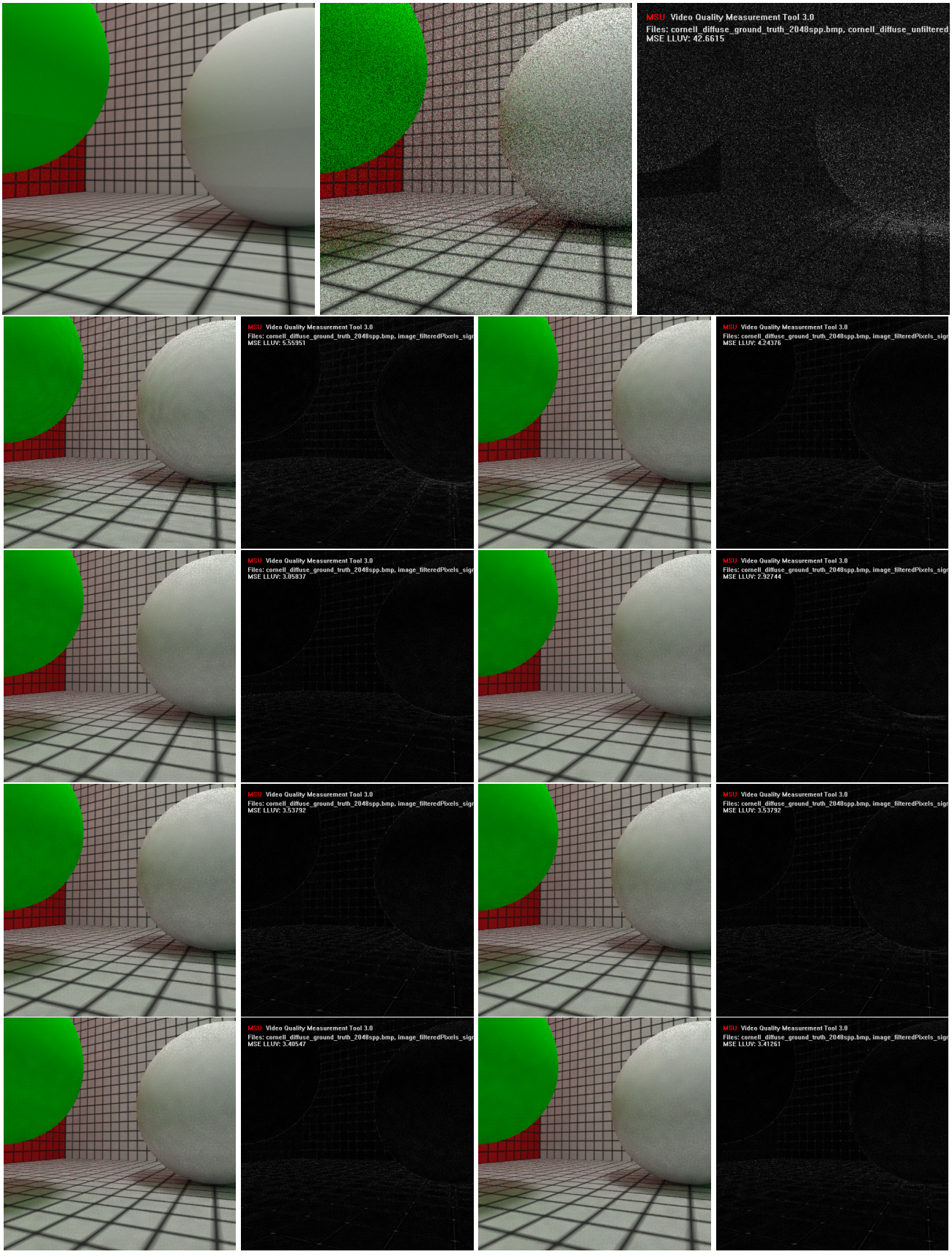


Figure B.18: Experiment 21×21, σ_f depth and σ_f normal. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

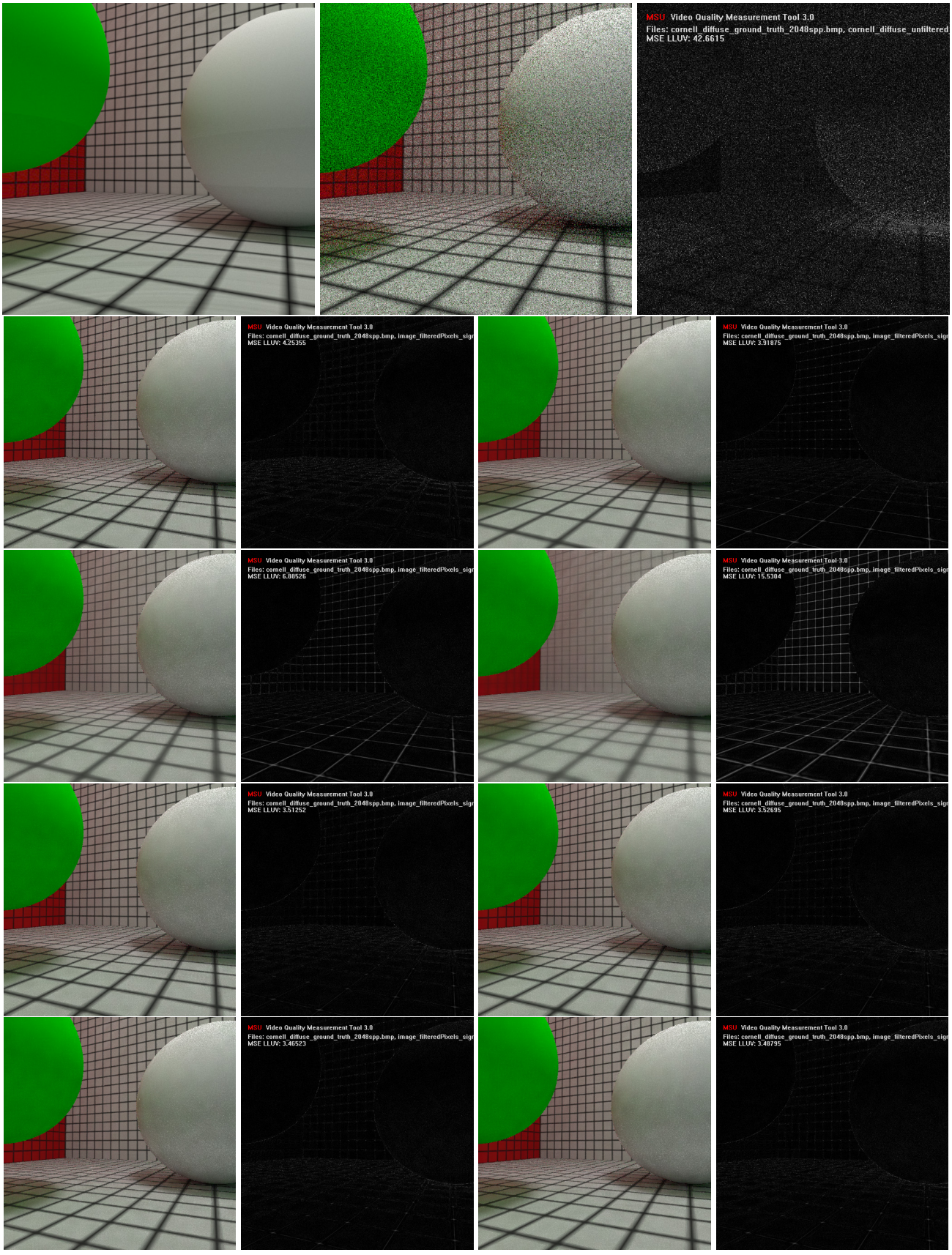


Figure B.19: Experiment 21×21, σ_f texture and σ_f direction. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

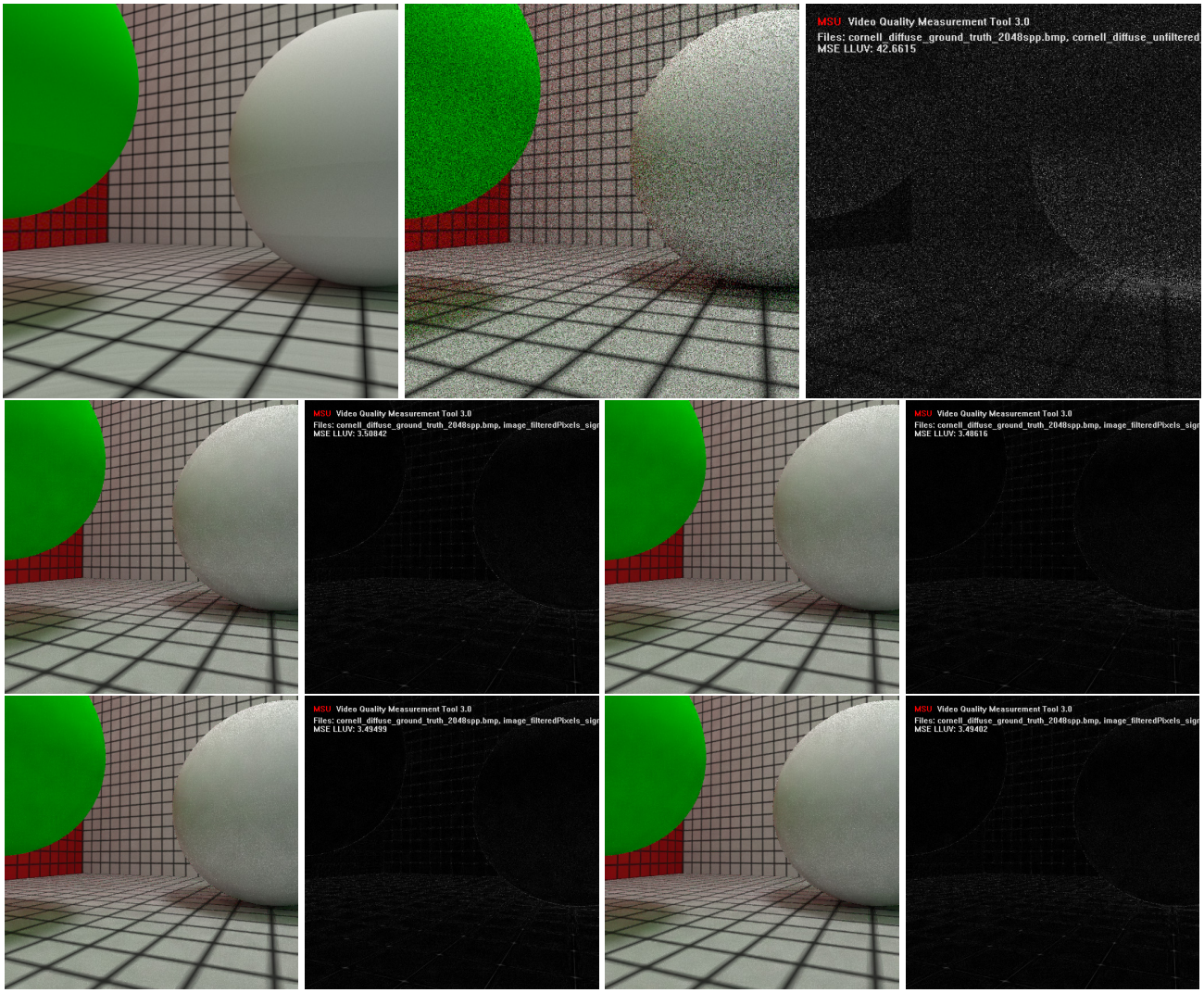


Figure B.20: Experiment 21×21, σ_f texture2. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

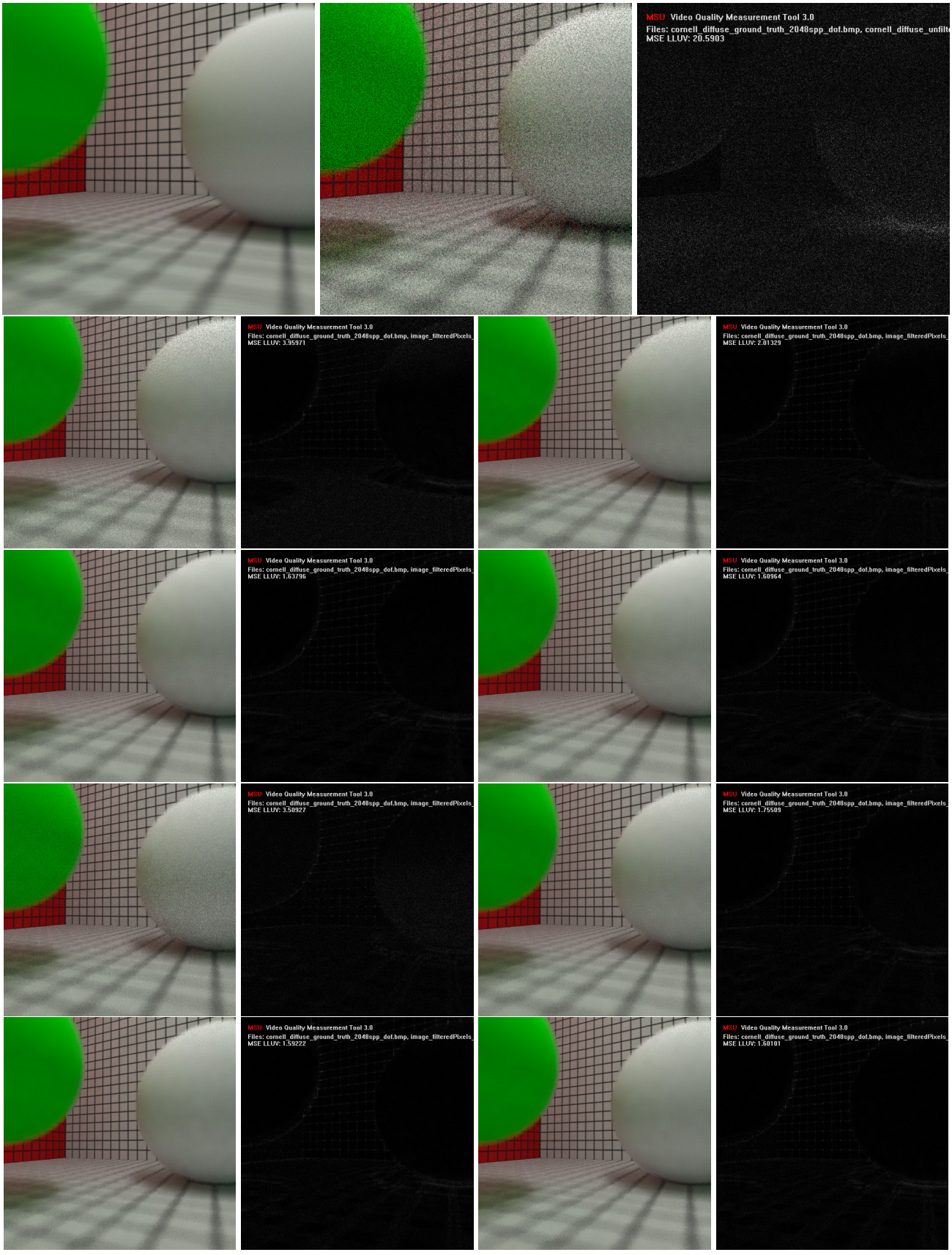


Figure B.21: Experiment 21×21, σ_{dc} and σ_{ic} with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_{dc}. Row 4 and 5: σ_{dc}.

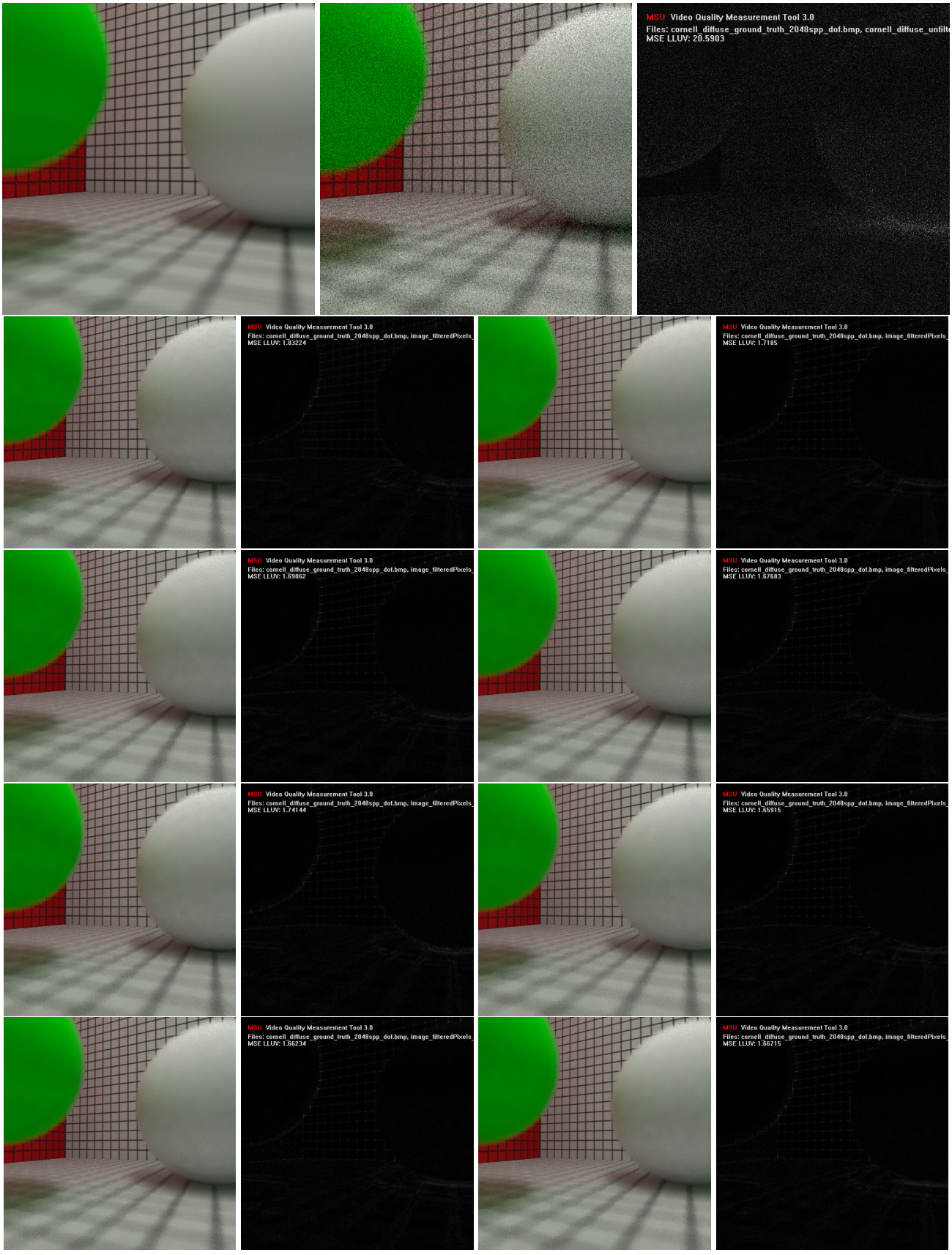


Figure B.22: Experiment 21×21, σ_f depth and σ_f normal with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f depth. Row 4 and 5: σ_f normal.

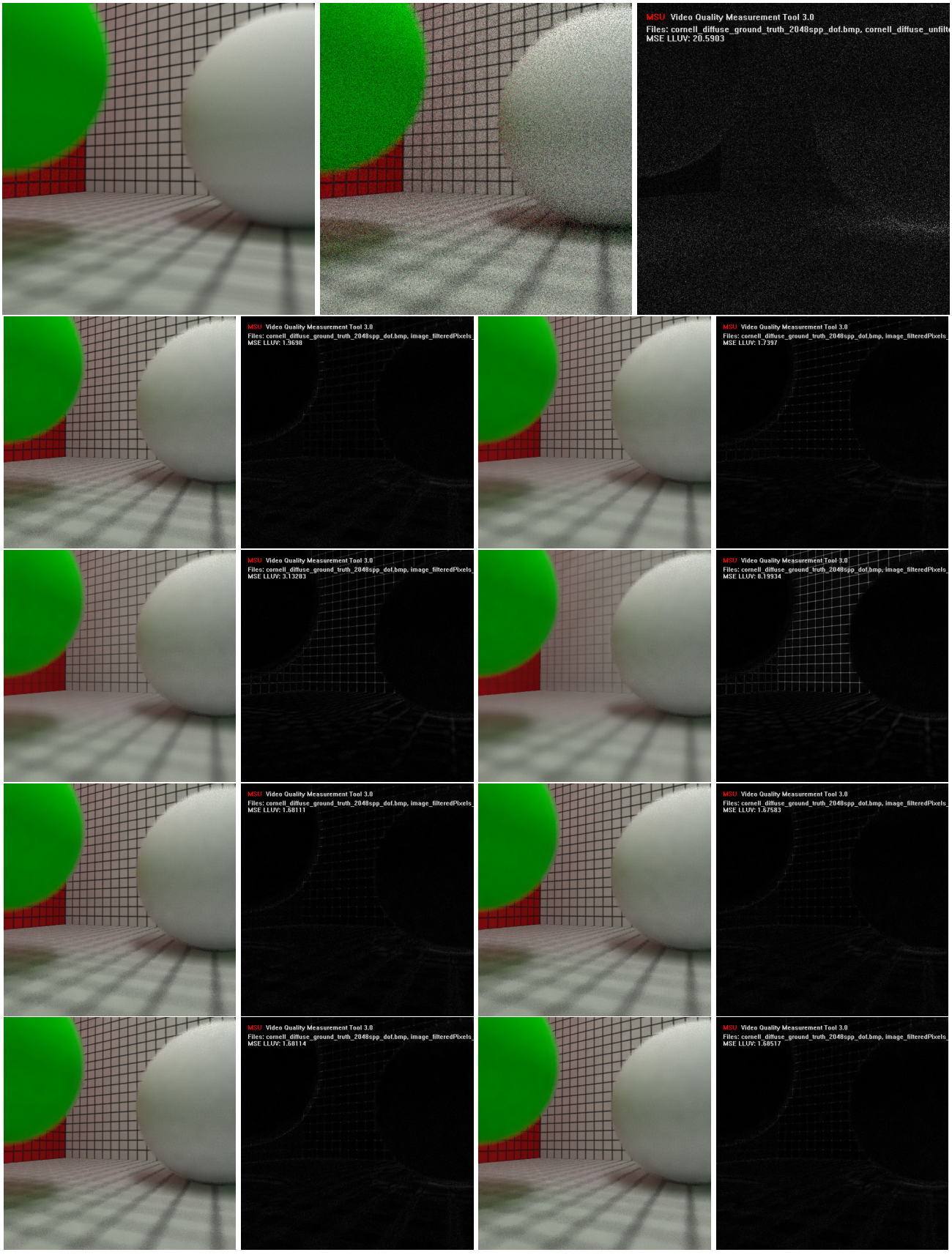


Figure B.23: Experiment 21×21, σ_f texture and σ_f direction with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture. Row 4 and 5: σ_f direction.

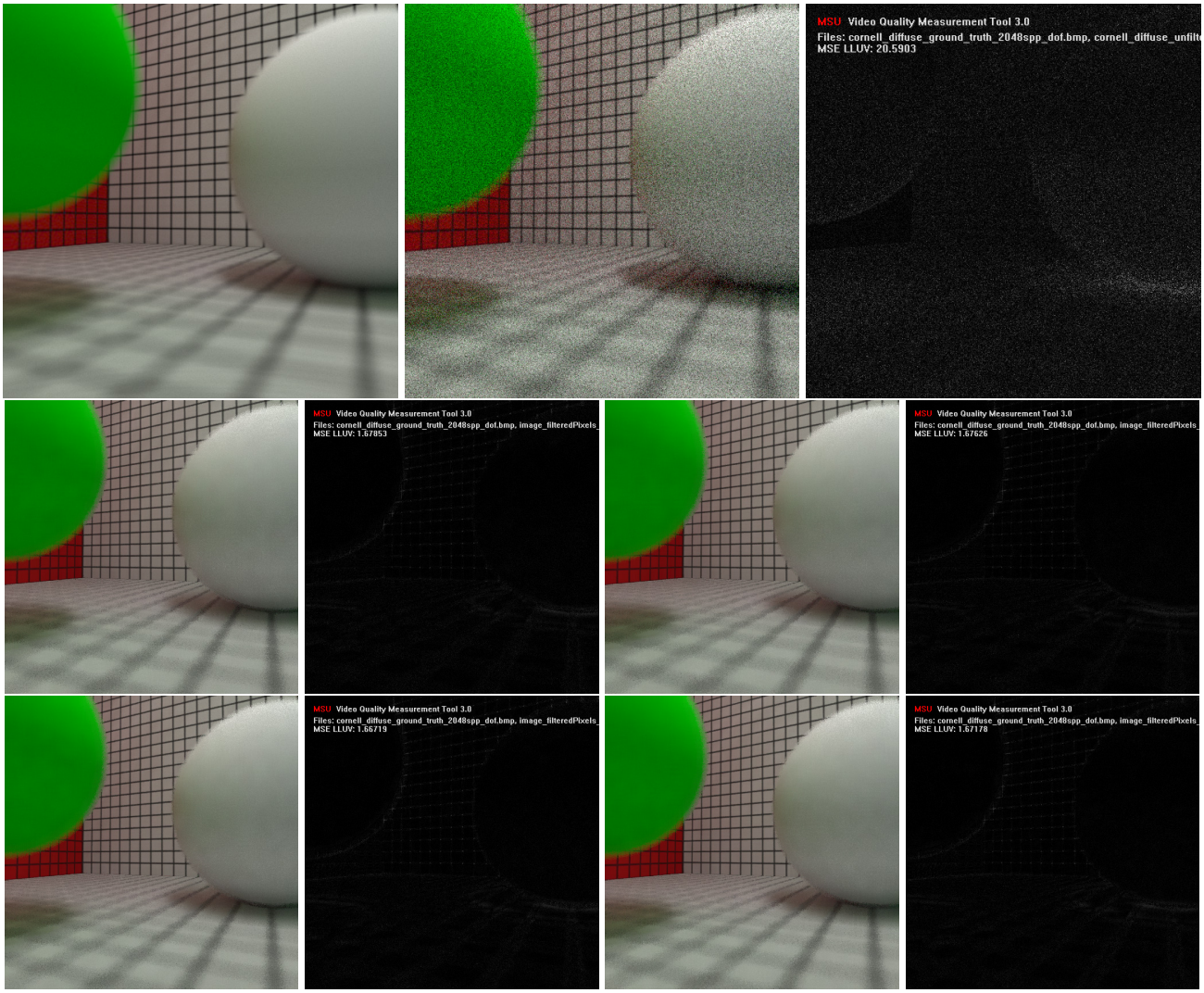


Figure B.24: Experiment 21×21, σ_f texture2 with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f texture2.

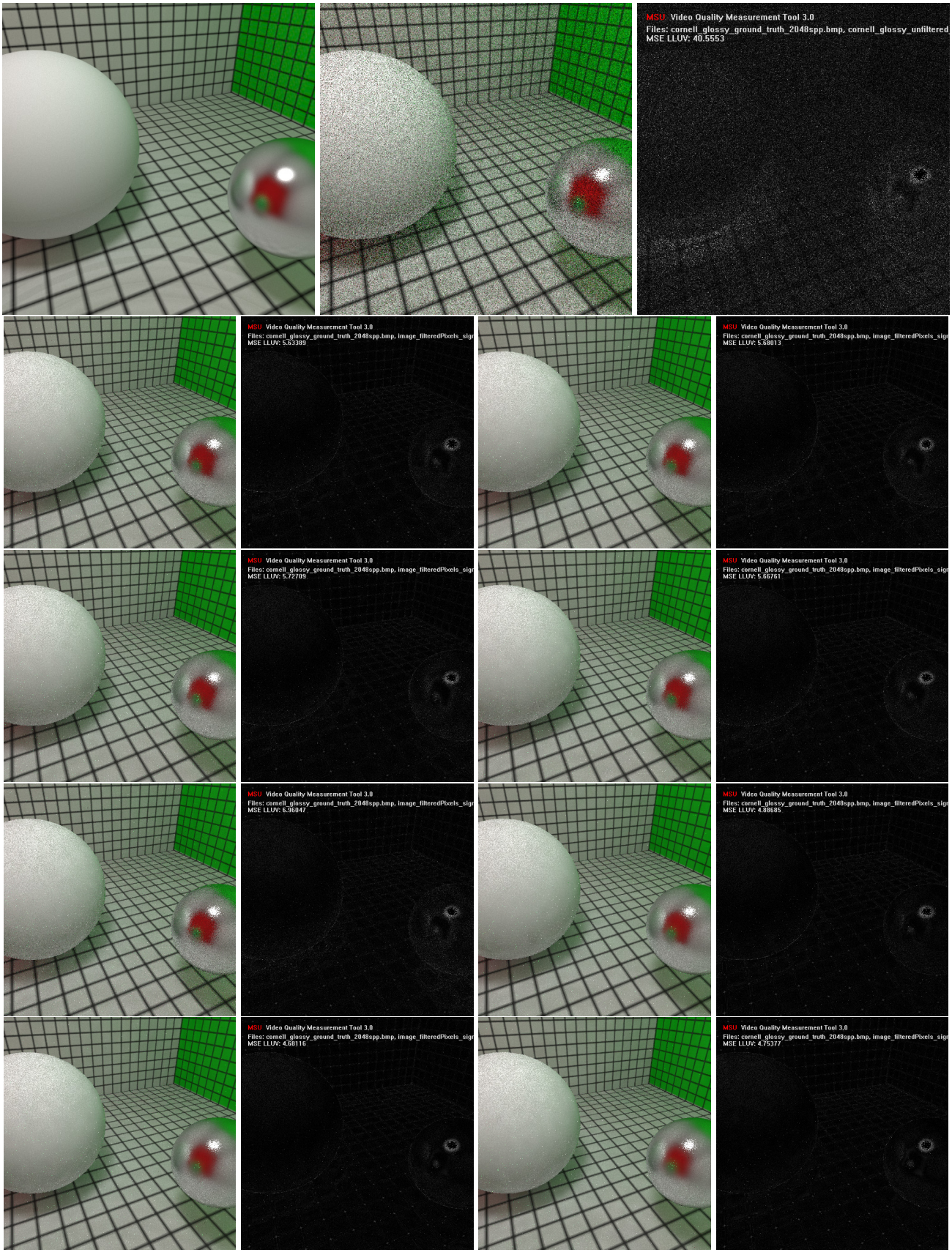


Figure B.25: Experiment 21×21, σ_f direction and σ_f texture2. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f direction. Row 4 and 5: σ_f texture2.

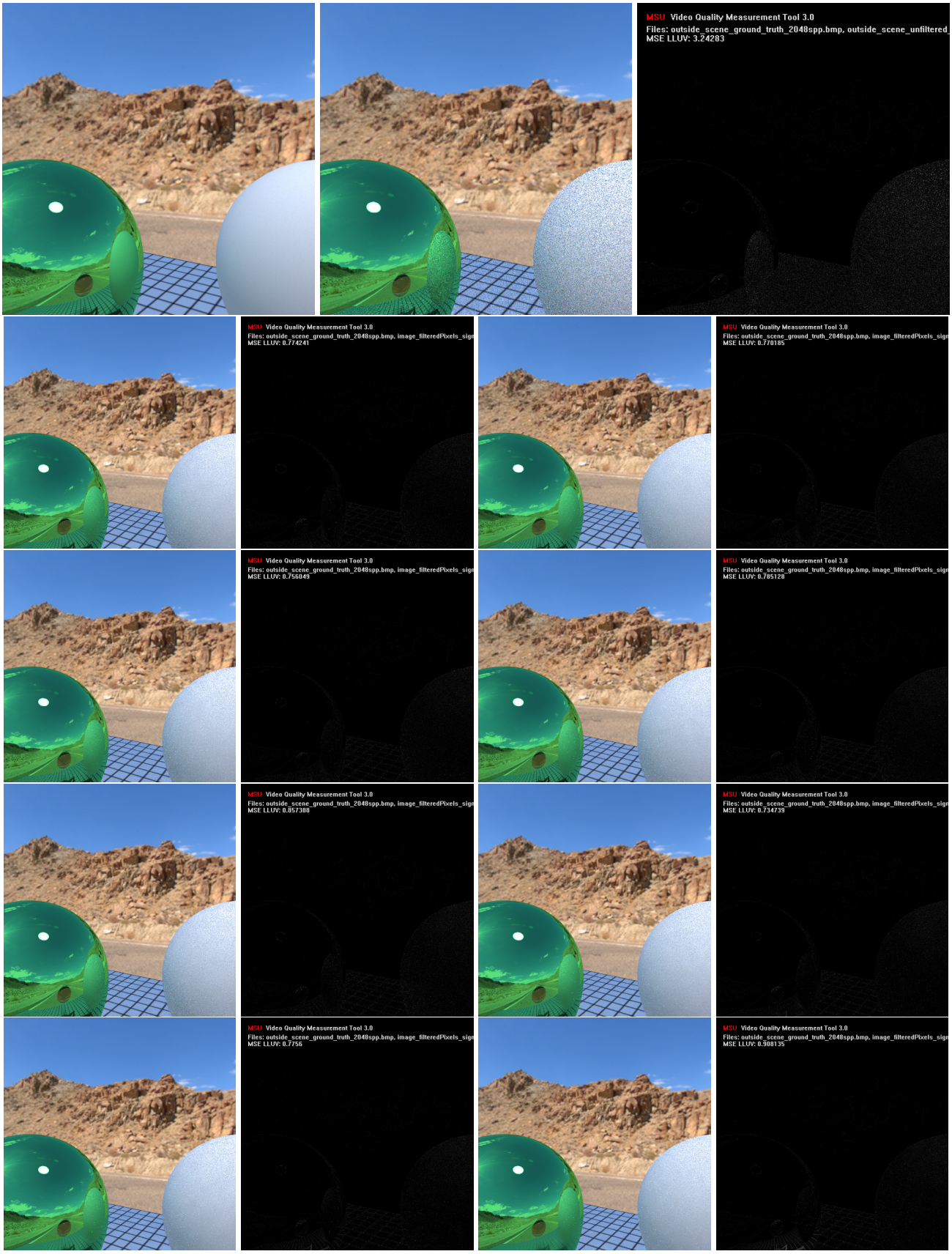


Figure B.26: Experiment 21×21, σ_f direction and σ_f texture2. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f direction. Row 4 and 5: σ_f texture2.

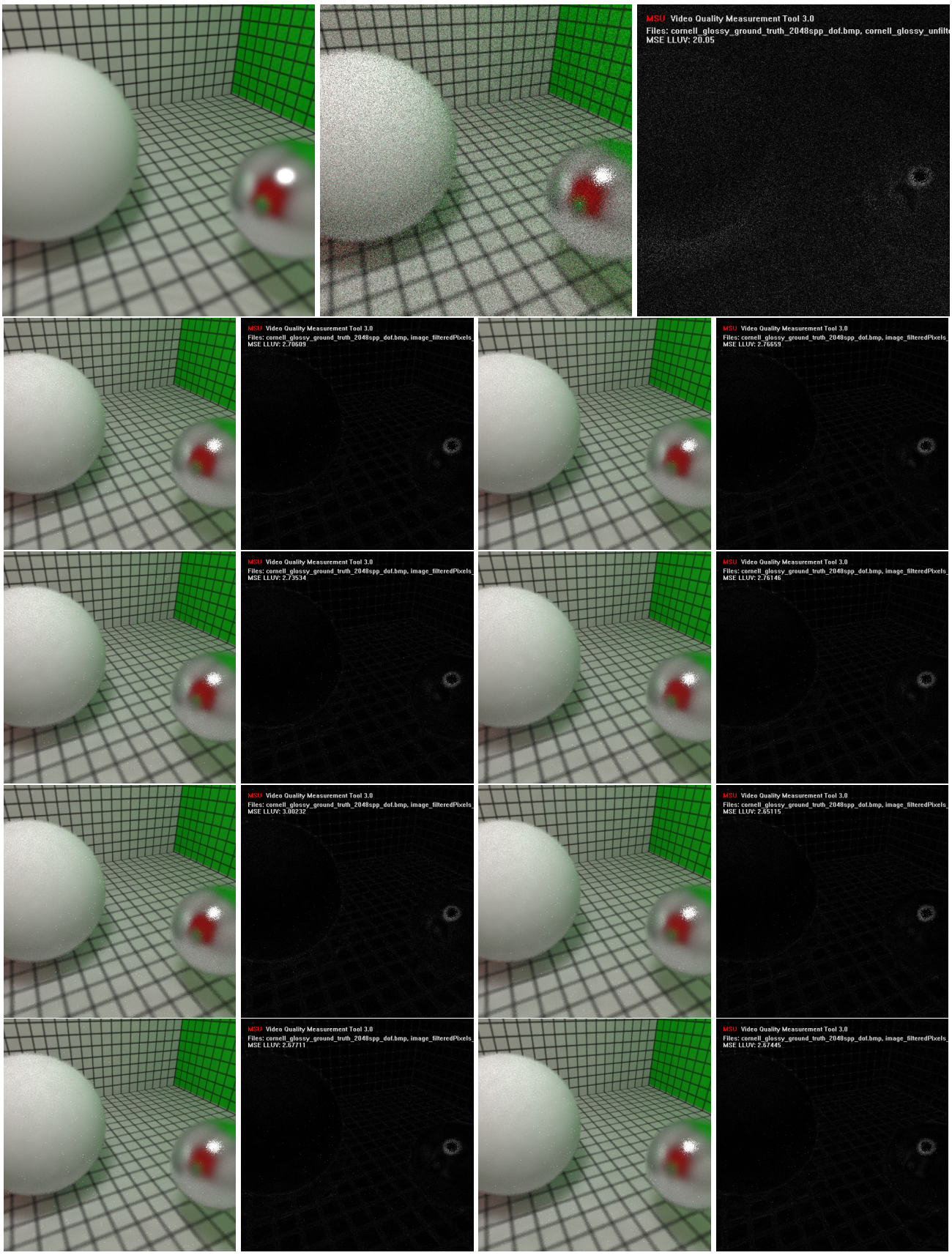


Figure B.27: Experiment 21×21, σ_f direction and σ_f texture2 with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f direction. Row 4 and 5: σ_f texture2.

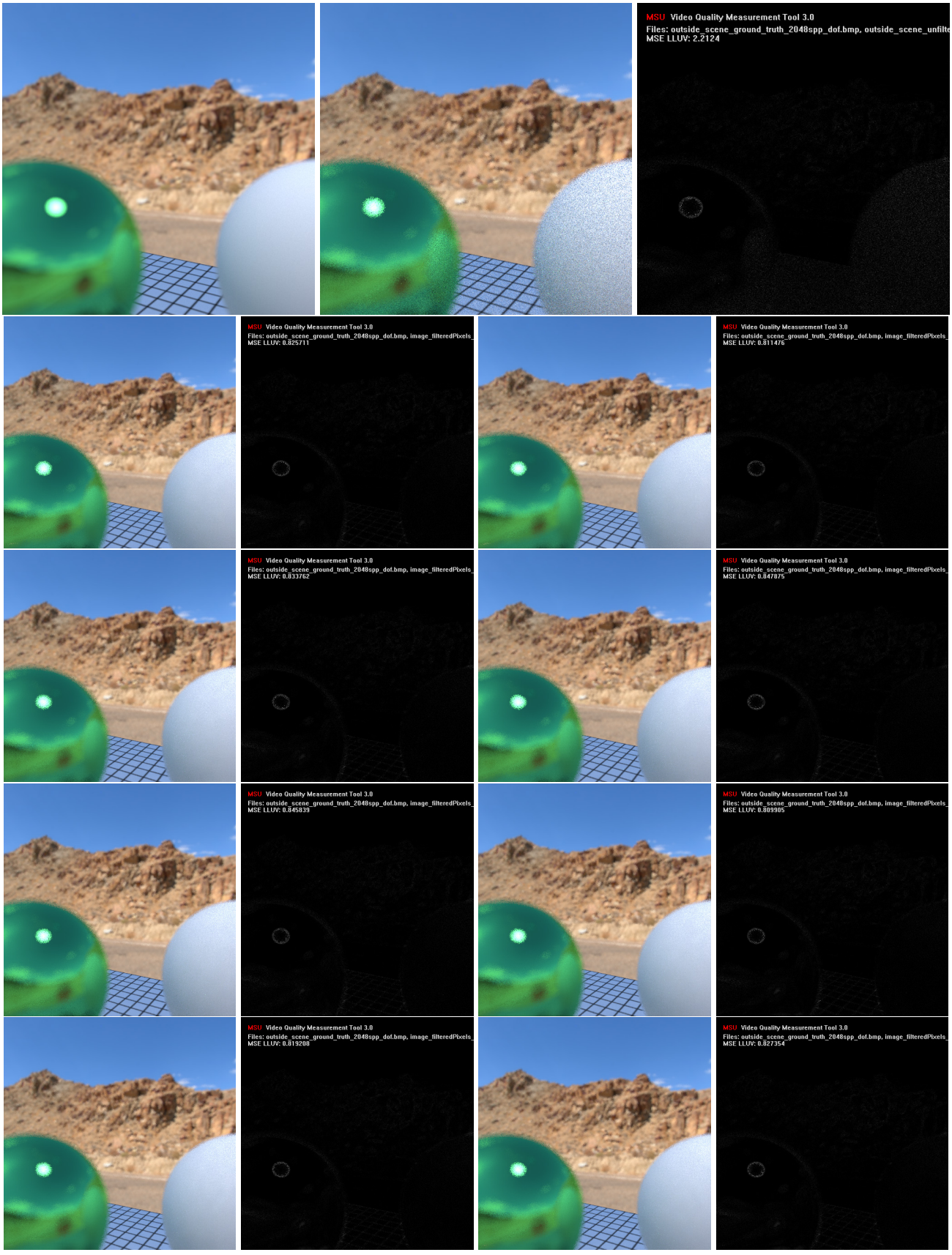


Figure B.28: Experiment 21×21, σ_f direction and σ_f texture2 with depth of field. Row 1: Ground-truth, Unfiltered and Unfiltered MSE. Row 2 and 3: σ_f direction. Row 4 and 5: σ_f texture2.

Appendix C

Hypothesis 3 figures

The three test scenes used to find the optimal parameters are used to test hypothesis 3. Each scene is filtered with filter sizes: 11×11, 15×15, 21×21 and 55×55 without and with depth of field. The following figures show each test scene in the following order:

- The results of each test scene with a filter size of 11×11 without and with dof
  First row: Diffuse scene filtered, filtered MSE,filtered with dof, filtered MSE
  Second row: Glossy scene filtered , filtered MSE, filtered with dof, filtered MSE
  Third row: Outside scene filtered, filtered MSE, filtered with dof, filtered MSE
- The results of each test scene with a filter size of 15×15 without and with dof
  First row: Diffuse scene filtered, filtered MSE, filtered with dof, filtered MSE
  Second row: Glossy scene filtered, filtered MSE, filtered with dof, filtered MSE
  Third row: Outside scene filtered, filtered MSE, filtered with dof, filtered MSE
- The results of each test scene with a filter size of 21×21 without and with dof
  First row: Diffuse scene filtered, filtered MSE, filtered with dof, filtered MSE
  Second row: Glossy scene filtered, filtered MSE, filtered with dof, filtered MSE
  Third row: Outside scene filtered, filtered MSE, filtered with dof, filtered MSE
- The results of each test scene with a filter size of 55×55 without and with dof
  First row: Diffuse scene filtered, filtered MSE, filtered with dof, filtered MSE
  Second row: Glossy scene filtered, filtered MSE, filtered with dof, filtered MSE
  Third row: Outside scene filtered, filtered MSE, filtered with dof, filtered MSE

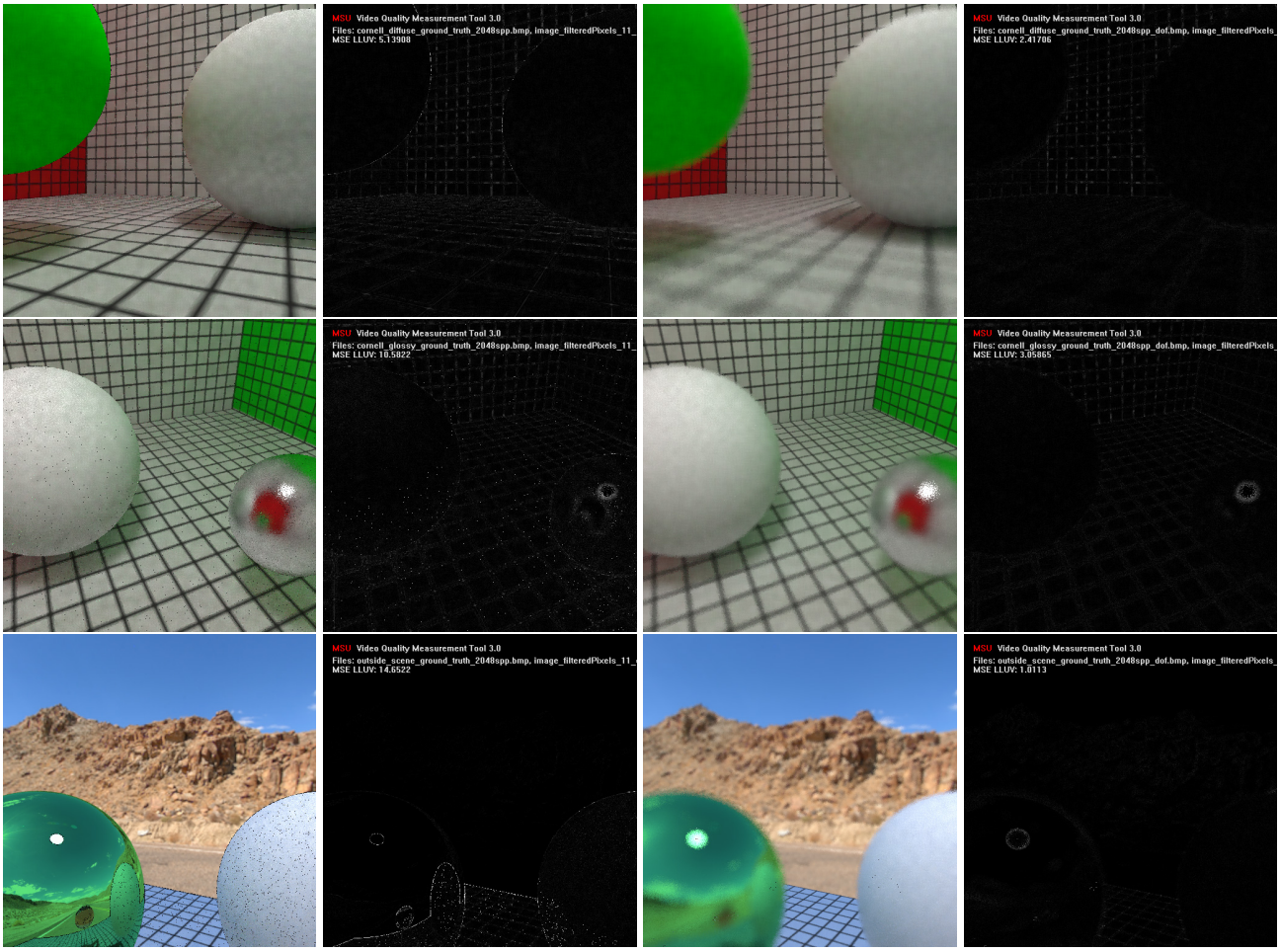


Figure C.1: Filter size 11×11

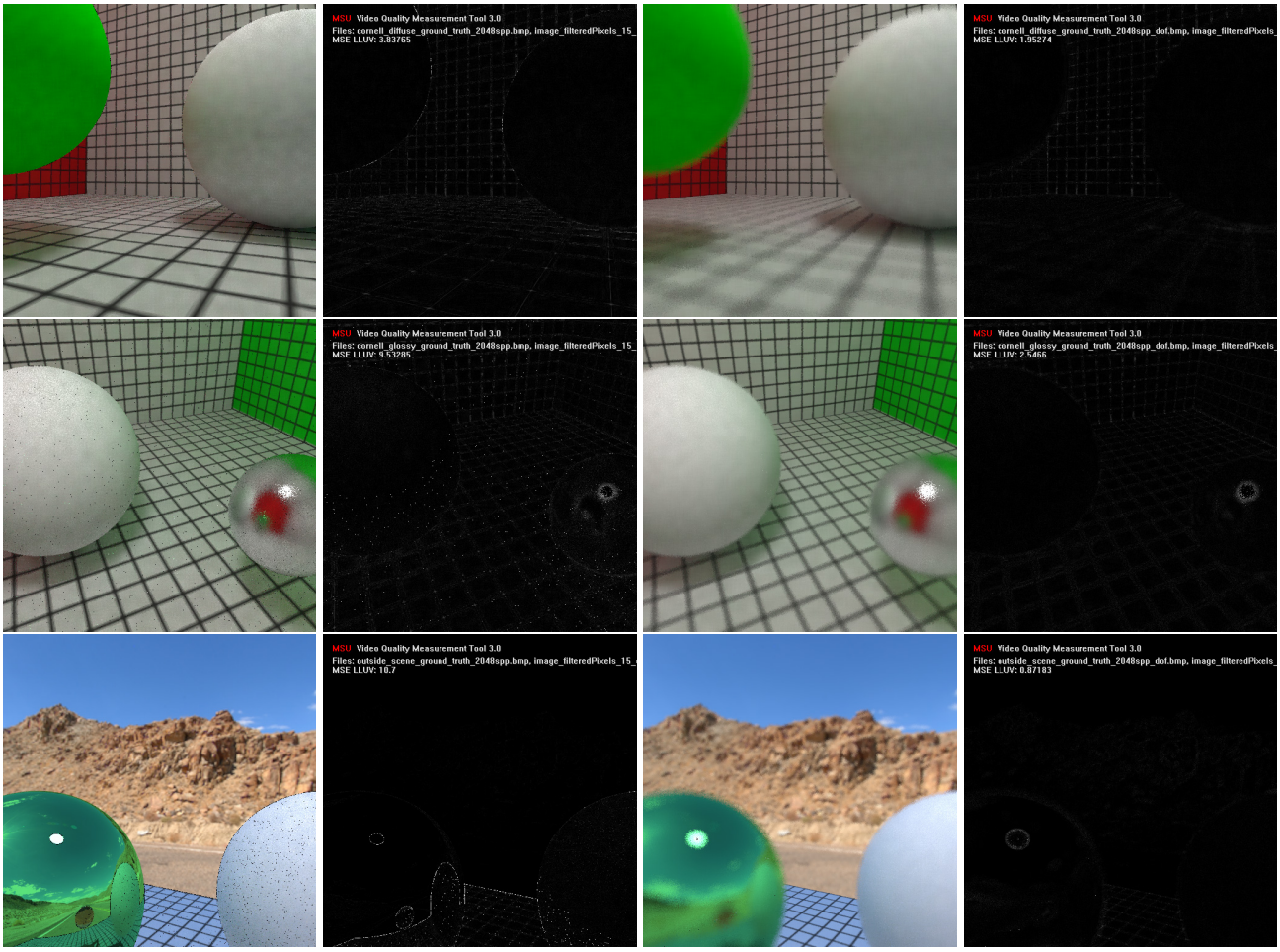


Figure C.2: Filter size 15×15

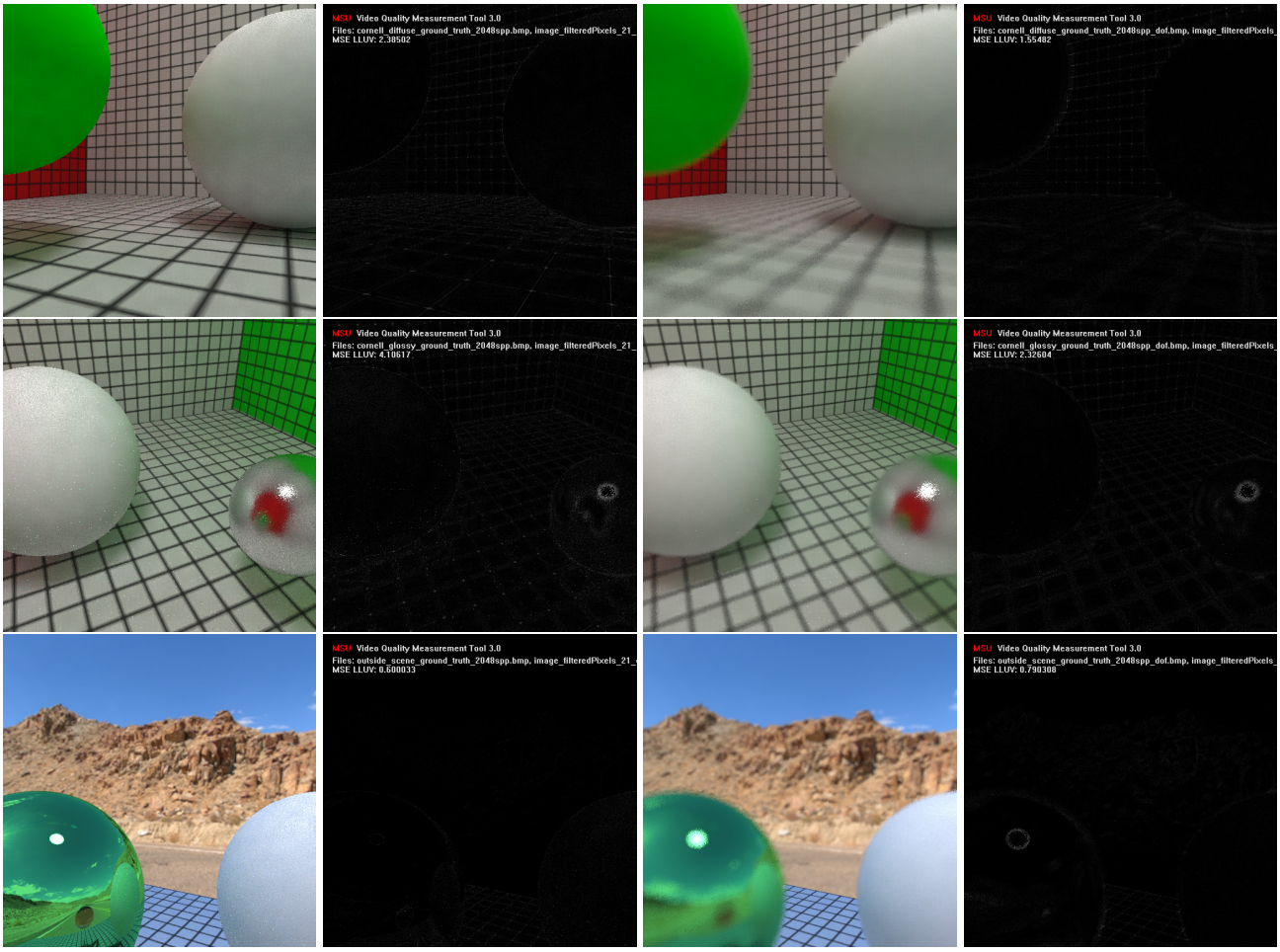


Figure C.3: Filter size 21×21

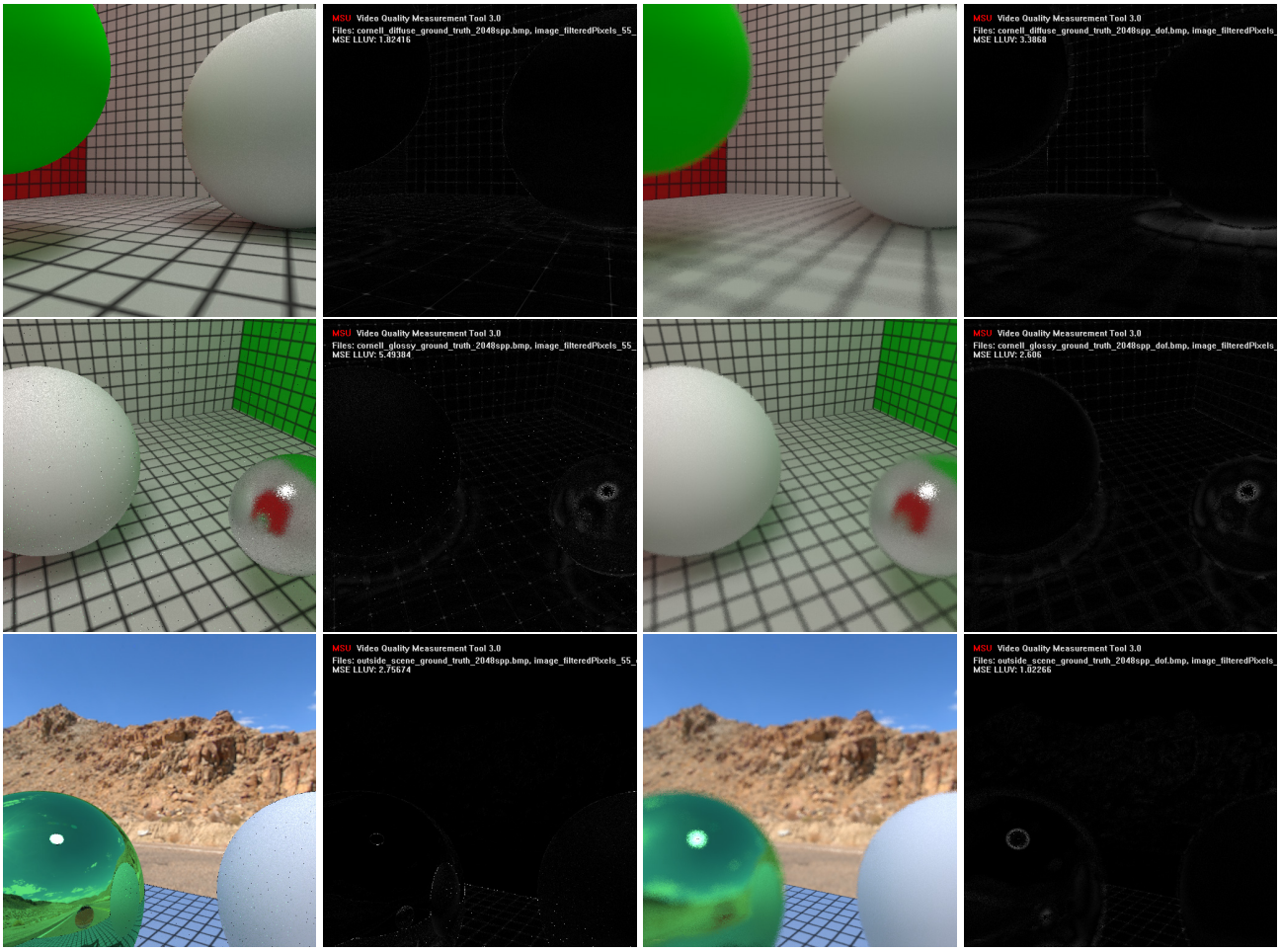


Figure C.4: Filter size 55×55

Appendix D

Comparison RPF vs Data driven filter

Figures D.1, D.2, D.3, D.4 and D.5 each show a different scene where RPF is directly compared to the our filter. The figures are ordered in the following way:

- Row 1: Ground truth without/with depth of field respectively.
- Row 2: RPF. Unfiltered input, Filtered output, Filtered output MSE without (first three) and with (last three) depth of field.
- Row 3: Our filter at 8 spp. Unfiltered input, Filtered output, Filtered output MSE without (first three) and with (last three) depth of field.
- Row 4: Our filter at 16 spp. Unfiltered input, Filtered output, Filtered output MSE without (first three) and with (last three) depth of field.
- Row 5: Our filter at 32 spp. Unfiltered input, Filtered output, Filtered output MSE without (first three) and with (last three) depth of field.

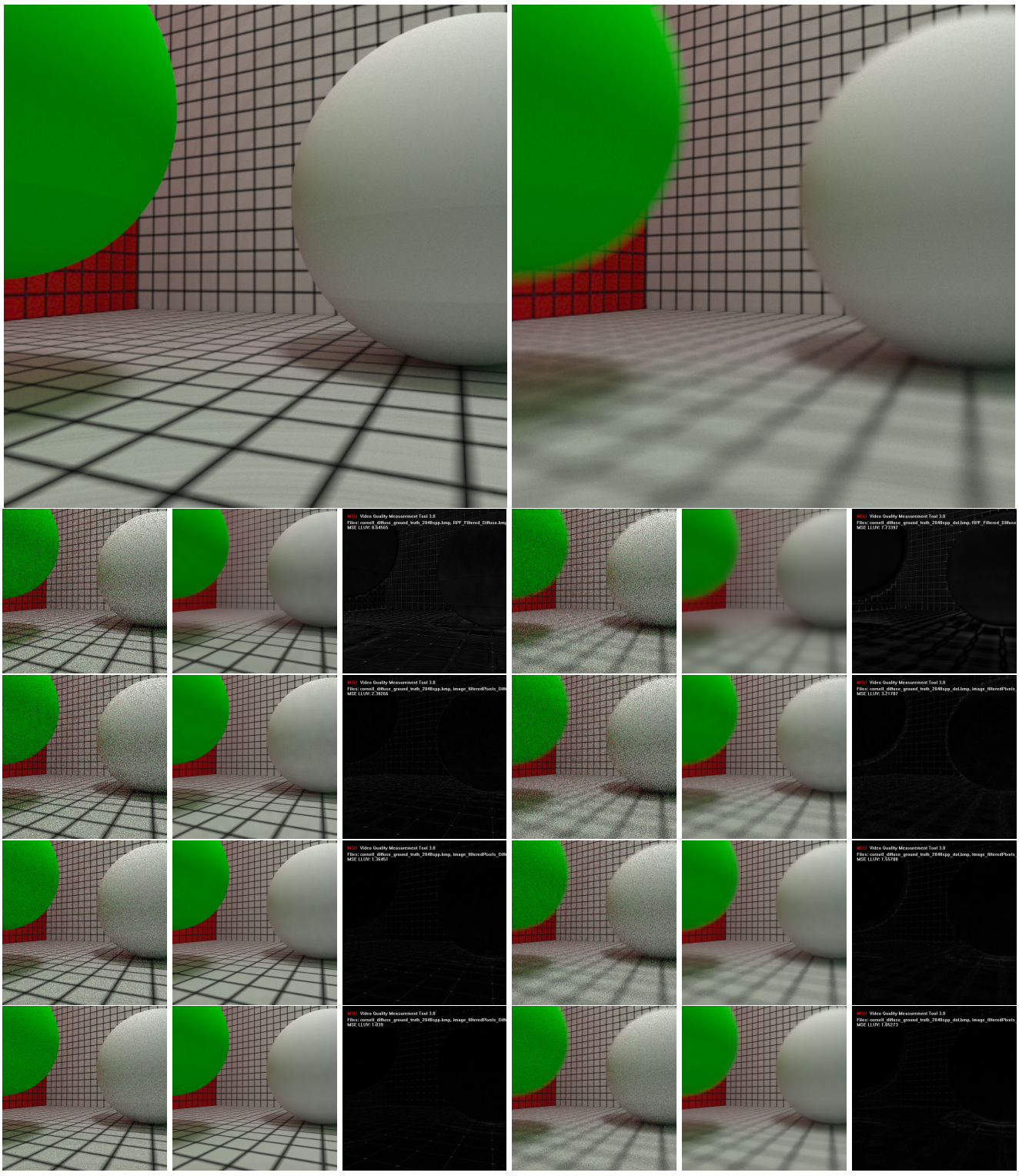


Figure D.1: Comparisons between RPF and data driven filter in the Cornell Diffuse scene.

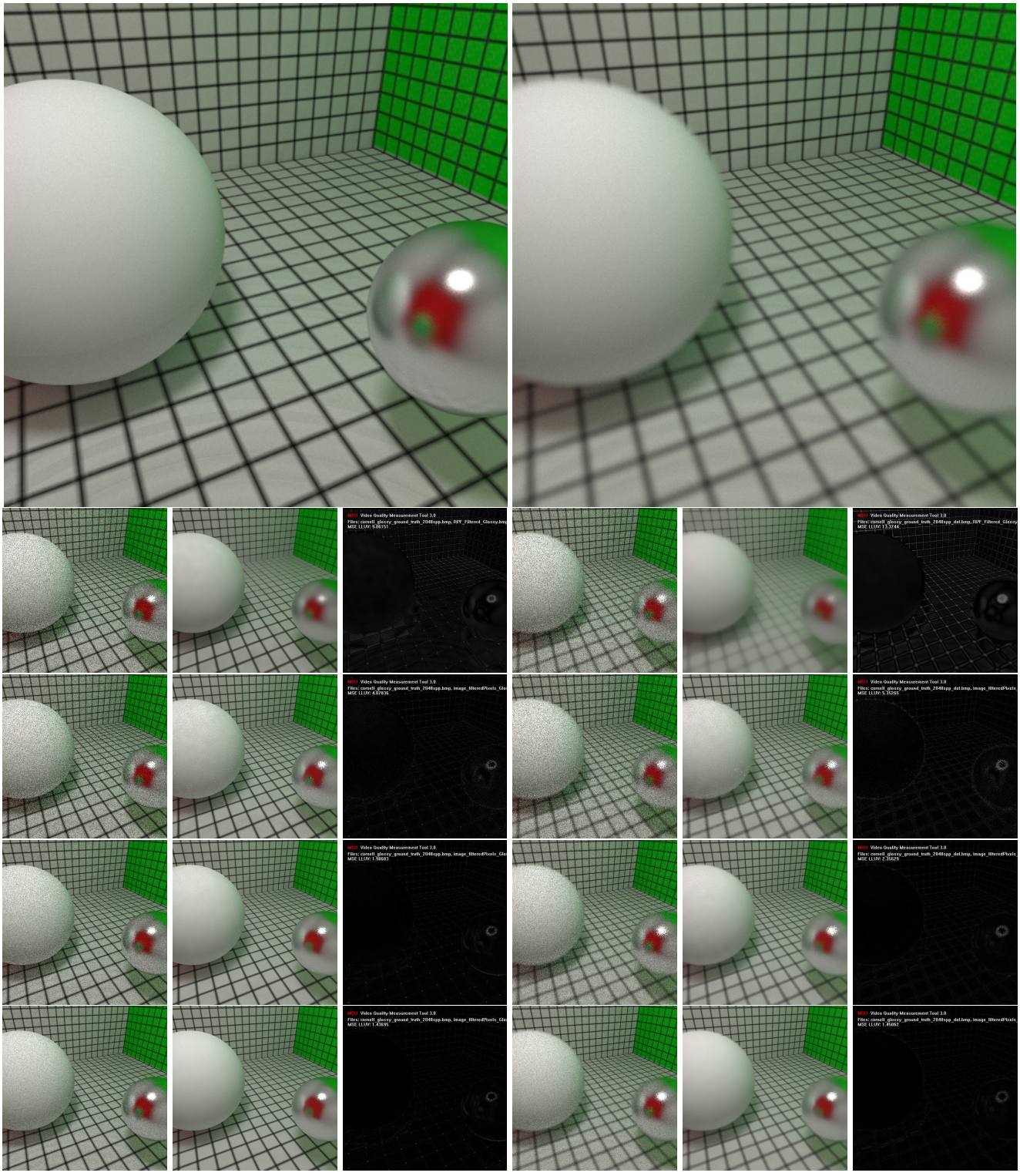


Figure D.2: Comparisons between RPF and data driven filter in the Cornell Glossy scene.

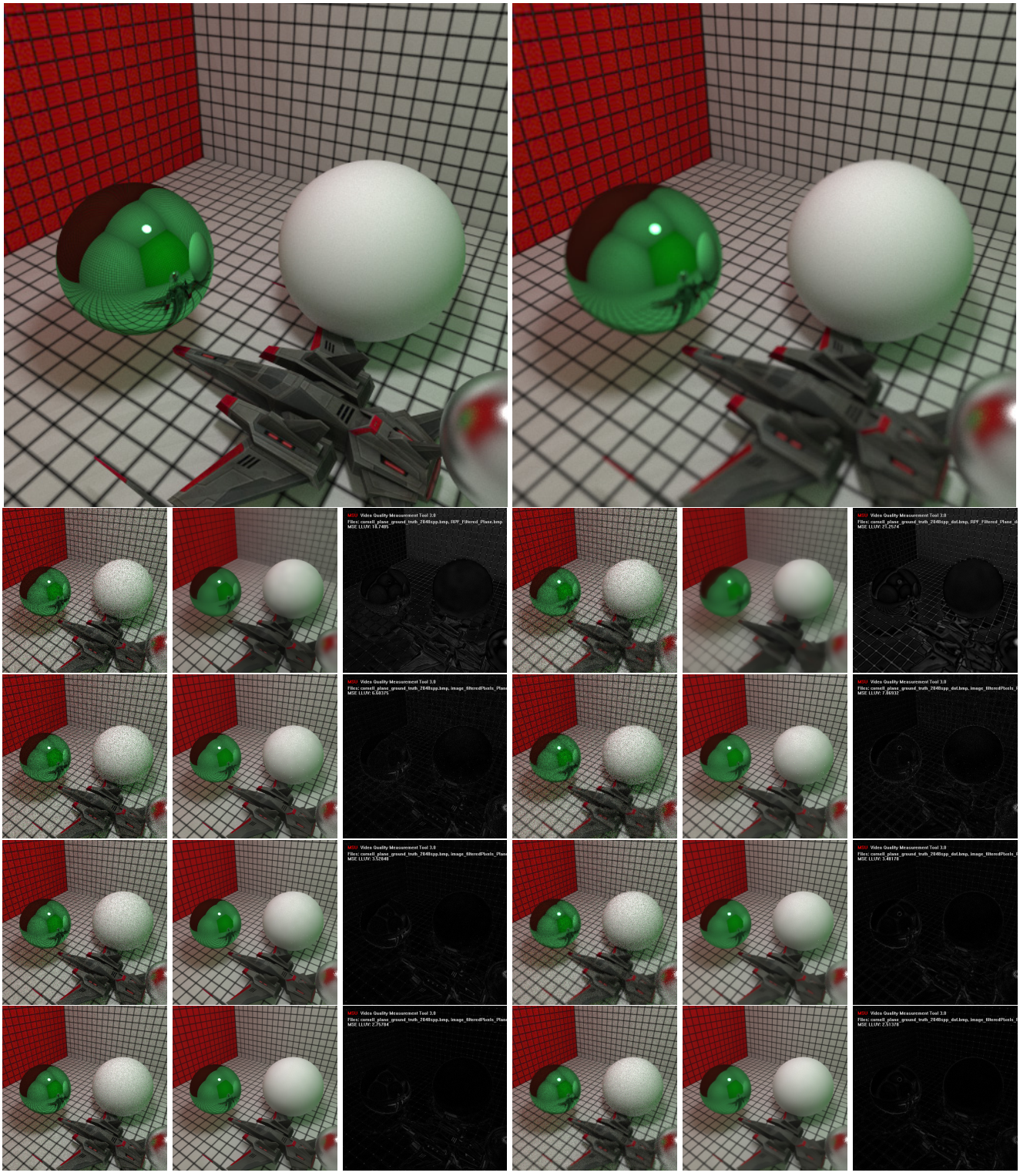


Figure D.3: Comparisons between RPF and data driven filter in the Cornell Plane scene.

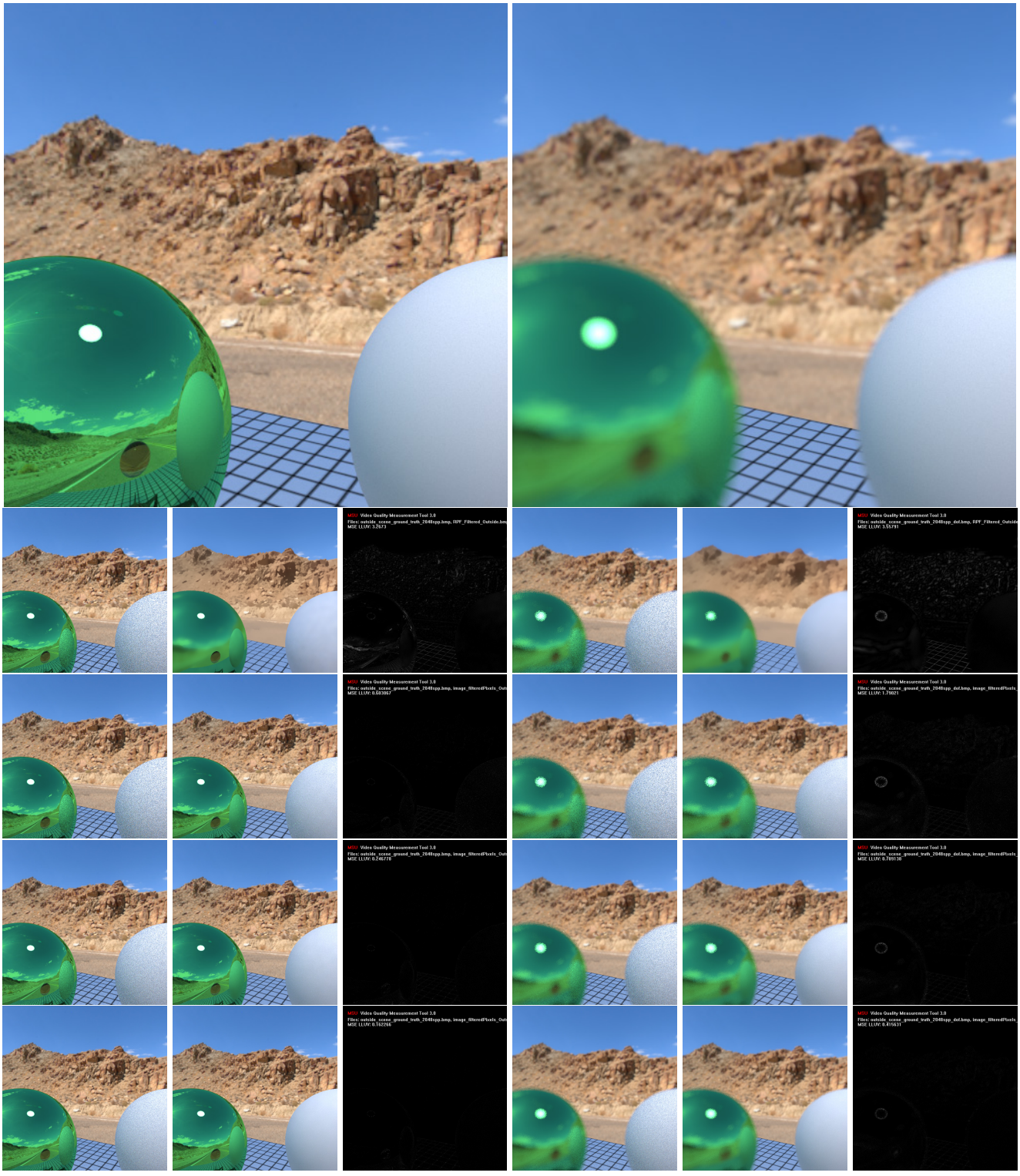


Figure D.4: Comparisons between RPF and data driven filter in the Outside scene.

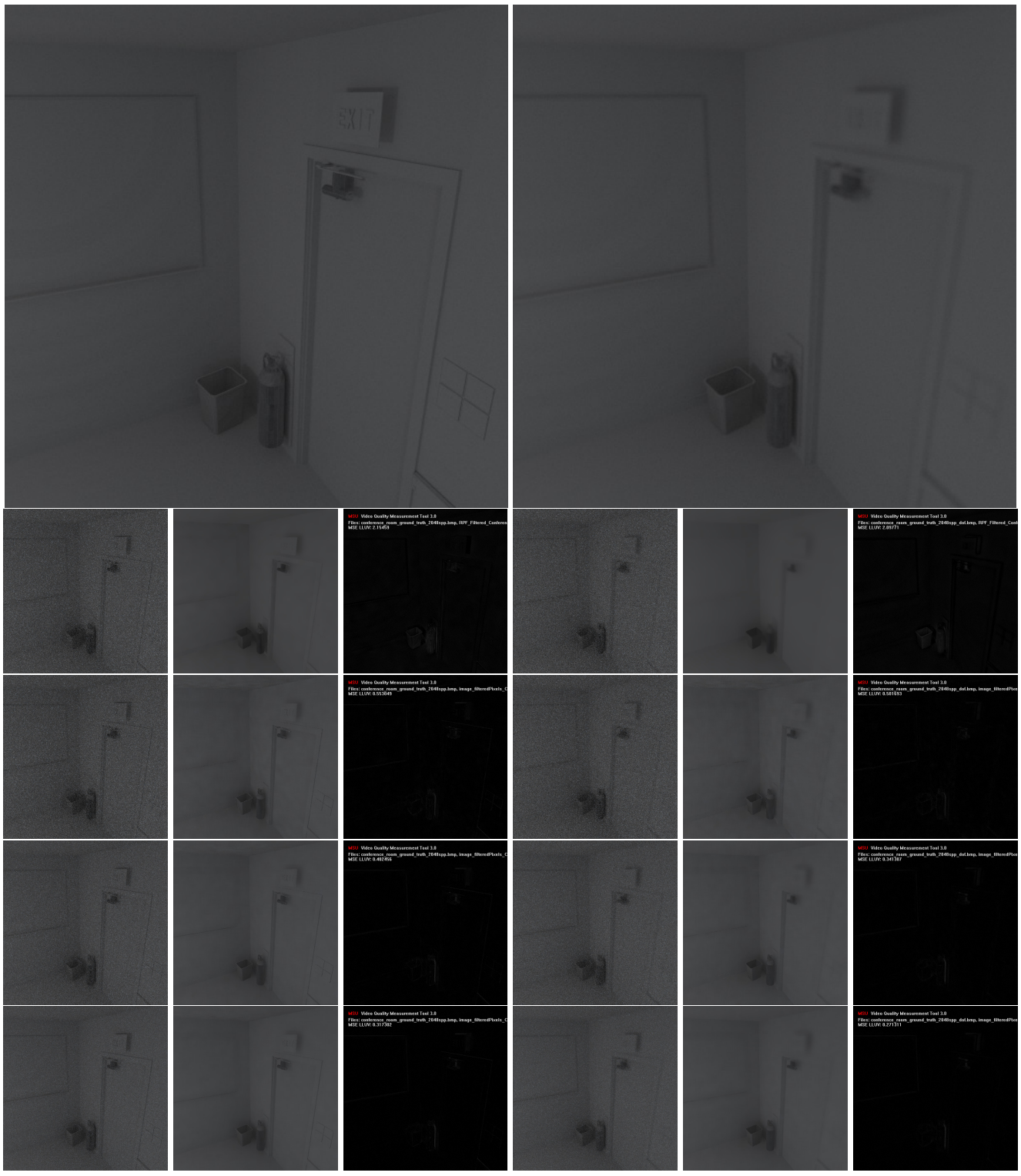


Figure D.5: Comparisons between RPF and data driven filter in the Conference room scene.