UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JONAS DEYSON BRITO DOS SANTOS

A Framework for Developing and Benchmarking Sampling and Denoising Algorithms

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Computer Science

Advisor: Prof. Dr. Manuel Menezes de Oliveira Neto

Porto Alegre
November 2019

CIP — CATALOGING-IN-PUBLICATION

Brito dos Santos, Jonas Deyson

A Framework for Developing and Benchmarking Sampling and Denoising Algorithms / Jonas Deyson Brito dos Santos. – Porto Alegre: PPGC da UFRGS, 2019.

96 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2019. Advisor: Manuel Menezes de Oliveira Neto.

1. Monte carlo rendering. 2. Adaptive sampling and reconstruction. 3. Denoising. 4. Benchmark. I. Oliveira Neto, Manuel Menezes de. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. Rui Vicente Oppermann
Vice-Reitora: Prof[a]. Jane Fraga Tutikian
Pró-Reitor de Pós-Graduação: Prof. Celso Giannetti Loureiro Chaves
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenadora do PPGC: Prof[a]. Luciana Salete Buriol
Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

ABSTRACT

In the context of Monte Carlo rendering, although many sampling and denoising techniques have been proposed in the last few years, the case for which one should be used for a specific scene is still to be made. Moreover, developing a new technique has required selecting a particular rendering system, which makes the technique tightly coupled to the chosen renderer and limits the amount of scenes it can be tested on. In this work, we propose a renderer-agnostic framework for developing and benchmarking sampling and denoising techniques for Monte Carlo rendering. It decouples techniques from rendering systems by hiding the renderer details behind a general API. This improves productivity and allows for direct comparisons among techniques using scenes from different rendering systems. The proposed framework contains two main parts: a software development kit that helps users to develop and and test their techniques locally, and an online system that allows users to submit their techniques and have them automatically benchmarked on our servers. We demonstrate its effectiveness by using our API to instrument four rendering systems and a variety of Monte Carlo denoising techniques — including recent learning-based ones — and performing a benchmark across different rendering systems.

Keywords: Monte carlo rendering. adaptive sampling and reconstruction. denoising. benchmark.

Um Ambiente para Desevonvoimento de Algoritmos de Amostragem e Remoção de Ruído

RESUMO

No contexto de *Monte Carlo rendering*, apesar de diversas técnicas de amostragem e remoção de ruído tenham sido propostas nos últimos anos, aportar qual técnica deve ser usada para uma cena específica ainda é uma tarefa difícil. Além disso, desenvolver uma nova técnica requer escolher um renderizador em particular, o que torna a técnica dependente do renderizador escolhido e limita a quantidade de cenas disponíveis para testar a técnica. Neste trabalho, um *framework* para desenvolvimento e avaliação de técnicas de amostragem e remoção de ruído para Monte Carlo rendering é proposto. Ele permite desacoplar as técnicas dos renderizadores por meio de uma API genérica, promovendo a reprodutibilidade e permitindo comparações entre técnicas utilizando-se cenas de diferentes renderizadores. O sistema proposto contém duas partes principais: um kit de desenvolvimento de software que ajuda os usuários a desenvolver e testar suas técnicas localmente, e um sistema online que permite que usuários submetam técnicas para que as mesmas sejam automaticamente avaliadas no nosso servidor. Para demonstramos a efetividade do ambiante proposto, modificamos quatro renderizadores e várias técnicas de remoção de ruído — incluindo técnicas recentes baseadas em aprendizado de máquina — e efetuamos uma avaliação utilizando cenas de diferentes renderizadores.

Palavras-chave: Renderização de Monte Carlo, Amostragem e Recostrução.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Aplication Programming Interface
ASR	Adaptive Sampling and Reconstruction
BRDF	Bidirectional Reflectance Distribution Function
CNN	Convolutional Neural Network
FBKSD	A Framework for Developing and Benchmarking Sampling and Denoising Algorithms for Monte Carlo Rendering
GAN	Generative Adversarial Network
GUI	Graphical User Interface
HVS	Human Visual System
MC	Monte Carlo
PBR	Pysically Based Rendering
SDK	Software Development Kit

LIST OF FIGURES

CONTENTS

1 INTRODUCTION

Rendering is one of the most important problems in computer graphics and has been the subject of over half a century of research. In particular, there has been a tremendous amount of exploration on Monte Carlo (MC) physically-based rendering (PBR) systems (COOK; PORTER; CARPENTER, 1984) such as path-tracing (KAJIYA, 1986) and its various extensions (VEACH; GUIBAS, 1997). To address shortcomings in Monte Carlo rendering, more than three decades of research has explored a wide variety of different ideas, including adaptive sampling and reconstruction algorithms (HACHISUKA et al., 2008), faster acceleration structures and intersection algorithms (SAMET, 2010), improved sampling patterns (HECK; SCHLöMER; DEUSSEN, 2013), and Monte Carlo denoisers (SEN; DARABI, 2012; ROUSSELLE; MANZI; ZWICKER, 2013; KALANTARI; BAKO; SEN, 2015) to name a few broad categories.

Although developing a new algorithm that successfully improves Monte Carlo rendering in some way is a challenging task in itself, researchers face further challenges, namely:

1. *They must implement their algorithm into an actual rendering system* so they can test it on complex scenes. After all, renderers have several key components required to produce high-quality images (*e.g.*, scene I/O, samplers, ray-traversal acceleration data structures, primitive-ray intersectors, shading systems, and reconstruction filters), and many of these components are often orthogonal to the algorithm being explored. Therefore, researchers often leverage the infrastructure provided by existing rendering systems. However, integrating the algorithm into a rendering system is often an error-prone and time-consuming task, since available systems usually do not provide the necessary features and need to be "hacked" (deeply modified) to support the new technique;
2. *They must find several high-quality scenes to test their algorithm and demonstrate its performance*. Since most researchers are not digital artists, constructing complex aesthetically pleasing scenes is often a non-trivial, time-consuming task, and "programmer-art" scenes do not tend to be of the same quality as those constructed by professional artists. Thus, researchers tend to stick to a handful of publicly-available test scenes that have been used in previous papers or are included in textbooks on physically-based rendering (PHARR; JAKOB; HUMPHREYS, 2016; PHARR; HUMPHREYS, 2010a);

3. To compare their new algorithms with previous ones, they often need to port the previous algorithms to the same rendering system they adopted, which can also be tricky, since the developers performing the port are usually not very familiar with the other algorithm's source codes. This process can introduce bugs and may not produce ideal results, since the algorithmic parameters that worked successfully for one rendering system might not work for the new one. Trying to determine the optimal parameters for an algorithm that one did not develop can be a very time consuming task.

The consequence of these challenges is that, in most occasions, researchers end up demonstrating their algorithms on a single rendering system using a small number of test scenes, and comparing them against a handful of competing methods converted in a hurry from other rendering systems. These competing methods are usually chosen for being considered the best ones, but this conclusion may have been made based on results from different test scenes. This significantly limits one's ability to thoroughly test and explore the proposed method, as well as the reviewers' ability to properly evaluate its performance. These problems can have a negative impact on the reproducibility of results on this field.

Having good "apples-to-apples" comparisons is important when trying to gauge the benefits of new and existing algorithms, and from the problems stated above, we consider that, in its current state, the rendering research community lacks the adoption of a standard benchmark. Such benchmark would benefit both the research community — by making it easier to develop a new technique, and to have a better picture of the state-of-the-art — and the end users — by making it easier to choose the best technique for a certain scene.

1.1 Contributions

In this thesis, *we present a novel framework that allows researchers to develop, test, and compare algorithms for Monte Carlo rendering*. Specifically, we propose an application program interface (API) and a software development kit (SDK) that allow developers to easily run their algorithms on different rendering systems by providing the necessary communication between such algorithms and the other components of an existing rendering system. In other words, instead of having the researchers port their algorithms to multiple rendering systems, we have done the leg work for them by instrumenting rendering systems to provide the necessary services through our API. Also, we are releasing an

online service that allows researches to submit their techniques, have them automatically benchmarked in our server, and the results published online.

Therefore, a researcher only needs to implement an algorithm once, and can immediately use it with all rendering systems that support our framework. This allows researchers to rapidly test and deploy their algorithms on a range of rendering systems, and test them on a wide variety of scenes. This allows for automatic independent benchmarking between algorithms, which is quite useful when submitting new techniques for publication.

We have initially instrumented three well known rendering systems used by the research comunity (PBRT v3, PBRT v2 and Mitsuba), and a fourth custom procedural renderer developed from scratch. Since our framework is publicly available on GitHub (https://github.com/fbksd/fbksd) under the MIT license, anyone can extend our API providing support to other rendering systems.

To demonstrate the effectiveness of our framework, we conduct a case study involving Monte Carlo (MC) denoising algorithms. Such a study illustrates key aspects of our system: provide easy integration of algorithms and rendering systems (by means of just a few calls to the API); provide an independent benchmark for MC techniques that works across various rendering systems; and, allow developers to evaluate the performance of rendering systems with various algorithms, and vice versa. These are desirable features for algorithm and rendering system developers, as well as for the academic, industry, and end-user communities, who should be able to make better informed decisions when choosing a technique for a given scene.

For our study, we have instrumented (adapted the original code from the authors to our API, or written the method from scratch based on the paper) a significant number of state-of-the-art MC denoising algorithms: NFOR (BITTERLI et al., 2016), LBF (KALANTARI; BAKO; SEN, 2015), RHF (DELBRACIO et al., 2014), LWR (MOON; CARR; YOON, 2014), RDFC (ROUSSELLE; MANZI; ZWICKER, 2013), RPF (SEN; DARABI, 2012), SBF (LI; WU; CHUANG, 2012), NLM (ROUSSELLE; KNAUS; ZWICKER, 2012), GEM (ROUSSELLE; KNAUS; ZWICKER, 2011), and KPCN (BAKO et al., 2017). This allows them to be used with the four rendering systems, even though they have been originally developed for a single renderer.

The techniques we chose form a representative sample of the techniques found in the field, including adaptive, non-adaptive, *a priori*, and *a posteriori* methods (ZWICKER et al., 2015). Furthermore, our system's ability to automatically generate benchmark reports allows for the comparison of the different methods on an even playing field. In our study,

we compare the performance of different MC denoising methods and discuss some of their identified potential limitations.

Although this work does not propose a new MC rendering algorithm *per se*, this kind of *meta-research* system (*i.e.*, a system designed to aid the research process) is not new to the graphics and vision communities. Successful examples include the Middlebury dataset (SCHARSTEIN; SZELISKI, 2002; SCHARSTEIN; SZELISKI; HIRSCHMüLLER, 2002), which has transformed the way two-frame dense stereo correspondence algorithms are developed and compared, as well as the benchmarks on Alpha Matting (RHEMANN et al., 2009b; RHEMANN et al., 2009a), optical flow (BARRON; FLEET; BEAUCHEMIN, 1994; BAKER et al., 2011b; BAKER et al., 2011a), and video matting (EROFEEV et al., 2015; EROFEEV et al., 2014). More recently, Anderson et al. (ANDERSON et al., 2017) proposed a framework to compile PDF sampling patterns for Monte Carlo.

Inspired by these works, our system provides test scenes intended to stress the limits of Monte Carlo techniques and reveal their potential limitations. It is extensible, allowing for easy support of new rendering systems, as well as sampling and reconstruction strategies. The community should be able to contribute new scenes and techniques in a simple way.

The summary of the **contributions** of this thesis include:

- A framework for developing and benchmarking sampling and denoising algorithms for MC rendering. Our framework decouples the algorithms from rendering systems by means of an API, allowing researchers to implement their techniques once and run them on any rendering system supporting our framework. Our framework can be dynamically extended to support new algorithms, rendering systems, and testing datasets;
- An automatic and independent benchmarking system for comparing Monte Carlo algorithms across multiple rendering systems and supporting a large number of scenes. This should be a useful tool for assessing the quality of new Monte Carlo algorithms against established ones, especially for submission purposes;
- An online submission system for sampling and denoising techniques, which allows users to have their techniques automatically benchmarked in our server and the results published online;
- A detailed evaluation of the state-of-the-art Monte Carlo denoising and sampling algorithms using our framework and a discussion of their performance and limitations.

1.2 Thesis Structure

We start by giving a brief overview of *physically based rendering* in Chapter 2. We cover the following topics: Radiometry, Rendering Equation, Monte Carlo Integration, Monte Carlo Ray Tracing, and Sampling and Reconstruction. For a more in-depth discussion on these and other topics on PBR, refer to Dutre et al. (2006) and Pharr and Humphreys (2010b).

In Chapter 3, we discuss the related work: *Meta-research in graphics*, *benchmark systems*, *Monte Carlo denoising*, and *sampling* algorithms.

In Chapter 4, we present the our proposed system — called FBKSD. Chapter 5 presents an *online system* powered by FBKSD, which allows users to submit sampling and denoising techniques and have them automatically benchmarked and published in our server. A case study showing results for several state-of-the-art denoising techniques is presented in Chapter 6, followed by our conclusions and a discussion on possible future work directions in Chapter 7.

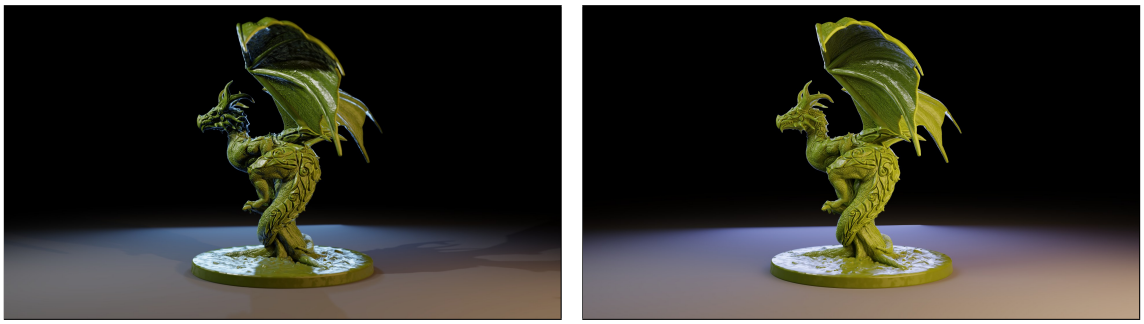
2 BACKGROUND ON PHYSICALLY BASED RENDERING

A *physically based rendering* (PBR) system seeks to produce — based on the description of a three-dimensional scene — *photorealistic* images. As defined by (FERWERDA, 2003), a photorealistic image is one that produces the same *visual response* as the real scene. In other words, a photorealistic image will look indistinguishable from a picture of a real-world scene to an average observer. Another term sometimes used to describe images produces by PBR systems is the term *physically correct*. A *physically correct* or *physically realistic* image is one that provides the same *visual stimulation* as the real scene. This can be seen as a stronger condition than *photorealism*. In other words, an image can be *photorealistic* without being *physically correct*. This makes physically correct rendering techniques the *gold standard*, when it comes to computer graphics image generation.

For some applications (e.g., games, interactive visualization, etc.), performance is more important then physically correctness, therefore, they usually employ more aggressive simplifications that sacrifice physically correctness, but still deliver a satisfactory level of realism for their purposes. Techniques that are not guaranteed to produce physically correct results due to those compromises are called *biased*. Figure 2.1 shows a comparison of the same scene being rendered with a biased real-time renderer and an unbiased physically-based one.

Figure 2.1: Scene *The Green Dragon* (from Zacharias Reinhardt). (a) Rendered with a real-time renderer (Blender's Eevee engine). (b) Rendered with an unbiased physically-based renderer (Blender's Cycles engine).

(a) (b)



Most unbiased PBR systems utilize a model of light based on *geometrical optics*. Although this model is not detailed enough to describe phenomena that occurs when light interacts with very small objects (sizes close to the light wavelength), such as dispersion and interference, it is good enough for solving most rendering problems in computer graphics

without being intractable.

2.1 Radiometry

Since the main task of a PBR system is simulating light propagation and interaction with objects, we will start by giving an overview of the fundamental units and light measurement concepts, a subject called *Radiometry* — the science of measuring light.

2.1.1 Flux

The energy Q of a single photon with wavelength λ is given by

$$Q = \frac{hc}{\lambda}, \tag{2.1}$$

where h is the Planck's constant ($h \approx 6.626 \times 10^{-34}\,\mathrm{J \cdot s}$), and c is the speed of light in vacuum ($c = 299{,}472{,}458\,\mathrm{m/s}$). The *flux* (also known as radiant power) is the total amount of energy traversing a surface or region per unit time:

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t}. \tag{2.2}$$

The unit of flux is the *watt* (W). Flux is commonly used to describe total power of light sources.

2.1.2 Irradiance

Consider an sphere of radius r with a light source of power Φ in its center. If we increase the radius of the sphere, its surface will appear dimmer, even though the same total flux is hitting the surface. To measure how much energy a point receives (or emits), we also need to account for area. Irradiance (E) is the amount of flux per unit area arriving at a surface. It is given by

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A}. \tag{2.3}$$

Usually, when the flux is leaving the surface, the term *radiant exitance* (M) (also known as *radiosity* (B)), is used.

In the example of the sphere with a light in the center we gave previously, the irradiance at any point on the surface is

$$E = \frac{\Phi}{4\pi r^2}\,.$$

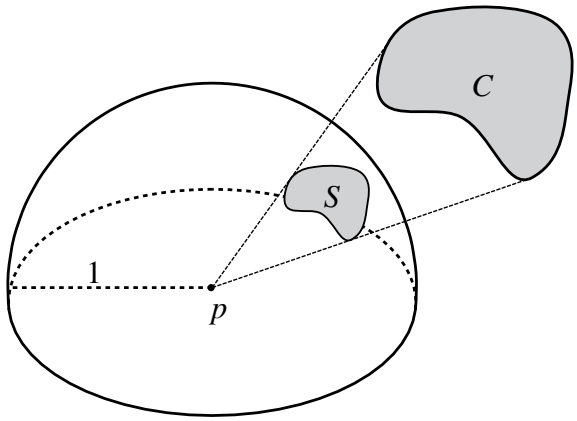
where Φ is the power of the light source, and r is the radius of the sphere. Note that the energy arriving at a point falls off with the squared distance.

2.1.3 Radiance

Although irradiance tells us how much light a point receives, it does not account for directionality: the energy arriving at a point comes from all directions. We want to be able to tell how much energy a point receives from a particular direction, but first we need to define *solid angle*.

The solid angle S of an object C viewed from a point p is the projected area of C onto the unit sphere centered at p (Figure 2.2). The unit of solid angle is the *steradian* (sr), and its value can range from zero to 4π.

Figure 2.2: Solid angle.



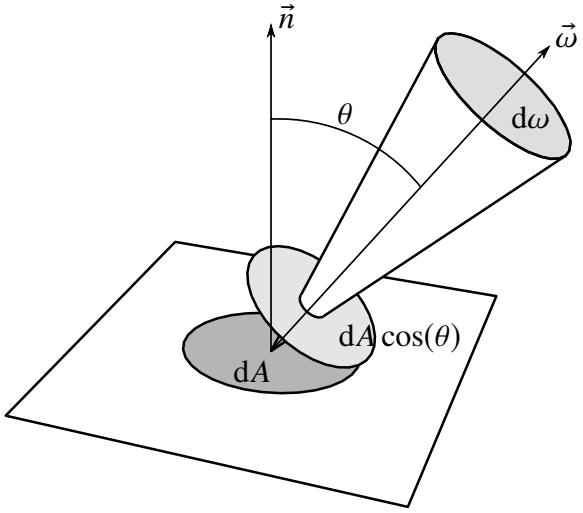
Radiance (L) is the quantity that considers both area and solid angle. It is defined as the flux per unit area perpendicular to the beam, per unit solid angle:

$$L = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\omega\,\mathrm{d}A\cos(\theta)}\,, \tag{2.4}$$

where θ is the angle between the surface normal and the direction ω (Figure 2.3).

Radiance is the most important radiometric quantity in PBR: all other quantities

Figure 2.3: Radiance measurement.



can be obtained from it by integrating area or solid angle, also, radiance remain constant along a ray in empty space.

2.2 Rendering Equation

The rendering equation (KAJIYA, 1986) gives the radiance L_o leaving a point p on a surface in a given direction ω_o, in terms of the incident radiance from all directions and the material properties on that point:

$$L_o(p, \omega_o) = L_e(p, \omega_o) + \int_\Omega f_r(p, \omega_o, \omega_i) L_i(p, \omega_i) |\cos(\theta_i)| \, d\omega_i \, , \tag{2.5}$$

where $L_e(p, \omega_o)$ is the radiance emitted by the point p in the direction ω_o (if the point p belongs to a light source), f_r is the BRDF (bidirectional reflectance distribution function), which represents the characteristics of the material at p, θ_i is the angle between the surface normal at p and ω_i, and $L_i(p, \omega_i)$ is the incident radiance on the point p coming from the direction ω_i.

The rendering equation is a Fredholm integral equation of the second type (ARVO, 1995). It has no analytical solution for the general case and the high dimensionality of the domain makes traditional numerical integration methods very inefficient (curse of dimensionality). Therefore, the *Monte Carlo* integration method has been the most successful at solving the general light transport problem. However, due to its stochastic nature, this method produces noise in the resulting image due to variance in the estimates, and requires a large number of samples to achieve high-quality results.

2.3 Monte Carlo Integration

Monte Carlo (MC) is a numerical integration method that uses random points generated in the integrand domain to estimate the result. It is conceptually a very simple method and is capable of elegantly handle high-dimensional domains, like the rendering equation: it only requires the ability to evaluate the function being integrated at arbitrary points in its domain.

The Monte Carlo estimator for the integral of a function g using n random points is given by

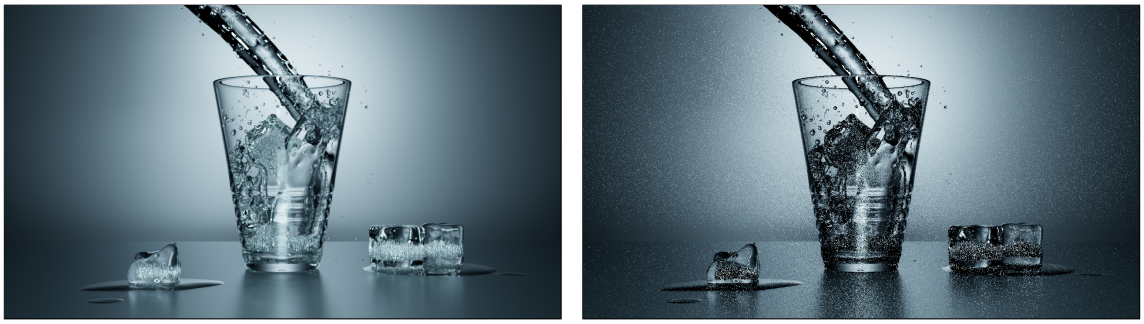
$$\int_D g(x)\,\mathrm{d}x \approx \frac{1}{n}\sum_{i=1}^{n}\frac{g(x_i)}{p(x_i)}, \tag{2.6}$$

where p is the *probability distribution function* (PDF) used to generate the random points $x_i \in D$.

The approximate solution given by Equation 2.6 is correct on average, but has a certain variance which, in rendering, manifests itself as noise in the generated images (Figure 2.4). This variance can be reduced by increasing the number of samples, but this strategy is not satisfactory since it is known that the variance of a Monte Carlo estimate drops at a rate of $O(\sqrt{n})$ of the number of samples (VEACH, 1998), Thus, it is necessary to quadruple the number of samples to halve variance.

Figure 2.4: Typical noise in an image rendered with a Monte Carlo technique. (a) Reference image. (b) Noisy image rendered with 16 samples per pixel.

(a) (b)



2.4 Monte Carlo Ray Tracing

Currently, most PBR systems utilize *ray tracing* to find light paths between the camera and the light sources. A light path is a sequence of vertices connected by straight lines, where the first vertex belongs to the camera sensor, the last vertex belongs to a light source, and the inner vertices represent interactions with the other objects in the scene.

The term *ray tracing* is very broad, since it can be used to describe techniques with very different capabilities, for example: *ray casting* (APPEL, 1968), *classic ray tracing* (WHITTED, 1980), *distributed ray tracing* (COOK; PORTER; CARPENTER, 1984), *path tracing* (KAJIYA, 1986), etc. A more specific term used to describe those techniques that use ray tracing in conjunction with Monte Carlo — which usually are the ones used in PBR systems — is *Monte Carlo ray tracing*.

To better describe the capabilities of different techniques, we can use a notation introduced by Heckbert (1990). This notation describes all possible light paths that a technique can compute using a regular expression[1] like $E(S|D)^*L$, where E is a vertex on the camera sensor, S and D are vertices on surfaces where specular and diffuse interactions occurred, respectively, and L is a vertex on a light source.

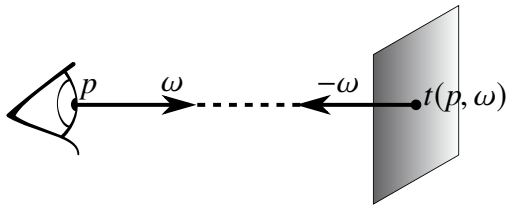
Using this notation, the *ray casting* (APPEL, 1968) method is capable of calculating paths of the type $E(D|\epsilon)L$, which is called local or direct lighting. The *classic ray tracing* (WHITTED, 1980) method includes perfect specular reflections and refractions by recursively tracing rays from specular objects, resulting in paths of the type $E(D|\epsilon)S^*L$. *Distributed ray tracing* (COOK; PORTER; CARPENTER, 1984) does not allow new light path types, but it includes *distributed* effects like motion blur and depth-of-field. Finally, *path tracing* (KAJIYA, 1986) includes paths of the type $E(D|S)^*L$, which is the most general type according to this classification. Later improvements, like *bidirectional path tracing* (LAFORTUNE; WILLEMS, 1993) and *Metropolis light transport* (VEACH; GUIBAS, 1997), are optimizations that allow faster convergence in harder lighting conditions, but the capabilities in terms of light path types are the same.

[1] Regular expression notation: Y^* denotes zero or more occurrences of Y, Y^+ denotes one or more occurrences of Y, $Y|Z$ matches either Y or Z, ϵ is the empty string, and parenthesis are used for grouping.

2.4.1 *Path Tracing*

The *path tracing* method finds light paths by tracing rays that start from the camera, bounce off objects in the scene, and reach the light sources. To formalize this idea, consider a ray tracing function $t(p, \omega)$ that returns the first intersection point found by tracing a ray from the point p in the direction ω (Figure 2.5). The camera measures the radiance L_i arriving at the point p on the sensor, and since radiance does not change along a ray, we can write $L_i(p, \omega) = L_o(t(p, \omega), -\omega)$. So, to compute the radiance arriving at a point in the sensor, we trace a ray from the camera with a direction ω, find the first intersection point with the objects in the scene, and compute the radiance L_o leaving that point in the direction $-\omega$.

Figure 2.5: Ray tracing function.



From the rendering equation (Equation 2.5), if we drop the subscripts from L_o and L_i, we can see that L appears on both sides of the equation:

$$L = L_e + TL\ , \tag{2.7}$$

where we use T as the integral operator, for brevity. If we expand L on the right side, we get the following Neumann series:

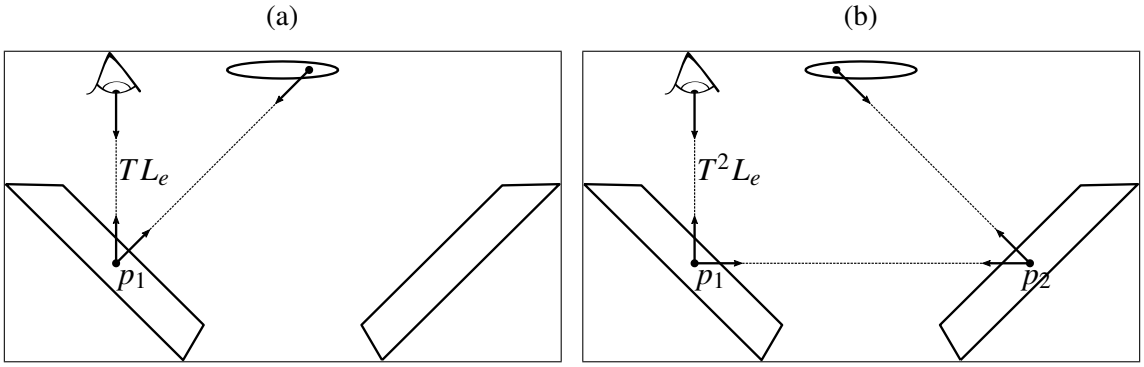
$$L = L_e + TL_e + T^2L_e + \ldots . \tag{2.8}$$

The L_e term gives the light emitted by the object (if the object is a light source itself), TL_e gives the light that bounces off once, which is usually called *direct illumination*, the T^2L_e term gives the light that bounces off twice, and so on. The total contribution from the terms T^iL_e, with $i > 1$, sums up to what is usually called *indirect illumination*.

The path tracing method estimates the term $T^{k+1}L_e$ using Monte Carlo and by reusing the previous path segment used for the term T^kL_e. Figure 2.6 illustrates the process. At the first intersection point p_1, the direct illumination (TL_e) is calculated by sampling a

light source. Then, from this point, another ray is traced by sampling a random direction, obtaining a second intersection point p_2. The direct illumination on this second point is also calculated, and the fraction of this radiance that bounces off the first point gives the estimate for $T^2 L_e$. The end of this process is determined by a probabilistic technique called *Russian roulette* (ARVO; KIRK, 1990), which allows the algorithm to remain unbiased even though paths of finite length are computed.

Figure 2.6: Path tracing method. (a) Estimate for the first term TL_e. (b) Estimate for the second term $T^2 L_e$. The circle represents a light source, and the squares represent other non-emitting surfaces in the scene.



Since the path tracing algorithm generates paths only starting from the camera, convergence can be very slow in difficult lighting situations — when only a small fraction of all the paths end up hitting a light source. A technique that performs better in these scenarios is to also trace paths starting from the light sources, and combine them with the ones generated from the camera. This is the basic idea of the *Bidirectional Path-tracing* (LAFORTUNE; WILLEMS, 1993) method.

2.5 Sampling and Reconstruction

The final step of a PBR system is to generate a digital image — a 2D matrix (grid) of pixels — from the radiance values computed by solving the rendering equation. To better understand this process, we first explain the distinction between the *scene function*, defined on the continuous camera raster space, and the digital image (grid of pixels) generated from it.

Given the camera sensor with dimensions $width \times height$ in raster space, the scene

function f returns a radiance value L for each point (x, y) on the sensor:

$$f(x, y) \rightarrow L\,, \tag{2.9}$$

where $(x, y) \in [0, height] \times [0, width]$. This function encapsulates the details of the particular method used to compute the radiance values. But since in PBR systems Monte Carlo is usually used at some point, the scene function also needs several other parameters as input, e.g., a point (u, v) in the camera lens, a time parameter t, points on the surfaces of the light sources, etc. Therefore, it is also common to write the scene function as:

$$f(x, y, t, u, v, r_1, r_2, \ldots) \rightarrow L\,. \tag{2.10}$$

Also note that, besides radiance values, PBR systems can also output other scene features, e.g., geometric features like normal and position for the ray/object intersection points found during ray tracing, texture values, etc. We will call the scene function input $(x, y, t, u, v, r_1, r_2, \ldots)$ *sample position*, and the output (radiance and any other features) *sample value*.

During rendering, a PBR system generates a set of random sample positions (Section 3.4), which are used to evaluate the scene function, the resulting sample values are then used to reconstruct the pixel values of the final image. This is done by interpolating sample values near the center of each pixel:

$$I_{ij} = \frac{\sum_k w(p_i - x_k, p_j - y_k) f(x_k, y_k)}{\sum_k w(p_i - x_k, p_k - y_k)}\,, \tag{2.11}$$

where I_{ij} is the value of the pixel (i, j), (x_k, y_k) are the sample positions on the sensor, (p_i, p_j) is the center of pixel (i, j) in raster space, w is the reconstruction filter, and f is the scene function. The filter w determines how much a certain sample contributes to the final pixel value based on the distance between the sample position and the center of the pixel.

Although the reconstruction filter w in Equation 2.11 uses only (x, y) coordinates to compute the distance between the pixel center and the sample position — which is true for simple filters — in practice, many sophisticated denoising techniques (Section 3.3) may leverage other features provided by the system.

3 RELATED WORK

We begin by discussing meta-research systems in both graphics and vision which, like our own framework, have been developed to facilitate/improve the research process. Afterwards, we focus on previous work on Monte Carlo denoising, and sampling, which is the application that we use to illustrate the benefits of our framework.

3.1 Meta-Research in Graphics

Several systems have been proposed over the years to facilitate research development in graphics. Some of the most popular ones include Cg (MARK et al., 2003), Brook (BUCK et al., 2004), and Halide (RAGAN-KELLEY et al., 2012). Cg is a general-purpose, hardware-oriented, programming language and supporting system designed for the development of efficient GPU applications, and providing easy integration with the two major 3D graphics APIs (OpenGL and Direct 3D). Brook (BUCK et al., 2004) is also a system for general-purpose computation that allows developers to use programmable GPUs as streaming co-processors, while abstracting GPU architectural details. Halide (RAGAN-KELLEY et al., 2012) tries to optimize image-processing algorithms by decoupling the algorithm's description from its schedule. This allows for an algorithm to be described once, while specific schedules are provided for different target platforms (*e.g.*, CPUs, GPUs, mobile devices, etc.). Automatic generation of optimized schedules in Halide has been addressed in a follow-up work (MULLAPUDI et al., 2016).

While the primary goal of these systems is to generate efficient code while abstracting hardware details from developers, our focus is on decoupling Monte Carlo algorithms from rendering systems. This greatly simplifies the task of porting algorithms to multiple rendering systems, freeing developers from the burden of knowing implementation details of specific renderers to be able to perform integration. Our system also makes a wider range of scenes available for testing, providing a comprehensive, multi-rendering system benchmark for Monte Carlo algorithms. Recently, Anderson et al. (ANDERSON et al., 2017) proposed an approach to compile sampling BRDFs for MC applications. Their method complements our work.

3.2 Benchmarking Systems in Computer Vision

Quantitative benchmarks have been proposed for several computer visions areas, including optical flow (BARRON; FLEET; BEAUCHEMIN, 1994; BAKER et al., 2011b), dense two-frame stereo correspondence (SCHARSTEIN; SZELISKI, 2002), and alpha matting (RHEMANN et al., 2009b). These initiatives have provided independent tools for assessing the quality of the results produced by existing and new algorithms, and have led to significant progress in these areas.

3.2.1 Optical Flow

Barron et al. (BARRON; FLEET; BEAUCHEMIN, 1994) compared accuracy, reliability, and density of velocity measurements for several established optical flow algorithms, and showed that their performance could vary significantly from one technique to another. Baker et al. (BAKER et al., 2011b) proposed another benchmark for optical-flow algorithms that considers aspects not covered by Barron et al. These include sequences containing non-rigid motion, realistic synthetic images, high frame-rate video to study interpolation errors, and modified stereo sequences of static scenes. The authors have made their datasets and evaluation results publicly available, and provide the option for one to submit his own results for evaluation (BAKER et al., 2011a)

3.2.2 Stereo Correspondence

The Middlebury benchmark (SCHARSTEIN; SZELISKI, 2002) provided a taxonomy and evaluation for dense two-frame stereo correspondence algorithms. The datasets and evaluation are publicly available on the web, and anyone can submit results for evaluation (SCHARSTEIN; SZELISKI; HIRSCHMüLLER, 2002).

3.2.3 Alpha Matting

Rhemann et al. (RHEMANN et al., 2009b) introduced a benchmark system for alpha matting techniques. The authors provide some training data and use a test dataset for which the corresponding ground truth has not been disclosed. Similarly to the optical-flow

and dense stereo correspondence benchmarks mentioned before, the results are available on-line, and anyone can submit results for evaluation (RHEMANN et al., 2009a).

3.2.4 Video Matting

Erofeev et al. (EROFEEV et al., 2015) extended the alpha matting benchmark to videos, supporting both objective and subjective evaluations of video matting techniques. Training and test datasets are provided, with results and submissions being available through the web (EROFEEV et al., 2014).

Unlike such systems, ours goes beyond rating submitted results computed off-line. It provides an API that allows Monte Carlo algorithms to be tested with different rendering systems using a variety of scenes. Thus, it can compare different techniques across multiple rendering systems, something that was not previously possible without requiring the developer to create multiple implementations tailored to individual rendering systems. We intend to make our system freely available to the research and industry communities soon.

3.3 Monte Carlo Denoising Algorithms

Although there has been a significant amount of work on reducing the variance of MC rendered images through sampling/reconstruction (see (PHARR; JAKOB; HUMPHREYS, 2016; ZWICKER et al., 2015)), for brevity we shall only focus on previous post-processing approaches that filter *general* Monte Carlo noise (*i.e.*, noise from any and all distributed effects, path tracing, and so on).

Soon after the seminal paper by Cook et al. (COOK; PORTER; CARPENTER, 1984) raised the problem of MC noise, there was some early work in general MC filtering, including approaches using nonlinear median and alpha-trimmed mean filters for edge-aware spike removal (LEE; REDNER, 1990) and variable-width filter kernels to preserve energy and salient details (RUSHMEIER; WARD, 1994). However, in the years that followed, researchers largely ignored general MC filtering algorithms in favor of other variance reduction techniques, due to the inability of these filters to successfully remove the MC noise while preserving scene detail.

Recently, interest in general MC filtering algorithms has enjoyed a significant revival. For example, Sen and Darabi (SEN; DARABI, 2012) demonstrated that filters

could effectively distinguish between noisy scene detail and MC noise. To do this, they used mutual information to determine dependencies between random parameters and scene features, and combined these dependencies to weight a cross-bilateral filter at each pixel in the image. Rousselle et al. (ROUSSELLE; KNAUS; ZWICKER, 2012) proposed to use a non-local means filter to remove general MC noise. Kalantari and Sen (KALANTARI; SEN, 2013) applied median absolute deviation to estimate the noise level at every pixel to use any image denoising technique for filtering the MC noise. Finally, Delbracio et al. (DELBRACIO et al., 2014) modified the non-local means filter to use the color histograms of patches, rather than the noisy color patches, in the distance function.

Other approaches have effectively used error estimation for filtering general distributed effects. For example, Rousselle et al (ROUSSELLE; KNAUS; ZWICKER, 2011) used error estimates to select different filter scales for every pixel to minimize reconstruction error. Furthermore, Li et al. (LI; WU; CHUANG, 2012) proposed to use Stein's unbiased risk estimator (SURE) (STEIN, 1981) to select the best parameter for the spatial term of a cross-bilateral filter. Rousselle et al. (ROUSSELLE; MANZI; ZWICKER, 2013) extended this idea to apply the SURE metric to choose the best of three candidate filters. Moon et al. (MOON; CARR; YOON, 2014) estimated the error for discrete sets of filter parameters using a weighted local regression. Bauszat et al. (BAUSZAT et al., 2015) posed the filter selection problem as an optimization and solved it with graph cuts. Although these methods attempt to minimize the error between the filtered image and the ground truth, they are limited and often have artifacts due to poorly-chosen filter candidates.

More recently, several techniques based on machine learning were proposed. The first one was proposed by Kalantari, Bako and Sen (2015), which employs supervised learning on a multilayer perceptron neural network to drive the weights of either a joint bilateral or a joint non-local means denoising filter. They first train the neural network by minimizing the error between the filtered and the ground truth image. Then, at run-time, the neural network estimates the parameters for the denoising filter to produce the final image. A more general approach was proposed by Bako et al. (2017), which instead of using explicit hardcoded filters, leverages a deep convolutional neural network (CNN) to allow more general kernels. This technique (KPCN) was later improved (VOGELS et al., 2018), enabling temporal and multiscale filtering, requiring less reference images on the training set, and allowing more control of the variance-bias tradeoff.

A CNN-based method that employs adaptive sampling was proposed by Kuznetsov, Kalantari and Ramamoorthi (2018). It uses two CNNs, both in a encoder-decoder

architecture. The first network uses a 1 spp noisy image to create a sampling map, which is then used to perform adaptive sampling with the remaining sample budget. The resulting image is then denoised by the second network. The encoder-decoder architecture is a simpler version (without the recurrent connections) of the one used by Chaitanya et al. (2017).

A sample-based approach — which uses individual sample values instead of pixel values and statistics — was proposed by Gharbi et al. (2019). It adopts a kernel-predicting CNN similar to Vogels et al. (2018), but operating on individual samples. Kettunen, Härkönen and Lehtinen (2019) proposed a combination of gradient-domain path tracing with a CNN-based reconstruction method.

Generative Adversarial Networks (GANs) (GOODFELLOW et al., 2014) have obtained impressive results tasks like image generation and restoration. The idea is to setup a competitive game between a generator and a discriminator network. While the generator tries to fool the discriminator by generating perceptually convincing images, the discriminator tries to distinguish the generated images from the real targets. Xu et al. (2019) proposed the use of GANs for Monte Carlo denoising, focusing on obtaining images of better perceptual quality, as opposed to using fixed image-space metrics. Similarly to KPCN (BAKO et al., 2017), their strategy also process diffuse and specular components separately.

All of these techniques have strengths and weaknesses in terms of the scene features they can satisfactorily handle, memory costs, execution time, etc. All these variables make a direct comparison of the various algorithms difficult. Without an independent, rendering-system agnostic benchmarking framework, the answer to the question *what combination of MC denoising algorithm and rendering system is most appropriate for rendering a given scene under a given time budget?* is non-trivial. Our framework is intended to fill-in this gap. Hopefully, it will help developers better understand the interplay among the various involved elements and available metrics, shedding some light on the occasional situations in which publications seem to disagree about the quality rank of different techniques.

3.4 Sampling Algorithms

An important class of techniques that also impact on the final image quality generated by Monte Carlo renderers, and that usually do not fall in the category of

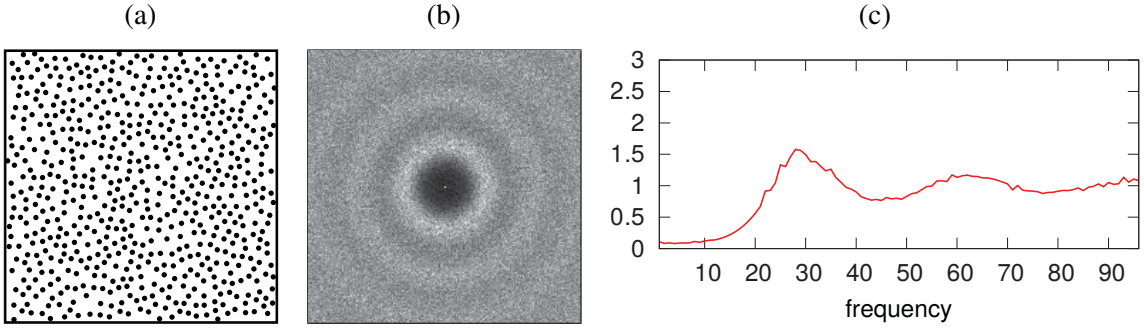
"denoising techniques" presented in Section 3.3, are *sampling techniques* (or *samplers*). These techniques form an important part of any MC rendering system, and are responsible for generating high-dimensional sample positions, which are used to: sample, and ultimately integrate using MC, the light transport equation; sample lens and time for distributed effects like motion blur and depth-of-field; and sample positions on the image plane for pixel reconstruction.

Sampling techniques impact both the amount of aliasing — which was first identified in rendering by Crow (1977) — and noise in the rendered image. The common approach is to use random sampling patterns to replace aliasing by noise — since aliasing is considered to be a more visually objectionable artifact than high-frequency noise. This approach was first used by Cook (1986) and Dippé and Wold (1985), who proposed the use of random distributions based on Poisson disk and jittered sampling.

Many techniques for generating *high-quality* sampling patterns have been developed. The term *high quality* used broadly here means: sampling patterns that produce visually good results. What precisely characterizes a good sampling pattern is a complex subject, since the visual quality of the resulting rendered images is hard to predict. Some commonly cited characteristics of such sampling patterns are: *blue noise* spectrum, *low discrepancy*, and the ability to maintain those properties when projected in lower dimensions.

Blue noise is a characteristic of the distribution's power spectrum that can be summarized as: central DC peak surrounded by a annulus of low energy, followed by a sharp transition region, and a flatter high-frequency region (LAGAE; DUTRé, 2008), as shown on Figure 3.1. Discrepancy is a quantitative measure of how much equidistributed the samples are, where very equidistributed samples have low discrepancy (SHIRLEY, 1991).

Figure 3.1: Typical blue-noise power spectrum. (a) Spacial sampling distribution. (b) Power spectrum. (c) Radial mean profile. (Figure adapted from Wei and Wang (2011))



3.4.1 Blue-noise/Poisson Disk Sampling

Poisson-disk samples are uniformly-distributed patterns where no two points are closer than a fixed minimum distance. The idea of using Poisson-disk sampling in rendering applications (DIPPÉ; WOLD, 1985; COOK, 1986) was inspired by previous investigations on the distribution of photoreceptors in the eyes of Rhesus monkeys (YELLOTT, 1983).

Brute-force Poisson-disk sampling by dart throwing — where a sample is generated uniformly and kept in the sequence only if it is farther than a minimum distance from the other samples already kept — is too slow, and several acceleration methods were proposed. An early survey of Poisson-disk sampling methods, which compares the quality of various algorithms (LAGAE; DUTRÉ, 2006; JONES, 2006; DUNBAR; HUMPHREYS, 2006; WEI, 2008), was made by Lagae and Dutré (2008). It concluded that the (DUNBAR; HUMPHREYS, 2006) was the fastest accurate technique (but still slow compared to approximate ones), and their corner-based Poisson disk tiles approximate technique (LAGAE; DUTRÉ, 2006) produced samples with the best spectral quality among the approximate techniques, while the technique proposed by (KOPF et al., 2006) was the fastest but with below average spectral properties.

Wei (2008) proposed a parallelization scheme that subdivides the sampling domain into grid cells, and concurrently draws samples from cells that are far apart. A slower, but more accurate method was proposed by Ebeida et al. (2011). This method can produce correct unbiased results and can be implemented on the GPU, but it is not as fast as approximate methods. Kalantari and Sen (2012) introduces a faster approximate method that uses a initial set of points generated using any Poisson-disk procedure, and replicates this initial set to fill the whole sampling space using a process similar to convolution, preserving the blue-noise property, but introducing some error.

An important observation made by Mitchell (1991) was that an n-dimensional Poisson-disk distribution is not ideal for general integration problems in graphics: it is good for distribution on the image plane to have the Poisson-disk property, but on the other dimensions, it is beneficial to have a more samples more widely spaced. A construction for n-dimensional distributions that maintain the Poisson-disk characteristic under projection onto lower-dimensional subsets was proposed by Reinert et al. (2015).

Yuksel (2015) proposed a simple greedy Poisson-disk sampling algorithm based on sample elimination. The algorithm works well in high dimensions and can produce progressive sample sets. Given an input sample set, the algorithm tries to find a subset with

the largest Poisson disk radius. This is an NP-Complete problem, hence the proposed greedy method is approximate — does not guarantee maximal coverage — and has $O(N \log N)$ time complexity on the number of samples.

From the power spectrum point-of-view, an idealized blue-noise distribution should have a step-like spectrum — zero power in frequencies lower than v_0, and constant in higher frequencies. Such sequencies are able to recover frequencies below v_0 and map higher frequencies to white noise, hence larger v_0 values are desired. Heck, Schlömer and Deussen (2013) noted that such ideal power spectra are not always realizable: a sample distribution with this power spectrum does not exist. They proposed a solution to increase the zero-frequency radius v_0 — while keeping the spectrum realizable — by adding an adjustable mid-frequency peak centered in v_0. This approach sacrifices spectral quality, introducing low-frequency artifacts and mid-frequency aliasing. Another technique that allows larger v_0 values, producing larger alias-free low-frequency regions was proposed by Kailkhura et al. (2016). It uses as stair-like power spectrum shape, instead of a single peak. Recently, a technique for high-dimensional blue-noise sampling coverage guarantees and approximately smooth step-like spectrum was proposed by Mitchell et al. (2018).

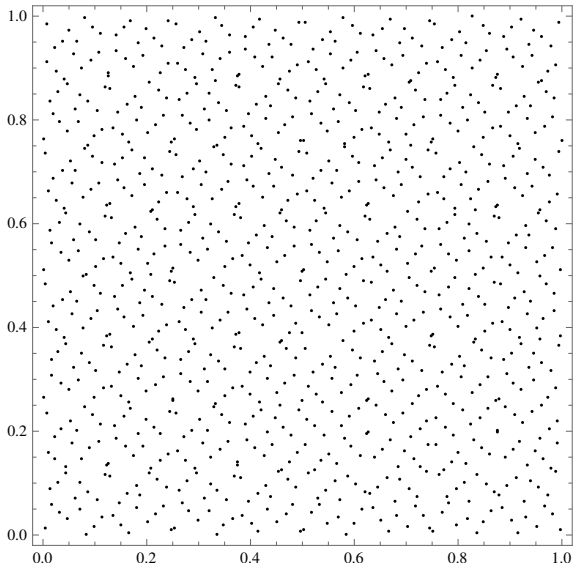
3.4.2 Low Discrepancy Sampling

The use of *discrepancy* to evaluate the quality of sampling patterns in computer graphics was first introduced by Shirley (1991), and later developments were done by Mitchell (1992), Dobkin and Mitchell (1993). Some theoretically good low-discrepancy patterns do not perform as well as expected when used for image sampling. An arbitrary-edge discrepancy measure that better reflects antialiasing behavior was proposed by Dobkin, Eppstein and Mitchell (1996).

The idea of using a deterministic (quasi-random) low-discrepancy sequence for sampling was introduced by Mitchell (1992). Such quasi-random sequences for the basis for quasi-Monte Carlo methods ((NIEDERREITER, 1992)). A popular quasi-random low-discrepancy sequence used in modern PBR systems is based on a the Sobol sequence ((SOBOL', 1967)). Figure 3.2 shows an example with 1024 sample points generated using the Sobol method.

A recent technique for generating low-discrepancy sequences with improved spectral characteristics was proposed by Perrier et al. (2018). The technique achieves almost blue-noise spectrum in 2D projections of the sampling domain, while inheriting the efficiency

Figure 3.2: First 1024 Sobol sample points in 2D.



and low discrepancy of quasi-random techniques.

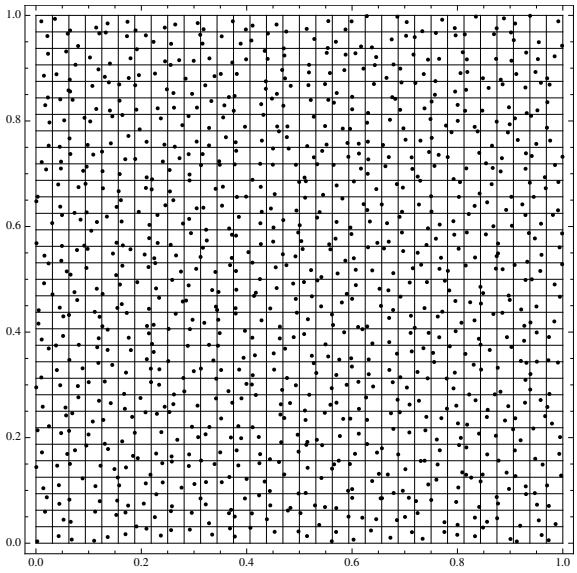
3.4.3 Stratified/Jittered Sampling

A good discussion on the benefits of stratified sampling patterns over random patterns was done by Mitchell (1996). It shows that stratified sampling performs better the smoother the function being sampled is. For scenes with high-dimensional scene functions with possible high-frequency content, the benefits of sophisticated stratified patterns decrease. Figure 3.3 shows an example with 1024 sample points generated using stratification.

Chiu, Shirley and Wang (1994) proposed a 2D sampling technique based on randomly shuffling the x and y coordinates of a canonical jittered pattern. A improvement that produces much better results by using the same permutation for both dimensions was proposed by Kensler (2013). It showed that this approach can produce lower discrepancy than Sobol pattern while being perceptually better (less aliasing artifacts).

Recently, a new technique that achieves results with lower variance, produce high-dimensional samples that preserve stratification across all lower-dimensional projections, while avoiding the structured artifacts found in quasi-random sequences was proposed by Jarosz et al. (2019). The technique is based on the concept of *orthogonal arrays* from statistics.

Figure 3.3: First 1024 stratified sample points in 2D. The samples were stratified in a 32×32 grid.

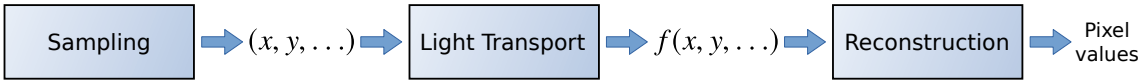


4 PROPOSED FRAMEWORK

A physically-based rendering (PBR) system has to perform several tasks in order to generate an image, for instance: read the scene description file, build an internal scene representation data structure for accelerating ray intersection computations, generate well-distributed sampling positions in a high-dimensional space, compute shading, compute global illumination, reconstruct the final image, and save the result. Some of these tasks hide a significant amount of complexity.

PBR systems implementations usually employ common abstractions that allows some level of modularity and extensibility. For example, it is common practice to have the concept of BSDFs, geometric shapes, cameras, samplers, reconstruction filters, etc., and facilitate adding new implementations for these concepts through the use of inheritance[1]. Figure 4.1 shows the typical pipeline for a Monte Carlo rendering system.

Figure 4.1: Typical Monte Carlo rendering system pipeline: The *sampling* module produces random sampling positions, the *light transport* module computes the radiance values from the sampling positions, and the *reconstruction* module produces the final image pixels from the radiance values.

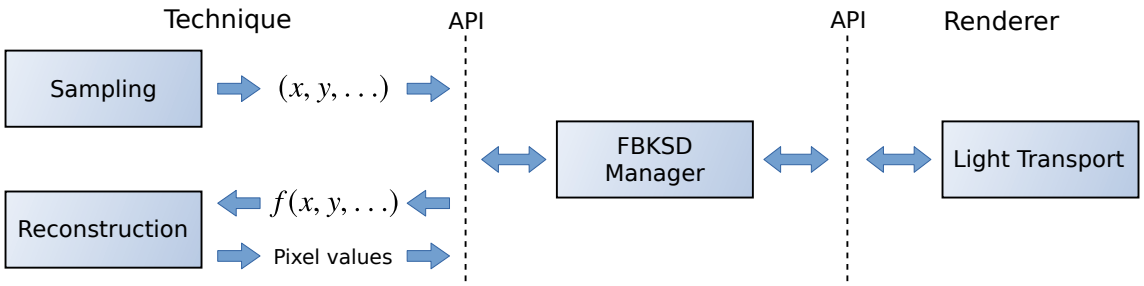


Although different renderers use similar abstractions, implementing a new technique — a new denoising filter, for example — still requires choosing a particular renderer. This makes it hard to compare techniques implemented in different systems. Our *Framework for Developing and Benchmarking Sampling and Denoising Algorithms for Monte Carlo Rendering* (FBKSD) avoids these limitations by decoupling the implementation of a new technique from any specific rendering system. This is done by hiding the specific sample value computation details associated to each rendering system behind a general sampling evaluation interface (API). Thus, it allows for any technique to be seamlessly integrated with different rendering systems, and provides a direct and simple mechanism for comparing these techniques. Figure 4.2 shows the rendering pipeline using our proposed API.

Our system assumes that a PBR system is a black box that receives sample positions as input, and outputs radiance values and other features for each sample position. The internals of this black box typically include global illumination calculations, BSDF evaluations, ray intersection computations, etc. These internal details are highly coupled

[1]Object-oriented programming.

Figure 4.2: Rendering pipeline using the FBKSD system: we modified the typical pipeline by factoring out the sampling, reconstruction, and light transport modules into separate processes. The communication between the processes is done through our API and is intermediated by the *FBKSD Manager* module.



and renderer-dependent, so we try to avoid disrupting their interplay. Therefore, we define the scope of the techniques supported by our system as the set of techniques that can reside outside the black box, namely: *sampling techiques*, which generate the sample positions that are used as input, and *reconstruction or denoising techniques*, which reconstruct the final image pixels from the radiance values return by the PBR system. Note, however, that although we make this distinction between sampling and denoising techniques, some techniques (adaptive denoising techniques) do both the job of sampling and reconstruction, in this case we classify them as just *denoising techniques*, since that is their main task.

4.1 API Overview

To be able to perform their job, sampling and denoising techniques may need to:

- Request information about the scene being rendered, like the dimensions of the image and the total amount of samples the technique can request/generate (sample budget). This is required by all techniques;
- Provide sample positions as input to the renderer. This is only required by sampling techniques, or advanced denoising techniques that also perform adaptive sampling;
- Read the sample values computed by the renderer. This is only needed by denoising techniques, since they need to reconstruct the image form the sample values.

Our system system provides a generic, renderer-agnostic API that supports these operations.

Also, as explained in Section 2.5, different techniques may have different requirements when it comes to sample data access: a simple box filter, for example, only needs the RGB color for each sample, while a more advanced denoising technique may also require several geometric features like world position, normals, etc. Our API allows techniques to

customize which sample components they need access to — what we call *sample layout*. The sample layout determines which sample components are present in the buffer used to deliver sample data to the technique.

Implementing a technique using our API has three main advantages: the technique can be written in a more generic and modular way, without having to deal with details of a particular renderer; it has access to a bigger set of scenes from all the renderers ported to our system; and it allows a convenient and fair way of comparing the results against existing techniques implemented using our API. Those advantages make the proposed system a good test bed for developing new techniques, as well as evaluating existing ones. For the complete C++ and Python API references, refer to Appendices A and B.

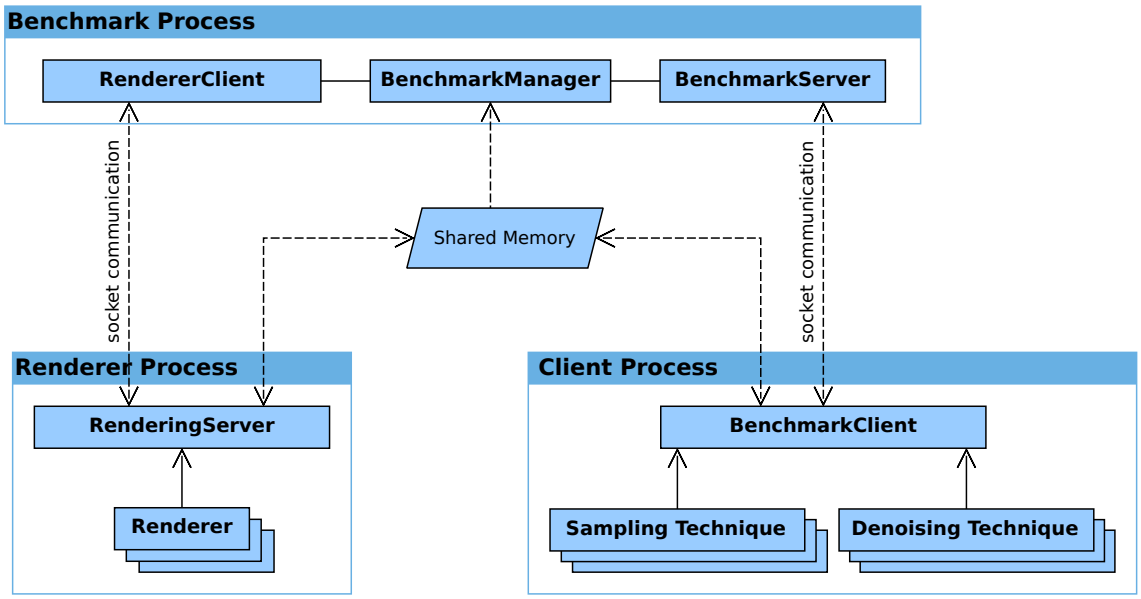
4.2 Main Components

The core of our system consists of a software development kit (SDK) that provides: C++ and Python libraries that support adding new techniques and renderers, a command line interface (CLI) used to manage and run benchmarks, and an interactive results visualization page used to display benchmark results. The libraries provide the API needed to make techniques (samplers and denoisers) compatible with FBKSD, so they can be benchmarked and compared using Image Quality Assessment (IQA) metrics. The libraries also support adapting renderers to be used as rendering back-ends, and implementing new IQA metrics to be used when computing benchmark results.

Figure 4.3 shows an overview of the architecture of the proposed system. There are three main components: the *client process*, the *benchmark process*, and the *renderer process*. The client process implements the technique being evaluated (e.g. a denoising or sampling technique). The rendering process is the actual rendering system that computes the sample values requested by the client process. The benchmark process controls and mediates the overall execution, and saves the final image along with some useful information, such as execution times.

Figure 4.4 shows a sequence diagram of a typical benchmark execution. The benchmark process is executed with a list of parameters specifying the scenes to be rendered, the sample budget for each scene, and the techniques to be evaluated. The rendering process is executed for each new scene, while the client process is executed for each new scene and sample budget. When the client process begins, it starts making several requests, which usually follow this sequence: request information about the scene

Figure 4.3: Main components of the system.



being rendered, set a sample layout, request sample values, and send the resulting image to the benchmark process. The benchmark process mediates the communication between the client and rendering processes, while keeping track of the execution time and sample budget limits. The samples requested by the client process are delivered in tiles. Once the client process receives the requested samples, it reconstructs the final image and sends it to the benchmark process.

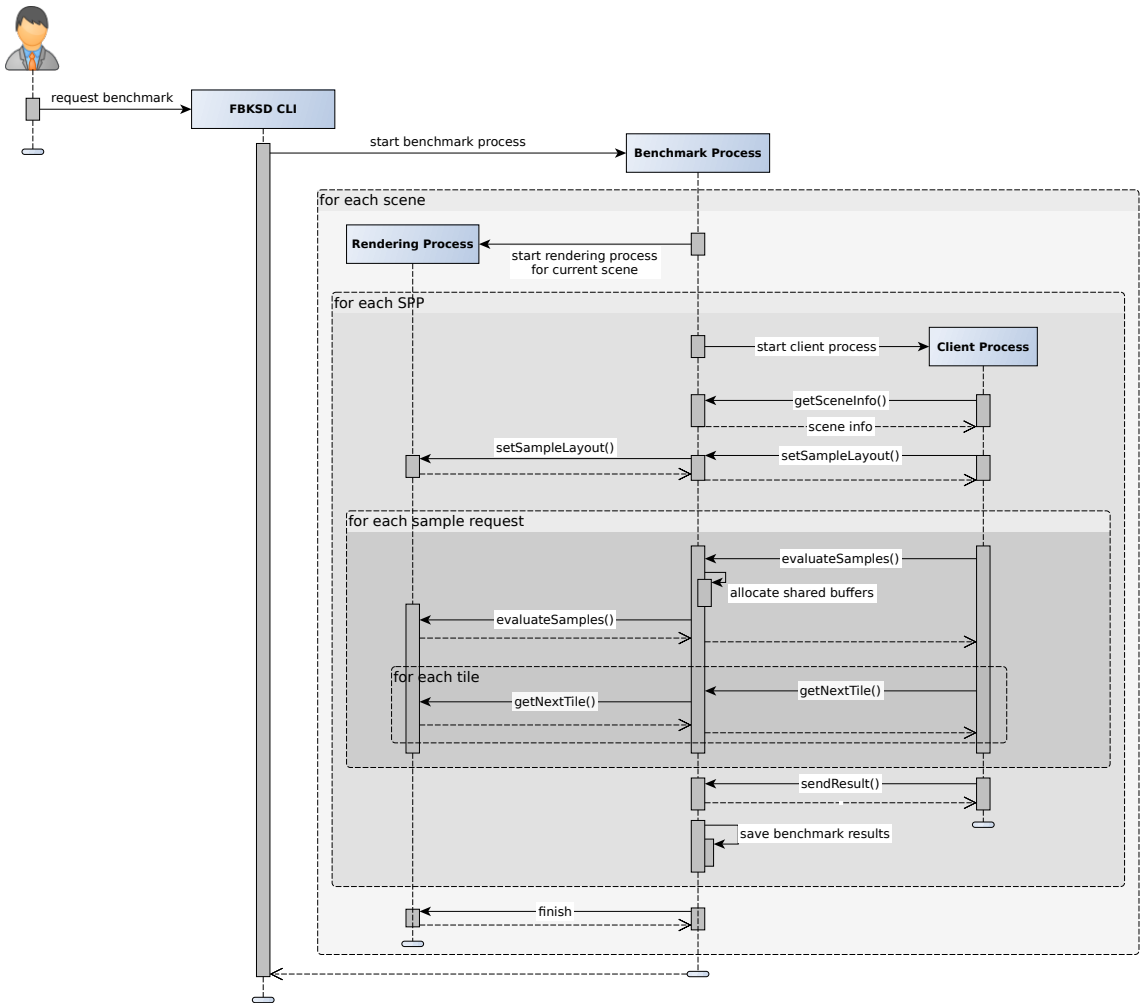
The separation between client and rendering processes allows us to provide a clean API to the client process, simplifying the task of implementing a new technique. Once a new technique is implemented using this renderer-agnostic API, it can be readily executed on a variety of scenes and compared against other techniques.

On the renderer side, this separation allows us to provide different renderers as back-ends to the system, which increases the availability of scenes. When rendering a scene, the client process does not need to know the specific renderer being used. This also tests the robustness of the technique to variations in sample values computed by different renderers.

4.2.1 Client Process

The client process implements a sampling or a denoising technique. Depending on the kind of technique being implemented, this process is responsible for generating sample positions — in the case of sampling techniques or denoisers that also perform

Figure 4.4: Sequence diagram of a typical execution of the system.

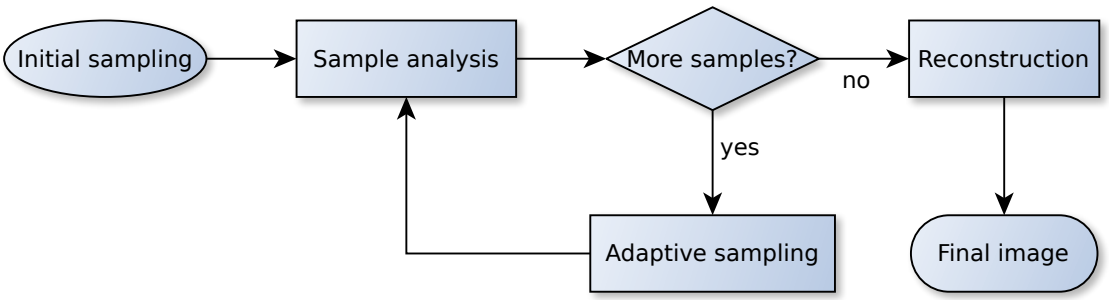


adaptive sampling — and generating the final image from the sample values computed by the renderer.

The system expects techniques to follow the structure shown in Figure 4.5. This structure is general enough to cover a large variety of techniques, including MC denoising — adaptive, non-adaptive, a priori and a posteriori (ZWICKER et al., 2015) — as well as sampling techniques.

When the client process starts, it is given a sample budget. In the *initial sampling* step, the technique decides what portion of the sample budget to spend initially. If the technique is non-adaptive, the entire budget is spent in this step. Otherwise, one or more iterations of *sample analysis* and *adaptive sampling* are performed, until the sample budget is completely consumed. After the final image is reconstructed, the client process finishes. Besides the sample budget, the client has access to more information about the scene being rendered through a scene information querying API. This information allows the technique to adjust its parameters depending on the characteristics of the scene.

Figure 4.5: Structure for techniques supported by our system.



Our framework is general enough to support advanced techniques with adaptive sampling, allowing them to generate sample positions based on information from previous iterations. If the technique does not perform adaptive sampling, the renderer itself generates the sample positions using a uniform sample pattern generator.

4.2.2 Benchmark Process

The benchmark process manages the system execution and mediates the communication between client and rendering processes (Figure 4.4). It is responsible for starting the rendering process with the scene to be rendered, and later, collecting the computed samples to be forwarded to the client technique. The benchmark process keeps track of the current sample budget and client process execution time, and also saves the image reconstructed by the client process, along with an execution log.

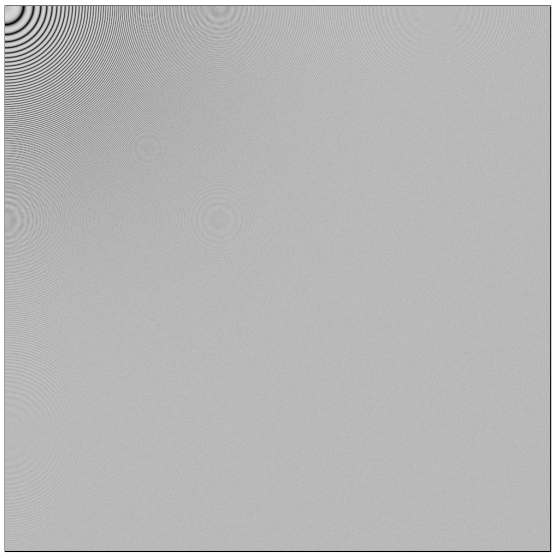
4.2.3 Rendering Process

The rendering process consists of a rendering system that has been instrumented to communicate with the benchmark process and provide the required API endpoints. It is responsible for computing the samples needed by the client process, as well as providing information about the current loaded scene. To help instrumenting existing rendering systems, we provide a few auxiliary classes that implement the necessary API and help collecting the sample data throughout the system.

4.3 Scenes

Our system includes two general categories of scenes: *production*, and *experimental*. The first category includes scenes one would usually find in a production environment. They usually contain more detailed geometry and textures, a bigger variety of illumination settings, and aesthetically pleasing results. The second category includes scenes we designed specifically to stress certain aspects of the filters. Figure 4.6 shows examples of scenes from both categories. By including a good variety of scenes in both categories, we hope to avoid biases when comparing different techniques.

Figure 4.6: Examples of *production* (left) and *experimental* (right) scenes. While production scenes provide a combination of several features (global illumination, motion blur, etc.), the experimental ones are design to stress specific aspects of the techniques under test.



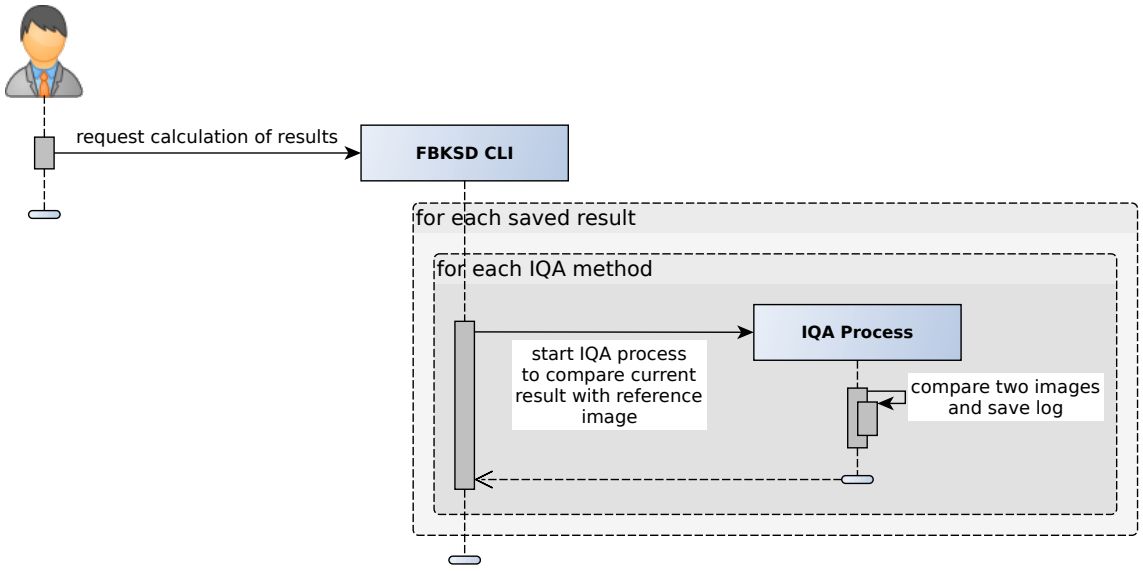
When evaluating a scene, we consider two main aspects: *features* and *noise sources*. Features are legitimate details that denoising techniques must preserve, like textures and materials, geometric details, shading highlights, etc. Noise sources are elements that introduce undesired noise artifacts, like camera effects (motion blur and depth-of-field), glossy materials, area lights, and indirect illumination.

4.4 Image Quality Assessment

After executing a benchmark, FBKSD uses *Image Quality Assessment* (IQA) metrics to compare the *test images* produced by the techniques with *reference images* computed

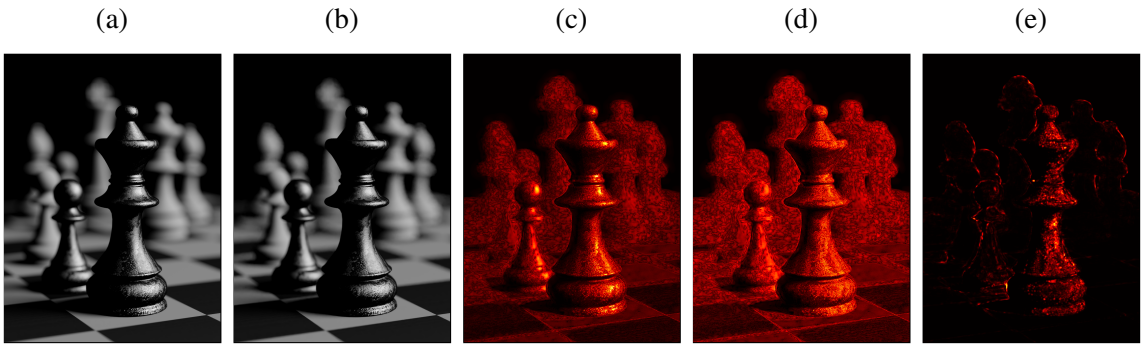
with a large number of samples. The values produces by each IQA metric are compiled in the charts displayed on the results visualization page. Figure 4.7 shows a sequence diagram of the results computation process. Besides the included IQA techniques (MSE, rMSE, PSNR, and SSIM), our framework also allows users to implement new IQA methods. The definitions for the provided IQA methods are presented in the next subsections.

Figure 4.7: Sequence diagram showing the computation of benchmark results using Image Quality Assessment techniques.



IQA techniques can also produce error maps, which are also displayed in the results page. The error maps are images with larger pixel values indicating larger error. The images are color mapped using a heat color map (Figure 4.8).

Figure 4.8: Error maps produces by the IQA methods included in FBKSD: (a) Reference image. (b) Image produced by the NFOR denosing technique with 16 spp. (c) MSE error map. (d) rMSE error map. (d) SSIM error map.



4.4.1 Mean Squared Error (MSE)

The Mean Squared Error (MSE) is the most commonly used metric to asses the error between two images. The MSE definition is given in Equation 4.1.

$$\mathrm{MSE} = \frac{1}{N}\sum_{i=1}^{N}(\hat{c}_i - c_i)^2\,, \tag{4.1}$$

where N is the number of pixels in the image, and $\hat{c}_i$ and c_i are the color values of the pixel i from the test and reference images, respectively.

4.4.2 Relative Mean Squared Error (RMSE)

The Relative Mean Squared Error (RMSE) is a modification of the standard MSE proposed by Rousselle, Knaus and Zwicker (2011). When compared to the standard MSE, the RMSE metric gives more importance to the darker regions of the image. This is a desirable characteristic, since the human visual system is more sensitive to noise in dark areas. Equation 4.2 gives RMSE definition.

$$\mathrm{RMSE} = \frac{1}{N}\sum_{i=1}^{N}\frac{(\hat{c}_i - c_i)^2}{c_i^2 + \epsilon}\,, \tag{4.2}$$

where N is the number of pixels in the image, $\hat{c}_i$ and c_i are the color values of the pixel i from the test and reference images, respectively, and ϵ is a small value to avoid division by zero.

4.4.3 Peak Signal-to-noise Ratio (PSNR)

The Peak Signal-to-noise Ratio (PSNR) is another popular image quality assessment method. It is most commonly used in evaluating quality of lossy compression codecs. The PSNR definition is given in Equation 4.3.

$$\mathrm{PSNR} = 10\log\left(\frac{255^2}{\mathrm{MSE}}\right)\,. \tag{4.3}$$

4.4.4 Structural Similarity (SSIM)

The Structural Similarity (SSIM) (Zhou Wang et al., 2004) method is based on the hypothesis that the human visual system (HVS) is highly adapted for extracting structural information. The method then tries to estimate the perceived changes in structural information variation between two images.

To compute the SSIM between a reference image X and a test image Y, the method first calculates statistics from small windows $\boldsymbol{x}$ and $\boldsymbol{y}$ around each pixel of the two images X and Y, respectively:

$$\begin{aligned}
\mu_x &= \sum_{i=1}^{N} w_i x_i \\
\mu_y &= \sum_{i=1}^{N} w_i y_i \\
\sigma_x &= \left(\sum_{i=1}^{N} w_i (x_i - \mu_x)^2 \right)^{\frac{1}{2}} \\
\sigma_y &= \left(\sum_{i=1}^{N} w_i (y_i - \mu_y)^2 \right)^{\frac{1}{2}} \\
\sigma_{xy} &= \sum_{i=1}^{N} w_i (x_i - \mu_x)(y_i - \mu_y),
\end{aligned}$$

where N is the number of pixels in each window, x_i and y_i are the color values of pixel i from the windows $\boldsymbol{x}$ and $\boldsymbol{y}$, and w_i are the weights of a normalized Gaussian kernel ($\sum_{i=1}^{N} w_i = 1$) — in our implementation, we use a Gaussian kernel with standard deviation 1.5. Using these statistics, the SSIM index between two windows is given by:

$$\mathrm{SSIM}(\boldsymbol{x}, \boldsymbol{y}) = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)},$$

where C_1 and C_2 are constants — in our implementation, we use $C_1 = 6.5025$ and $C_2 = 58.5225$. The final SSIM value for the whole images X and Y is calculated by averaging the SSIM index from all the M windows:

$$\mathrm{SSIM} = \frac{1}{M} \sum_{i=1}^{M} \mathrm{SSIM}(\boldsymbol{x}_i, \boldsymbol{y}_i). \tag{4.4}$$

4.5 Implementation Details

4.5.1 Inter-process Communication and Tile-based Data Transfer

During a benchmark execution, the inter-process communication between the three processes (renderer, benchmark, and client) is done using TCP socket messages on predefined local ports, and a shared memory buffer. The TCP messages are mainly for synchronization purposes and do not transfer large amounts of data. The shared memory buffer is used for transferring the sample data requested by the client and generated by the renderer, as well as the result image generated by the client.

Since we use a shared buffer to transfer sample data between the processes, trying to allocate a buffer big enough to hold all the sample budget available to the client, the size of this buffer could be a problem. This is because some techniques require many sample components. To prevent this problem, we avoid allocating a full-sized buffer by using a tile-based approach.

A parallelization strategy commonly employed by rendering systems is to partition the image being rendered in tiles — small pieces of the full image, usually consisting of 64×64 pixels each — and render each tile in parallel. A popular implementation is to use a task queue: a task is created to render each tile, and all tasks are put in a queue. The tasks are then executed in parallel by a pool of threads: when a thread finishes executing a task, it gets another one from the queue and the cycle continues until the queue is empty.

A key observation is that the the maximum number of tasks being executed at the same time is fixed — it usually corresponds to the number of cores available in the machine. This allows us to allocate a buffer with a size based on this fixed number, and synchronize its use with the client process.

When a rendering thread starts executing a task, we allocate a chunk of memory from our buffer to that thread. The thread then renders the tile, saves the sample data in the memory chunk, and finishes. We then communicate to the client that the tile is ready to be consumed by calling a callback function provided by the client. When the client finishes consuming the tile (the callback function returns), we mark that memory chunk as available, so it can be used by another thread. If no memory chunk is available when a task begins (because the client does not consume the tiles fast enough, for example), the thread bocks until one is available.

If the client needs to write on the memory buffer — which is the case for sampling

and adaptive denoisers — we perform an additional communication step: before we make a memory chunk available to a rendering thread, we block it and request the client to write the input sample data on that tile, only then we give the memory chunk to the thread, which will now contain the input data.

4.6 Summary

In this chapter, we presented the main motivation, design decisions, scope, and implementation aspects of our proposed framework (FBKSD). An overview of the proposed API was presented in Section 4.1. Then we presented the main system components, their role, and how they interact (Section 4.2). In Section 4.3, we explained what aspects guide our scene selection/creation process. Section 4.4 showed how *Image Quality Assessment* methods are employed by our system to compute the error of the images produced by the techniques under test. Finally, some implementation details were presented (Section 4.5), including a discussion about how our system avoids large memory overheads by using a tile-based data transfer.

5 ONLINE SUBMISSION SYSTEM

In Chapter 4, we discussed our proposed framework that allows writing and benchmarking techniques in a renderer-independent way. In this chapter, we introduce an online submission system that allows users to submit their techniques and have them benchmarked in a controlled environment, and the results published online.

5.1 Motivation

Being able to download and install our system locally is an important feature that allows authors to quickly prototype new techniques, benchmark them with different scenes, fine-tune their parameters, and compare them with other techniques. However, there are a few caveats with such a local solution, namely:

- To compare their techniques against existing ones, the user needs to download and execute them with all the desired scenes, which can be a computational intensive task if the number of scenes is high;
- This approach does not allow comparing results generated by different users;
- Sharing results with others is not very convenient, since it requires hosting a web server with the results page, or sending a large amount of data.

To solve those limitations, we provide an experimental online submission system (https://fbksd.inf.ufrgs.br). This system allows users to submit new techniques, and have them automatically executed in our server. The results are then made available online, first in a hidden location disclosed only to the technique's author, and later in a public location (if the author requests the publication of the results). This can be an invaluable tool for researchers, rendering systems authors, and anyone interested in sampling and denoising techniques for Monte Carlo rendering.

5.2 Requirements

We specified the following main functional requirements for our online system:

- **User account management:** Users need to be able to manage (create/edit/delete) their accounts. Accounts need to include an e-mail address for account validation

and notification purposes.

- **Project management:** Users need to be able to manage projects. A project contains the source code for a technique.
- **Benchmark execution:** Users can request a benchmark execution for a project. The execution will happen in our server.
- **Private results review:** After a benchmark is executed for a technique, the results are made available to the user in a private location. The user can review the results and decide to publish or discard them.
- **Results publication:** Users can decide to publish the results of a benchmark execution. When the results are published, they become available online in a known public location.

After analysing the requirements, we decided to search for an off-the-shelf solution that could satisfy as many of them as possible, while being flexible enough to allow us to implement the others. The solution we decided to adopt was GitLab (GITLAB, 2019). GitLab is a popular git project hosting solution (akin to GitHub, BitBucket, and others (GITHUB, 2019; BITBUCKET, 2019)) that provides a free self-hosted version with a built-in continuous integration (CI) system. "Self-hosted" in this context means that we are installing GitLab in our own server, as opposed to using their cloud-based alternative. Although this approach incurs the cost of maintaining our own server, it give us better control of the system and provides the level of customization we need.

As a git project hosting solution, GitLab supports user account and project management out-of-the-box. Moreover, the continuous integration (CI) system allows projects to configure a set of tasks that are triggered by certain events (a new commit, or a manual request, for example). This set of tasks (called a *pipeline*[1]), when triggered, is executed by a *runner*[1], which is a special process responsible for acquiring and running tasks. Our idea is to leverage the CI system to enable the remaining functional requirements. To accomplish this, we can configure each request as a task for the CI system. When a task is executed, it calls a custom application in the server to execute the corresponding request. The main remaining challenge is then to implement this custom server application that is responsible for executing the requests from the users.

Another advantage of using GitLab's CI infrastructure is that it supports executing the tasks in a sandboxed environment. This is an important non-functional requirement,

[1]GitLab nomenclature.

since we are dealing with untrusted code submitted online by users, and this code needs to be automatically compiled and executed in our server. GitLab's CI system accomplishes this by running the tasks inside Docker (DOCKER, 2019) containers. A Docker container can be seen as a lightweight virtual machine, used as a sandbox, running on top of the Linux kernel. If a malicious program is executed and is able to gain root privileges, for example, it does so only inside the container, and should not be able to compromise the host machine.

From the point of view of the user, adopting a well-known off-the-shelf solution like GitLab has the benefit of bringing all the collaborative features, for example, bug tracking (users can report bugs in other projects), forks (users can create forks of existing projects), and merge requests (users can contribute to other projects by submitting patches). This creates an environment that makes it easer to do incremental improvements to existing projects, which stimulates development and is more approachable to newcomers.

5.3 Workflow

The workflow for submitting, benchmarking, reviewing, and publishing a technique using our online system is explained next. The requests are done using the online user interface (standard GitLab interface), and should feel familiar to anyone who have used an online git project hosting solution before.

5.3.1 Registration

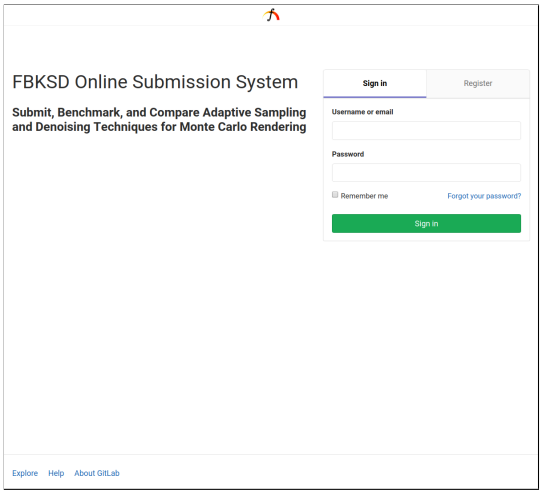
The first step is to create an account in our GitLab server. Figure 5.1 shows the initial page, where the user logs in or creates a new account. After registering, the user receives a confirmation message in the provided e-mail account. Once the account is validated (the user clicks the validation link in the confirmation message), the user can sign-in and create a new project.

5.3.2 Submission

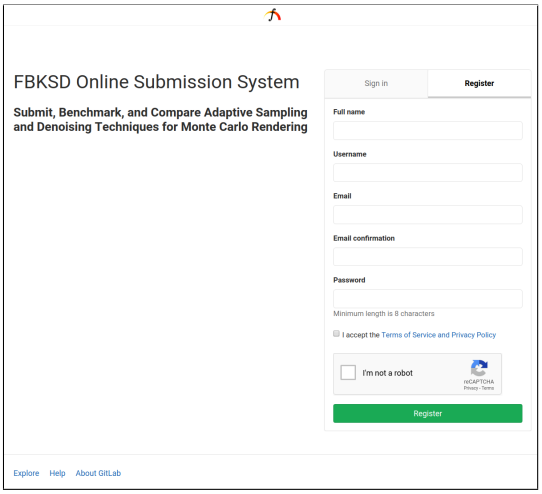
To submit a new technique, the user needs to create a project and upload (push) the technique's source code. Besides the technique's source code, a project may also contain

Figure 5.1: Initial pages shown when a user accesses the online submission system website (https://fbksd.inf.ufrgs.br)

(a) Sign-in page.



(b) Registration page.



any documentation the author wishes to provide. A project can be set as private (only the author can see), public (anyone can see), or internal (only logged-in users can see).

The source code must follow a structure specified in the FBKSD documentation (available at https://fbksd.inf.ufrgs.br), which allow us to automate the build process in our server. Namely, it should include:

- A `info.json` file containing some information about the technique (e.g., technique's type, name, citation, etc.);
- A `CMakeLists.txt` build file;
- A `.gitlab-ci.yml` file, which contains the CI configuration.

Once the source code is uploaded (and every time a new commit is pushed), our server will automatically execute some sanity checks (check if the code follows the expected structure, etc.) and try to compile it. If any error occurs during this process, the user is notified by e-mail. After the code is successfully compiled, the user can then request a benchmark execution.

5.3.3 Benchmarking and Results Reviewing

To request a benchmark execution, the user needs to manually run a pipeline passing the variable `FBKSD_RUN`, as shown in Figure 5.2. When the user requests a benchmark execution, the server will execute the technique with all the scenes internally configured

in our server and generate a results page that is made available online in a secret location disclosed only to the user (Figure 5.3). This page includes the results from the technique that was executed, as well as all other techniques already published by other users. This gives users a chance to compare their techniques with others, and decide if they want to publish the results or tweak their techniques and try again.

Each private results page generated from successful benchmark executions has the format `https://fbksd.inf.ufrgs.br/results/<key>`, where `<key>` is a random 128-bit *universally unique identifier* (UUID) string. This key is used to uniquely identify a benchmark execution, and should be provided by the user when requesting the publication of the results.

5.3.4 Results Publication

Once the user is satisfied with the results, he can then request the publication of the results, which will make them available online, alongside the results from the other published techniques, in a location known by everyone (the public results page https://fbksd.inf.ufrgs.br/results/).

To request the publication of results for a technique, the process is analogous to the one explained in Figure 5.2. The difference is that the variable name should be `FBKSD_PUBLISH`, and the variable value should be the key that identifies the results to be published (obtained from the log page in Figure 5.3(b)).

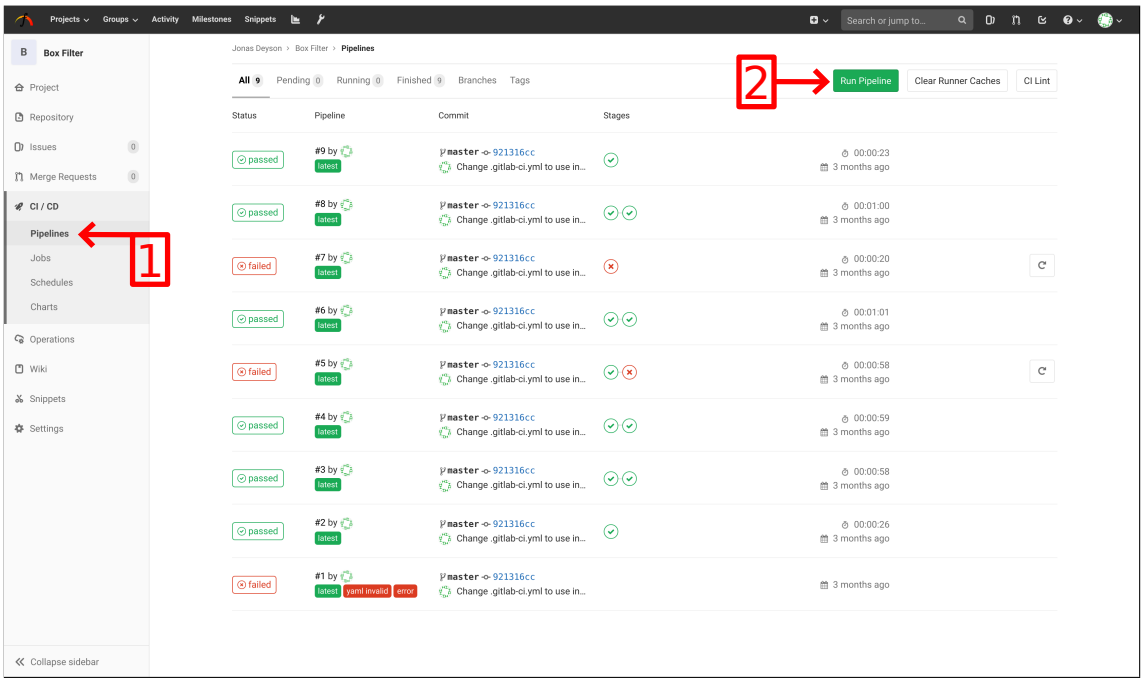
5.4 Implementation Details

As explained in Section 5.2, one important non-functional requirement is isolating the processes dealing with code submitted by the users. More specifically, we have to assume the submitted code is potentially malicious, and should not have direct access to sensitive data.

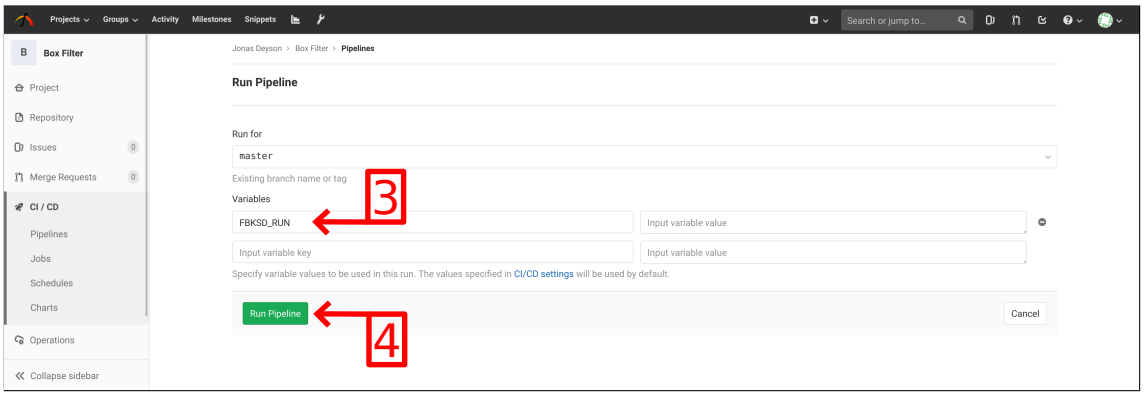
Although the GitLab CI system supports executing tasks inside Docker containers, a new container is created for each task, and is destroyed when the task finishes. Since we need to keep some data after tasks finish (e.g., results data, users metadata, etc.), the containers need to access a database in a permanent storage location on the host machine (outside the container), which is a security risk. To mitigate this problem, instead of giving

Figure 5.2: Pages showing the steps needed to requesting a benchmark execution: click on "CI/CD -> Pipelines" on the left panel (1) to open the page shown on (a), then click on "Run Pipeline" (2) to open the "Run Pipeline" page shown on (b). Type the variable `FBKSD_RUN` (3) and click on the "Run Pipeline" button (4). The pipeline will be queued for execution.

(a)



(b)

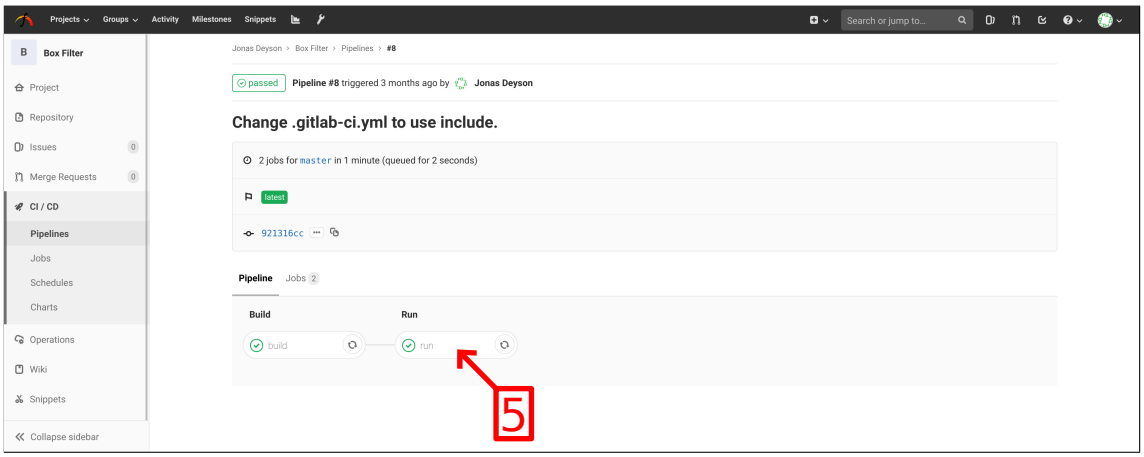


the container that runs the untrusted code direct access to the database, we do so through a separate process, permanently running on a different container. The communication between these two processes — the one running the untrusted code, and the one that has access to the database — is done through a TCP socket API.

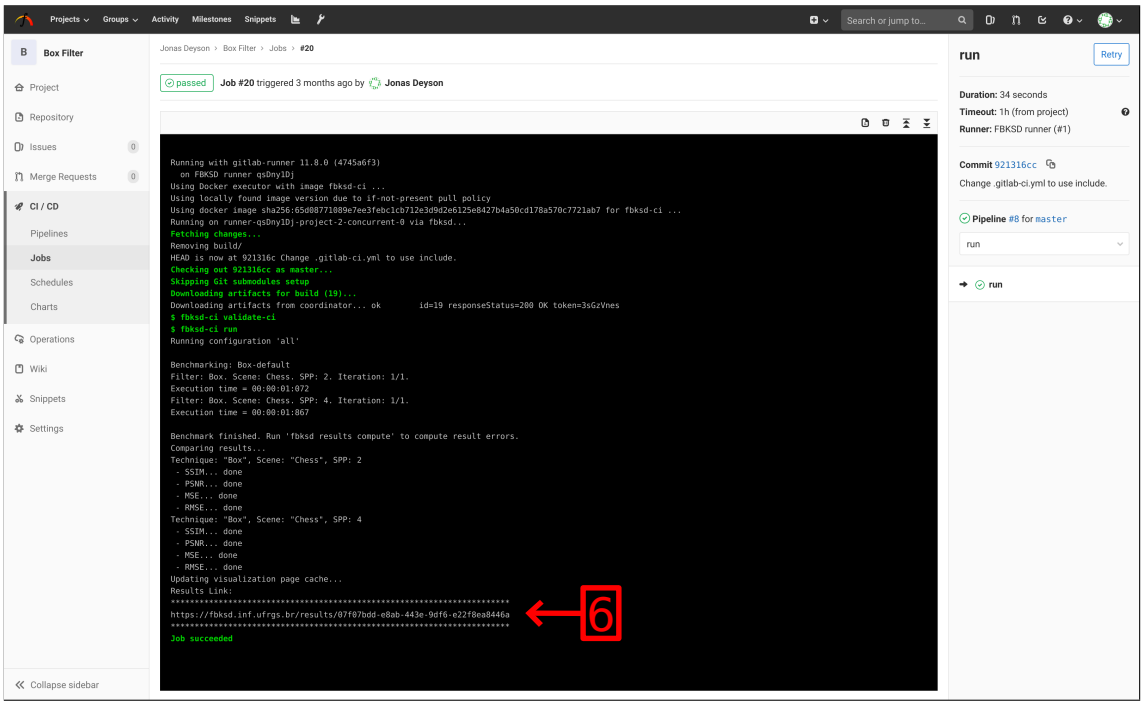
Figure 5.4 shows the main components of the server application. The GitLab runner process constantly polls the web server for new tasks. When a new task is available, the runner runs it by starting the `fbksd-ci` process inside a new container. The `fbksd-ci` process only has direct access to non-sensitive read-only data (e.g. scene files) and a cache

Figure 5.3: Pages showing the pipeline execution status: (a) shows the pipeline status with its two tasks (“build” and “run”) successfully finished (green marks). Click on the “run” task (5) to open the log shown on page (b). In the log, you can see the link for the private results page.

(a)



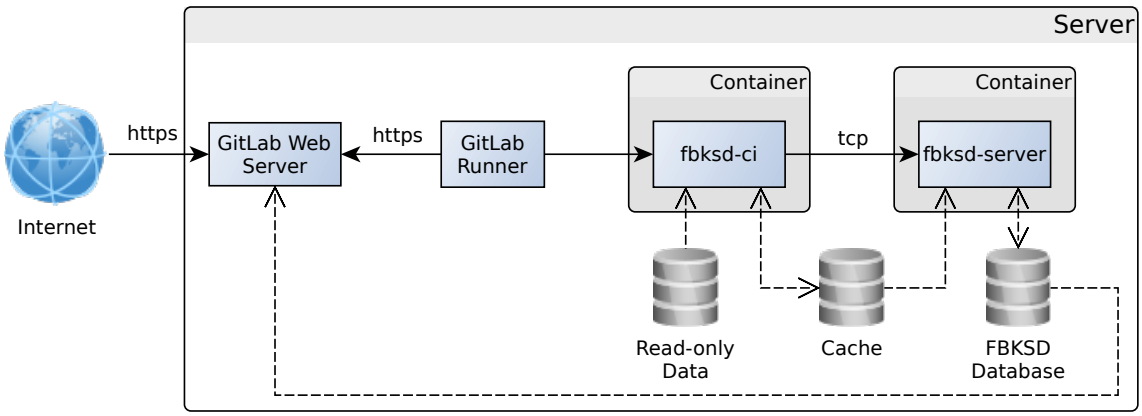
(b)



storage location (e.g. for writing results data when running a benchmark). Sensitive data containing user information, result images, etc., is kept in a location accessible only by the `fbksd-server` process, which runs in a separate long-lived container. This process exposes an API on a local TCP port that can be reached by the `fbksd-ci` process. Both `fbksd-ci` and `fbksd-server` were implemented in Rust (RUST, 2019)

Since the techniques submitted by the users are compiled and executed inside a

Figure 5.4: Server application architecture. The arrows with solid lines represent call directions ($A \rightarrow B$ means A calls B). Arrows with dashed lines represent data flow directions. Blue squares represent processes.



Docker container, this container needs to have all the required libraries installed. Also, techniques can have wildly different requirements, for example, a technique written in Python may require a certain Python deep learning framework, while another written in C++ does not. While we do not allow users to install their own packages inside the container, we provide a list of different environments the user can choose from. To select a different environment, the user only needs to edit the `.gitlab-ci.yml` file located in the root directory of the techniques' project according to the documentation we will provide.

Currently, we run both the GitLab web server and the GitLab runner in the same physical machine. This is not an ideal setup because if the web server experiences high load while a technique is being benchmarked, it can potentially interfere negatively on the execution time results for that technique. Fortunately, GitLab already allows the web server and the runner to reside in different machines, the only requirement is that the runner must be able to reach the web server over `https`. We hope to be able to migrate the system to a better setup in the future — separate dedicated machines to host the web server and the runner — once the system matures.

5.5 Summary

This chapter presented a complementing system to the FBKSD SDK — an online submission system that allows users to submit, benchmark, and compare results with other techniques. Section 5.1 presented a motivation for the online system, showing that a local solution — one that users download and execute in their own machines — while important, only goes so far. Section 5.2 gave the requirements the online system needs to fulfil. The

main workflow users need to follow to use the online system is presented in Section 5.3. The workflow has three main steps: registration, submission, and benchmark execution and results reviewing (Subsections 5.3.1, 5.3.2, and 5.3.3, respectively). Finally, some implementation details are presented in Section 5.4, including the overall architecture of server application, and how it deals with security concerns.

6 CASE STUDY

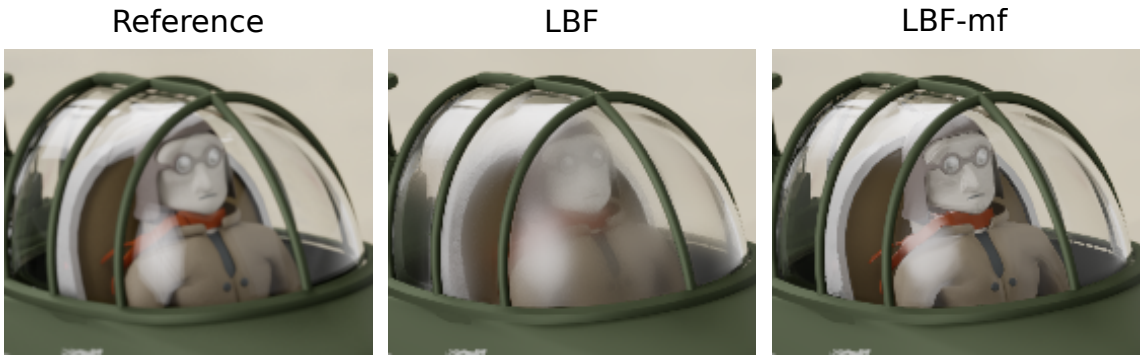
To demonstrate the effectiveness of our system, we adapted a several denoising methods for Monte Carlo rendering: LWR (REN et al., 2013), NFOR (BITTERLI et al., 2016), LBF (KALANTARI; BAKO; SEN, 2015), RPF (SEN; DARABI, 2012), SBF (LI; WU; CHUANG, 2012), RHF (DELBRACIO et al., 2014), NLM (ROUSSELLE; KNAUS; ZWICKER, 2012), RDFC (ROUSSELLE; MANZI; ZWICKER, 2013), GEM (ROUSSELLE; KNAUS; ZWICKER, 2011), and KPCN (BAKO et al., 2017). For this, we have instrumented the original source code provided by the rendering systems' developers and by the authors of these techniques with calls to our API. In the case of NFOR, the authors only provided some high-level pseudo-code, so we had to implement it from scratch. All results were generated on a 4 GHz i7-4790K CPU with 32 GB of RAM.

As rendering beck-ends to our system, We ported three well-known renderers used by the research community: PBRT-v2 (PHARR; HUMPHREYS, 2010a), PBRT-v3 (PHARR; JAKOB; HUMPHREYS, 2016), and Mitsuba (JAKOB, 2010). We have also developed a special-purpose custom procedural renderer (Appendix C), which is capable of rendering scenes consisting of arbitrary mathematical expressions. These four renderers provide a good variety of scenes for our tests.

We provide several examples illustrating the use of the four rendering systems and eight MC denoising algorithms. Some techniques use geometric features from the first intersection point to help them preserve scene details. This strategy tends to perform poorly on scenes with transparent glass and mirrors, as shown in Figure 6.1 (center). To make comparisons among techniques fairer, we implemented modified versions of these techniques using the first non-specular intersection point instead. We indicate the modified versions by a suffix "-mf" (modified features) — Figure 6.1 (right).

Figure 6.7 shows results of a benchmark created with seven MC denoising techniques and nine scenes from our scene pool. The scenes were selected as to form a good representative set of situations that can challenge a denoiser. *Measure One* contains several glossy highlights, and *Measure One Mooving* adds motion blur on top of that. *Crown in Glass* contains intricate bumpy textures with sources of caustics, all behind a layer of glass. *Furry Bunny* and *Curly Hair* contain fine geometric features that can easily be overblurred. *Bathroom* is a typical interior scene with several fine textures reflected by mirrors. *Country Kitchen Night* is a challenging global illumination scene with hidden light sources, being prone to fireflies (artifacts consisting of bright single pixels scattered over the image).

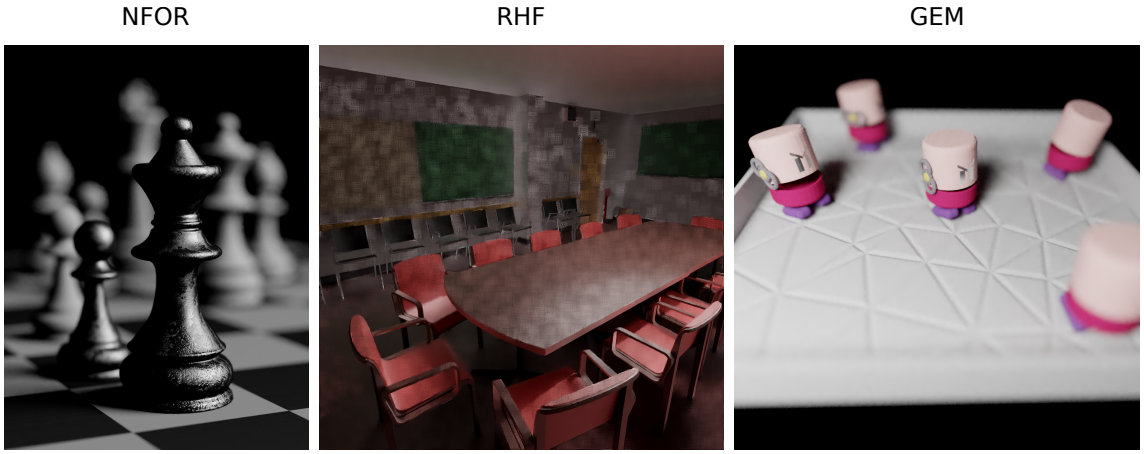
Figure 6.1: Rendering using geometric features. Reference image (left). Overblurring on transmitted scene details caused by relying on features at the first intersection point (center). Using features from the first non-specular intersection allows the denoiser to preserve those details (right).



Finally, *Glass of Water* is a mostly specular scene with many specular highlights.

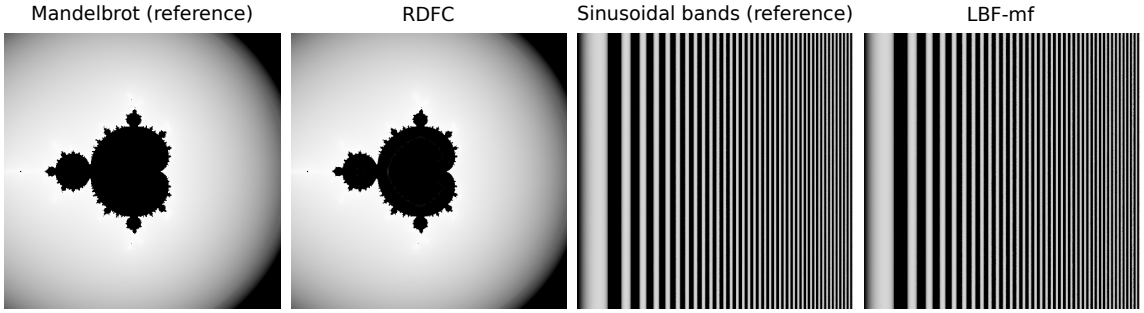
The first row of Figure 6.7 shows thumbnails of the reference images (computed with a very large number of samples per pixel) for the selected scenes. Although the image resolutions vary, their typical size is about 1,024 × 1,024 pixels. A small square highlights a challenging region in each scene. The corresponding regions for the noisy result, for the outputs generated by the various denoising techniques, and for the reference images are shown in the subsequent rows. From the scenes shown in Figure 6.7, *Bathroom*, *Glass of Water*, and *Country Kitchen Night* were rendered using Mitsuba; the remaining six were rendered using PBRT v3. Figures 6.2 and 6.3 show examples of images generated with our framework using PBRT v2 and our custom procedural renderer, respectively.

Figure 6.2: Images generated with our system using PBRT v2 and the techniques NFOR, RHF, and GEM, respectively.



The results in Figure 6.7 show that all techniques have some degree of trouble with glossy highlights, as shown in the scene *Measure One*. The glossy highlights are often

Figure 6.3: Examples of experimental scenes rendered with our system using a procedural renderer. (left) Mandelbrot set. (right) Increasing sinusoidal bands ($\sin(x^2)$).



overblurred or contain patchy artifacts. Glossy highlights are troublesome because the extra features used by the denoisers to tell legitimate scene details from noise do not help detecting the highlights. Another instance of this problem can be seen in Figure 6.4. The subtle checker patterns seen on the reference image (Figure 6.4 (bottom right)) come from a texture applied to the specular component of the material. This specular component is not part of the albedo feature used by the denoisers, causing them the remove the detail.

Back to Figure 6.7, scene *Measure One Moving* is a motion blur version of the previous scene. The strong motion blur effect makes the overblurring of the glossy highlights less visible, but it may also lead to other situations that may cause denoisers to produce overblur. All techniques have trouble preserving the fine motion blur details over the noisy glossy background.

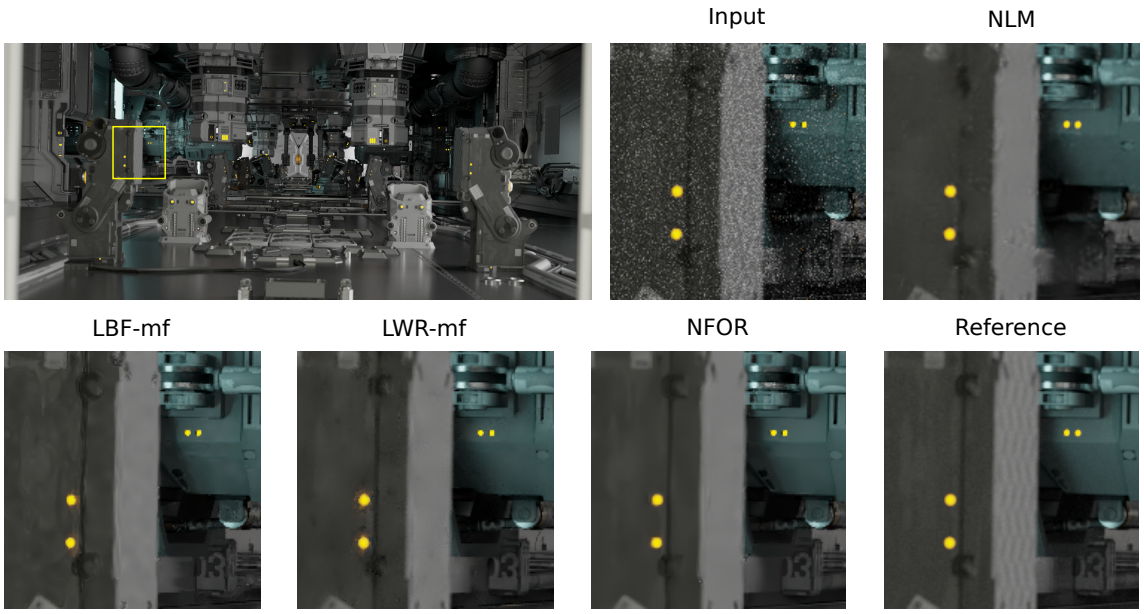
The *Crown in Glass* scene contains bump-mapping details behind a layer of glass. Some techniques do a good job at preserving these details on the less noisy areas (*e.g.*, NLM and RDFC). In darker, noisier regions, all techniques introduce some degree of overblurring.

Very fine geometry details, as commonly found in hair (*Curly Hair*) and fur (*Furry Bunny*) is also a frequent source of problems. Notice that even denoisers that rely on geometric features, as in the case with LBF, can overblur these details — although hair and fur are being captured by the geometric features, the high-frequency detail in the presence of noise constitutes a challenge.

Scenes with very challenging illumination conditions — which translates to high levels of noise — are also problematic. High-energy spikes (fireflies) are very difficult to spread out while preserving energy, causing blob artifacts. As the *Country Kitchen Night* scene example shows, some techniques like *RDFC* do a good job at spreading fireflies, but small variations in the geometry of the scene can cause artifacts.

The results shown in Figure 6.7 and the previous discussion illustrate the potential of

Figure 6.4: Texture details in the specular component of some materials (see the checker pattern on the light gray rectangle in the reference image) are not part of the "albedo" feature, making the denoisers to remove such details.



our framework to provide qualitative assessments of MC denoising techniques, as well as to identify potential limitations of current approaches. As such, our system provides relevant information for guiding future research in the area. Our framework also provides a GUI for interactive exploration of benchmark results, which include quantitative assessments based on several metrics (MSE, rMSE, PSNR, SSIM, and execution time). Figure 6.6 shows the values of rMSE, PSNR, and SSIM for all the examples in Figure 6.7.

The benchmark data corresponding to Figure 6.7 and our web-based GUI can be download from <https://goo.gl/znHKRD>. We would like to encourage the reader to download and explore such material. We also provide a video (<https://goo.gl/cmUqcS>) showing a brief tutorial on how to interactively explore the benchmark results. Since our GUI can used with the results obtained for any technique that uses our framework, together they can be an invaluable tool for the academic and industry communities, as well as for general users. Figure 6.5 shows a snapshot of some of the quantitative information obtained when using our framework to render the *Bathroom* scene using the seven denoising techniques shown in Figure 6.7. The graphs compare the performance of the techniques according to PSNR, rMSE, SSIM, and execution time for 16, 32, 64, and 128 spp.

Figure 6.5: Use of our system's GUI for interactive exploration of the quantitative results generated by a benchmark.

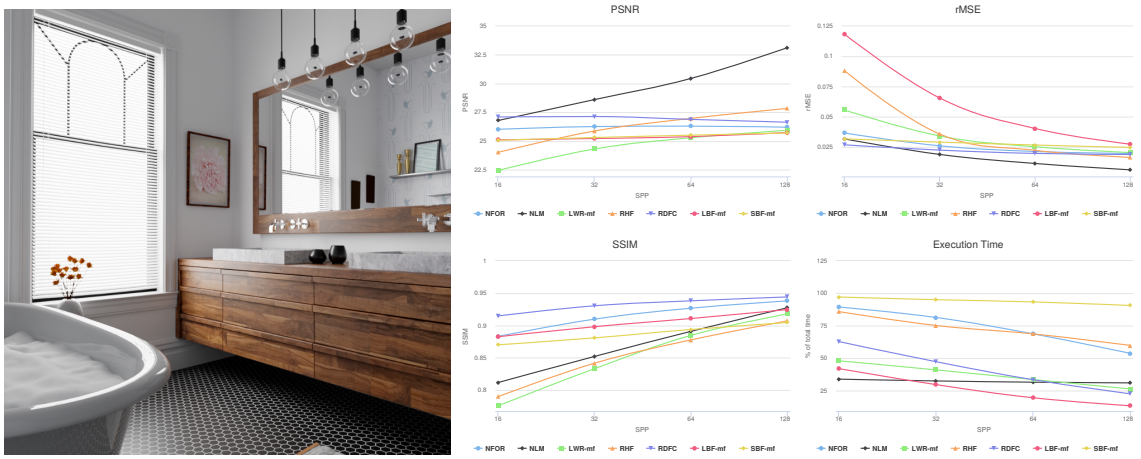


Figure 6.6: Quantitative results for the images shown in Figure 6.7 according to the rMSE, PSNR, and SSIM metrics.

		NLM			LBF-mf			RHF			NFOR			LWR-mf			RDFC			SBF-mf		
	SSP	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM	rMSE	PSNR	SSIM
Curly Hair	16	0.0068	33.4182	0.8559	0.0088	32.3764	0.8240	0.0097	32.0924	0.8249	0.0055	34.0426	0.8894	0.0096	32.1409	0.8138	0.0076	32.8546	0.8428	0.0094	32.0692	0.8239
	32	0.0048	34.8156	0.8879	0.0067	33.3657	0.8527	0.0071	33.3176	0.8558	0.0041	35.2452	0.9114	0.0083	32.7106	0.8265	0.0058	33.9093	0.8700	0.0072	33.1470	0.8457
	64	0.0032	36.6072	0.9200	0.0051	34.3698	0.8843	0.0050	34.8007	0.8892	0.0031	36.4066	0.9294	0.0066	33.6052	0.8502	0.0043	35.1458	0.8985	0.0057	34.1826	0.8676
	128	0.0021	38.4619	0.9451	0.0043	35.0522	0.9076	0.0032	36.5669	0.9207	0.0024	37.5243	0.9442	0.0048	34.8144	0.8817	0.0031	36.5432	0.9248	0.0045	35.1782	0.8877
Measure One	16	0.0114	29.9136	0.8230	0.0145	29.0600	0.8061	0.0111	30.1186	0.8066	0.0095	30.2342	0.8614	0.0150	28.3846	0.8136	0.0104	30.5375	0.8602	0.0113	29.8422	0.8251
	32	0.0087	31.0691	0.8501	0.0112	30.0560	0.8267	0.0076	31.7795	0.8569	0.0068	31.6495	0.8882	0.0097	30.1050	0.8497	0.0071	31.9288	0.8865	0.0085	30.9448	0.8530
	64	0.0065	32.3305	0.8737	0.0082	31.1411	0.8555	0.0054	33.2394	0.8899	0.0050	33.0332	0.9089	0.0064	31.8815	0.8806	0.0052	33.2241	0.9070	0.0063	32.0505	0.8744
	128	0.0046	33.7154	0.8971	0.0057	32.7334	0.8851	0.0038	34.7063	0.9138	0.0037	34.3675	0.9246	0.0044	33.5219	0.9053	0.0037	34.5333	0.9239	0.0047	33.1541	0.8814
Measure One Moving	16	0.0065	32.6101	0.8931	0.0117	30.5025	0.8228	0.0077	31.8505	0.8472	0.0073	31.9393	0.8967	0.0103	30.4283	0.8831	0.0061	32.8040	0.9096	0.0082	31.4845	0.8365
	32	0.0048	33.8975	0.9093	0.0082	31.9737	0.8444	0.0049	33.9239	0.8940	0.0046	33.8510	0.9220	0.0069	32.2812	0.9021	0.0041	34.4503	0.9275	0.0057	33.0608	0.8808
	64	0.0034	35.3087	0.9226	0.0057	33.3805	0.8671	0.0032	35.8296	0.9225	0.0031	35.5887	0.9382	0.0049	33.9451	0.9180	0.0029	36.0028	0.9404	0.0041	34.5003	0.9069
	128	0.0024	36.9161	0.9363	0.0039	34.9353	0.8939	0.0021	37.6765	0.9406	0.0020	37.5113	0.9509	0.0034	35.6560	0.9319	0.0020	37.7502	0.9511	0.0029	36.0110	0.9122
Bathroom	16	0.0318	26.8069	0.8119	0.1184	25.1307	0.8826	0.0882	24.0579	0.7900	0.0367	26.0398	0.8831	0.0557	22.4632	0.7759	0.0269	27.1138	0.9148	0.0318	25.0894	0.8702
	32	0.0189	28.6246	0.8522	0.0656	25.2398	0.8981	0.0358	25.8946	0.8418	0.0261	26.2783	0.9098	0.0342	24.3322	0.8333	0.0225	27.1502	0.9305	0.0291	25.3261	0.8809
	64	0.0115	30.4638	0.8910	0.0404	25.4054	0.9108	0.0223	26.9899	0.8778	0.0214	26.3141	0.9267	0.0253	25.3100	0.8848	0.0198	26.9017	0.9381	0.0267	25.5323	0.8938
	128	0.0062	33.1411	0.9275	0.0274	25.7599	0.9246	0.0165	27.8578	0.9071	0.0197	26.2213	0.9380	0.0205	25.9545	0.9180	0.0189	26.6566	0.9441	0.0247	25.7199	0.9045
Crown in Glass	16	0.0388	25.4848	0.7975	0.0716	22.3721	0.7458	0.0879	23.2257	0.7316	0.0508	24.5644	0.7813	0.0514	24.5724	0.7831	0.0383	25.8028	0.8292	0.1046	23.0870	0.6812
	32	0.0233	27.6083	0.8460	0.0496	23.7700	0.7866	0.0473	25.3741	0.7976	0.0393	25.6622	0.8131	0.0325	26.4063	0.8281	0.0278	27.1110	0.8571	0.0707	24.6677	0.7401
	64	0.0152	29.3789	0.8776	0.0325	25.7598	0.8290	0.0246	27.7020	0.8483	0.0276	26.9539	0.8413	0.0217	27.9946	0.8586	0.0198	28.4026	0.8780	0.0473	25.9973	0.7846
	128	0.0105	31.0564	0.9022	0.0223	27.7330	0.8632	0.0140	29.9011	0.8854	0.0178	28.6235	0.8705	0.0145	29.5715	0.8813	0.0138	29.8623	0.8959	0.0288	27.4464	0.8259
Furry Bunny	16	0.0125	30.7409	0.8577	0.0298	27.9409	0.8093	0.0175	29.7712	0.8426	0.0079	32.2501	0.9131	0.0211	29.0407	0.8246	0.0177	29.7484	0.8461	0.0190	29.3658	0.8348
	32	0.0082	32.3536	0.8948	0.0218	29.0784	0.8344	0.0116	31.3199	0.8791	0.0058	33.5862	0.9313	0.0185	29.6635	0.8395	0.0132	30.8930	0.8715	0.0125	30.9737	0.8718
	64	0.0057	33.8900	0.9216	0.0134	30.8606	0.8704	0.0076	32.9388	0.9088	0.0043	34.8727	0.9457	0.0157	30.3354	0.8532	0.0090	32.3368	0.8991	0.0092	32.2647	0.8978
	128	0.0040	35.3538	0.9422	0.0073	33.1554	0.9103	0.0050	34.5940	0.9328	0.0031	36.2011	0.9579	0.0126	31.1616	0.8685	0.0059	34.0126	0.9256	0.0067	33.4865	0.9173
Smoke	16	0.0005	44.1372	0.9807	0.0009	42.0877	0.9791	0.0007	43.0729	0.9784	0.0005	44.0520	0.9795	0.0006	43.3547	0.9761	0.0006	43.4443	0.9800	0.0006	43.5586	0.9786
	32	0.0004	44.9731	0.9818	0.0007	42.8350	0.9803	0.0006	43.9832	0.9801	0.0004	45.1488	0.9812	0.0005	44.2466	0.9782	0.0005	44.1988	0.9811	0.0004	45.0087	0.9809
	64	0.0004	45.4864	0.9825	0.0005	43.9138	0.9812	0.0005	44.8552	0.9813	0.0004	45.9017	0.9822	0.0004	45.0737	0.9799	0.0004	45.0065	0.9819	0.0004	45.8805	0.9822
	128	0.0004	45.8068	0.9829	0.0005	44.4741	0.9817	0.0004	45.4951	0.9821	0.0003	46.3656	0.9827	0.0004	45.6013	0.9808	0.0004	45.6926	0.9825	0.0003	46.3678	0.9828
Country Kitchen Night	16	0.0138	27.8739	0.8140	0.0133	28.5459	0.8743	0.0115	29.8702	0.8368	0.0146	28.1630	0.8335	0.0250	24.2337	0.7679	0.0112	30.4661	0.8835	0.0104	28.8515	0.8689
	32	0.0098	29.2396	0.8340	0.0090	30.2983	0.8920	0.0068	31.8874	0.8766	0.0088	30.1918	0.8689	0.0185	25.3280	0.7845	0.0066	32.4528	0.9055	0.0083	29.3832	0.8827
	64	0.0069	30.7094	0.8546	0.0060	31.8616	0.9046	0.0045	33.4613	0.9028	0.0056	31.9737	0.8964	0.0121	26.9291	0.8139	0.0039	34.1304	0.9220	0.0075	29.6946	0.8841
	128	0.0047	32.4360	0.8775	0.0039	33.4580	0.9179	0.0029	35.1759	0.9226	0.0033	34.1158	0.9192	0.0076	28.8330	0.8464	0.0023	35.9318	0.9367	0.1017	30.4582	0.8927
Glass of Water	16	0.0171	28.7763	0.9273	0.0755	25.0842	0.8977	0.0439	26.5674	0.9106	0.0435	26.1671	0.9067	0.0278	26.5721	0.9128	0.0349	27.0671	0.9206	0.0352	26.2145	0.8984
	32	0.0115	30.5121	0.9457	0.0643	25.6950	0.9065	0.0324	27.3999	0.9271	0.0305	27.0536	0.9167	0.0209	27.8729	0.9265	0.0265	27.6904	0.9287	0.0272	27.2131	0.9124
	64	0.0078	32.2051	0.9586	0.0587	25.9437	0.9128	0.0220	28.5058	0.9404	0.0247	27.6811	0.9266	0.0147	29.0346	0.9415	0.0215	28.4385	0.9381	0.0198	28.2481	0.9264
	128	0.0052	33.8967	0.9691	0.0537	26.1498	0.9168	0.0156	29.7450	0.9536	0.0182	28.5106	0.9394	0.0105	30.0555	0.9537	0.0156	29.2368	0.9480	0.0180	28.6043	0.9313
Averages:	16	0.0155	31.0847	0.8623	0.0383	29.2334	0.8491	0.0309	30.0674	0.8410	0.0196	30.8281	0.8827	0.0241	29.0212	0.8390	0.0171	31.0932	0.8874	0.0256	29.9514	0.8464
	32	0.0100	32.5659	0.8891	0.0263	30.2569	0.8691	0.0171	31.6533	0.8788	0.0140	32.0741	0.9047	0.0167	30.3274	0.8632	0.0127	32.1983	0.9065	0.0188	31.0806	0.8720
	64	0.0067	34.0422	0.9114	0.0189	31.4233	0.8906	0.0106	33.1470	0.9068	0.0106	33.1917	0.9217	0.0120	31.5677	0.8867	0.0096	33.2877	0.9226	0.0141	32.0390	0.8909
	128	0.0045	35.6427	0.9311	0.0143	32.6057	0.9112	0.0071	34.6354	0.9287	0.0078	34.3823	0.9364	0.0087	32.7966	0.9075	0.0073	34.4688	0.9370	0.0264	32.4659	0.9060

6.1 Learning-Based Techniques

Since the FBKSD API includes a Python version, it enables implementing techniques that use popular deep learning frameworks like TensorFlow (ABADI et al., 2016), Keras (CHOLLET et al., 2015), and PyTorch (PASZKE et al., 2017). To demonstrate such case, we adapted the KPCN (BAKO et al., 2017) technique to our Python API. The original technique's source code provided by the authors is written in Python and TensorFlow. We utilized the trained model included in the original source code, therefore, we did not train the neural network specifically for FBKSD. Figure 6.8 shows some results.

One particular characteristic of KPCN that is not present in the other techniques

Figure 6.7: Results from a benchmark including seven MC denoising techniques and nine scenes (from our scene pool) that pose challenges to denoising methods. All results were generated with 128 samples per pixel.

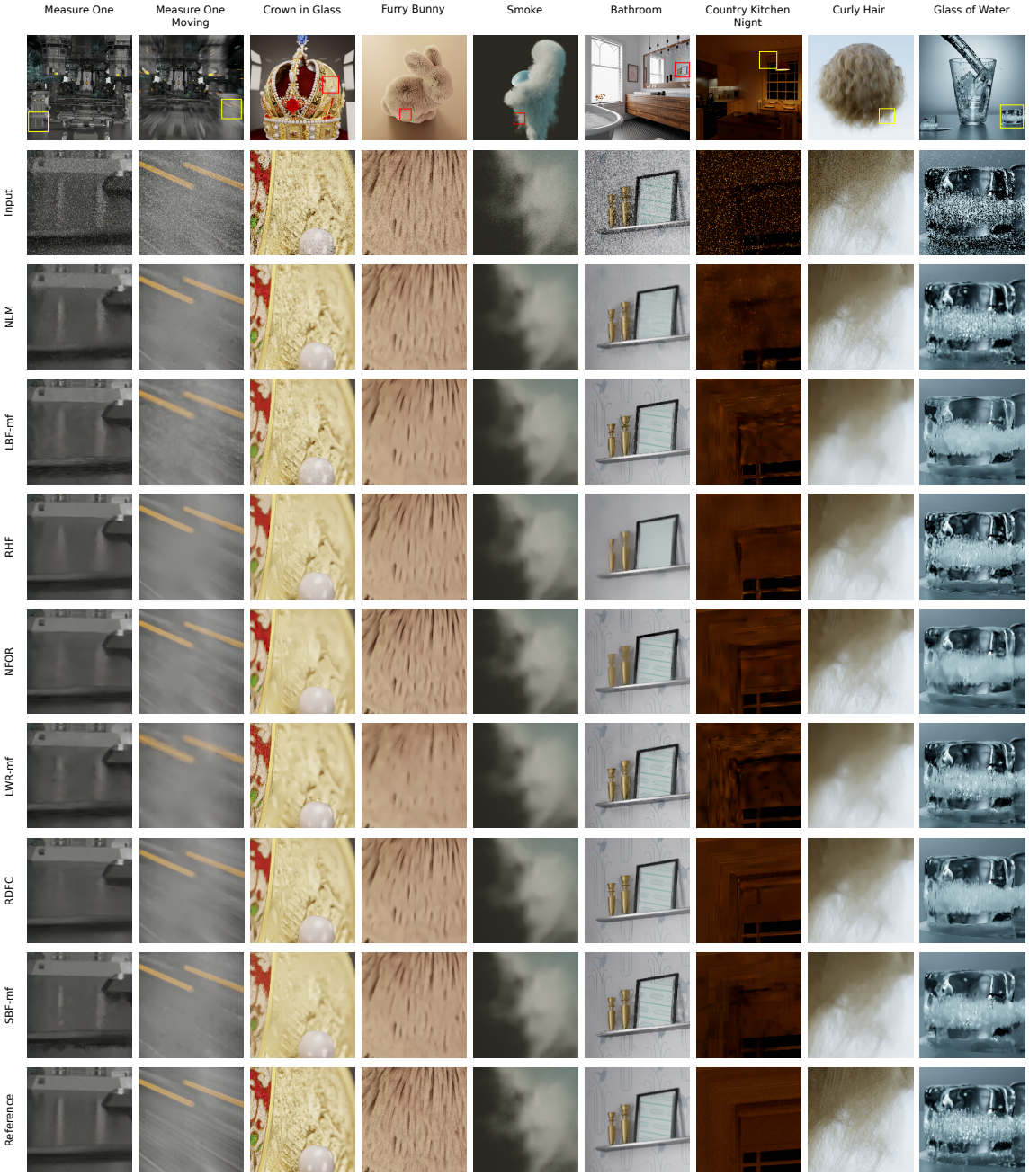
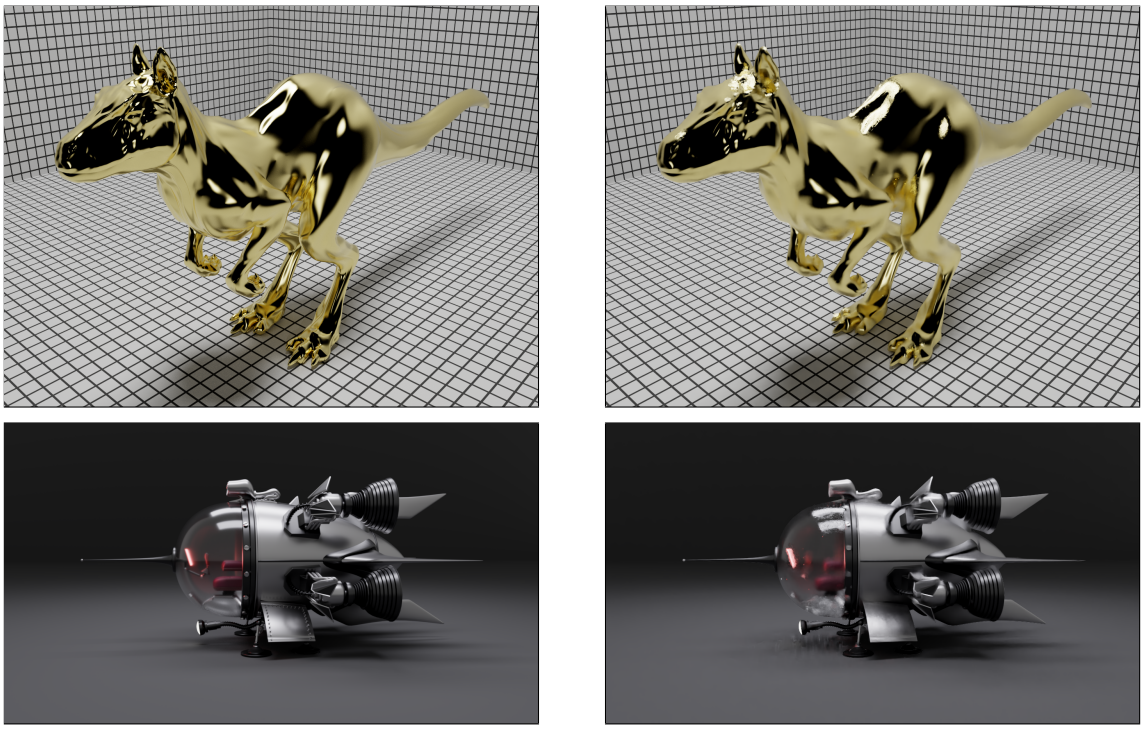


Figure 6.8: Results from the KPCN technique using FBKSD's Python API. The Golden Killeroo (top row) and the Spaceship (bottom row) scenes were rendered with PBRT-v2 and Mitsuba, respectively. Using scenes from different renderers poses a bigger challenge for CNN techniques trained with only one renderer.

(a) Reference (b) KPCN (16spp)



we adapted, is its sample feature requirements. This technique uses a diffuse/specular decomposition (ZIMMER et al., 2015) approach, which separates the input image into its diffuse and specular components, so they can be processed separately. FBKSD supports this decomposition by providing a diffuse component sample feature, which can then be subtracted from the full color feature to obtain the specular component.

Although our framework was not specifically designed for training learning-based techniques, it can be used for this purpose by saving the sample values as multilayer HDR images (e.g., OpenEXR (OPENEXR, 2019)) and using the images for training purposes. Using FBKSD in this manner offers the advantage of allowing scenes from different rendering systems to be included in the training set, which can make the resulting technique more robust to differences in dynamic range and noise signatures resulting from different renderers. Another approach for increasing the training set is to convert existing scenes from one renderer to the others (HAGEMANN; OLIVEIRA, 2018), which would better isolate the differences between different renderers.

6.2 Visualization Interface

We developed a web-based graphical user interface (GUI) to allow the visualization the results generated by the system. The GUI was implemented using the Angular JavaScript framework (ANGULAR, 2019). For now, the GUI allows qualitative and quantitative quality assessments. The interface allows visual comparisons, visualization of error maps, and charts for all the included image quality assessment (IQA) metrics (MSE, rMSE, PSNR, and SSIM). Figure 6.9 shows visual comparison page, which allows interactive visual inspection of the images. It displays the current selected image in the center, and the miniatures for all the techniques at the bottom. The miniatures show a zoomed in portion of the image centered at the cursor. As the user moves the cursor, the miniatures change accordingly. The top bar contains the controls for changing the current scene, image (reference image or the result image for one of the techniques), color buffer or IQA error map, and the spp value for the current technique.

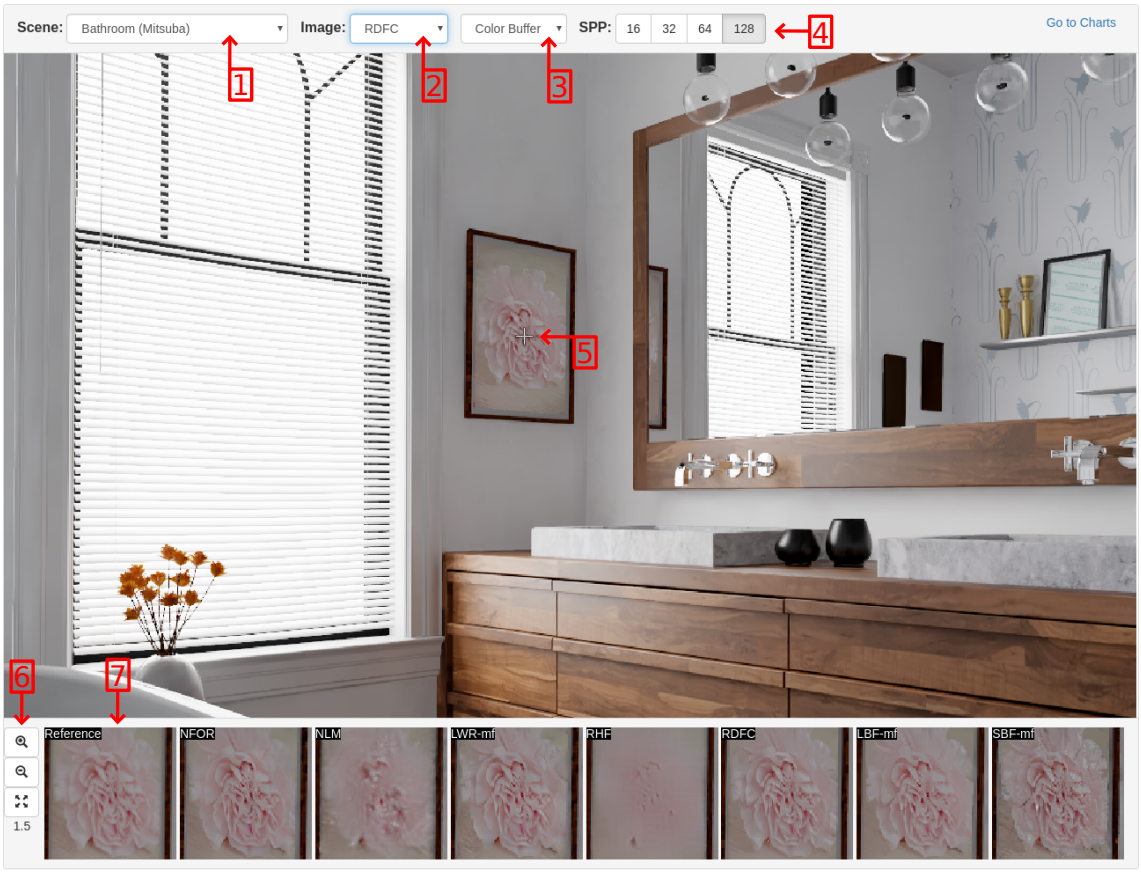
For the quantitative assessments, charts showing execution time and error values generated by the IQA metrics are included in the GUI (Figure 6.10). The charts support common user interactions: zooming, displaying the numeric values by hovering the cursor, and showing/hiding lines. The charts can also be exported in several bitmap (PNG and JPEG) and vector (SVG and PDF) formats suitable for publication.

6.3 Discussion

6.3.1 Communication Overhead

The communication overhead can be significant depending on how a technique requests samples. If the entire budget is requested in a single call, the overhead is negligible. It increases with the number of calls. If, for instance, each call requests a single sample, as in the case of MDAS (HACHISUKA et al., 2008), the overhead becomes prohibitive for anything but a very small number of samples and image sizes.

Figure 6.9: Image comparison panel included in the results visualization GUI. The top bar contains several controls for selecting the current scene (1), image (2), buffer (3), and spp value for the current technique (4). The central area shows the currently selected image, and allows zooming and panning. At to bottom, the miniatures for all the techniques are shown (7). The miniatures are zoomed in portions of the corresponding images centered at the cursor position (5). The size of the miniatures can be adjusted (6).

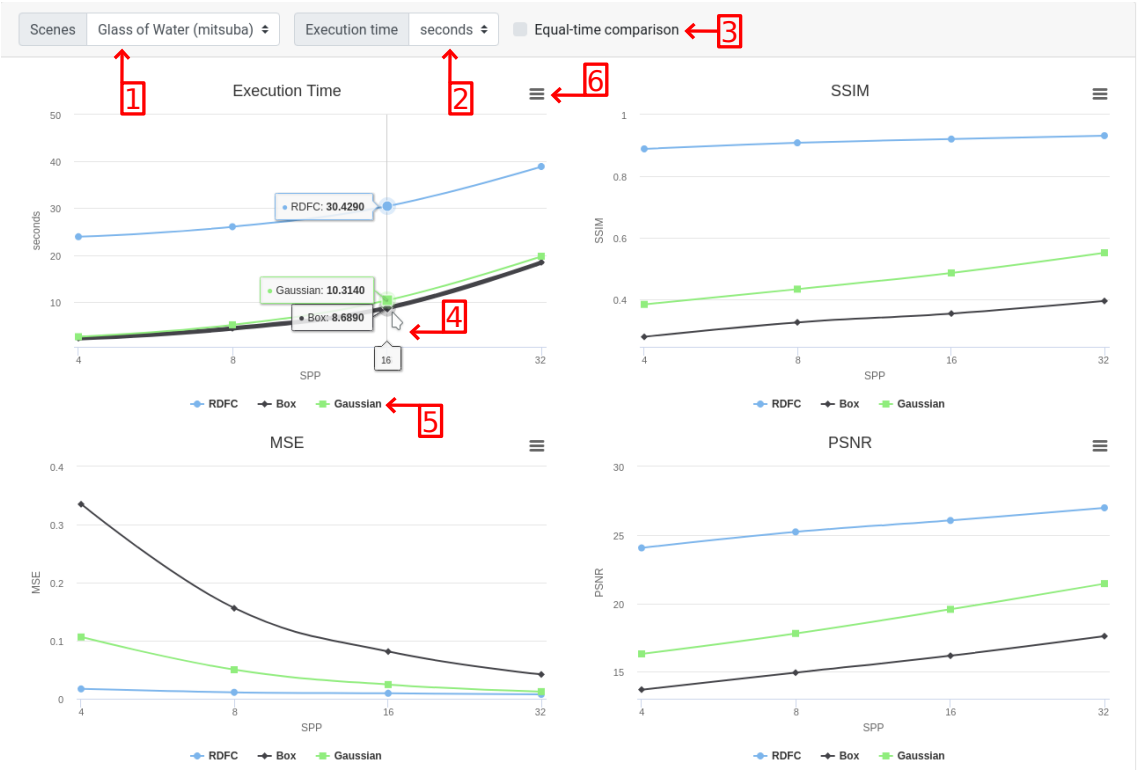


6.3.2 Memory Overhead

Our system utilizes two main strategies to minimize memory overhead: use a shared buffer to transfer the sample data between renderer and technique; and avoid allocating a full-sized buffer by using a tile-based transfer approach. However, the current implementation of some techniques do not leverage this design, and try to store all sample data in another buffer allocated locally. This is the case of the RPF (SEN; DARABI, 2012) technique, for example.

For such techniques, the memory overhead can be significant. For example, rendering a full HD image (1920×1080) with 256 samples per pixel, and assuming the technique configures a sample layout with 14 components, the total amount of main memory required to store all sample data at once in a single buffer would be around 27.7 GB.

Figure 6.10: Charts panel included in results visualization GUI. The top bar contains controls for selecting the current scene (1), changing the execution time unit to seconds or minutes (2), and showing equal-time comparison charts (3). Each individual chart also supports user interaction: showing numeric values by hovering the cursor (4), showing/hiding individual lines by clicking on the corresponding label (5), and exporting the chart in several bitmap and vector formats by opening the pop-up menu (6).



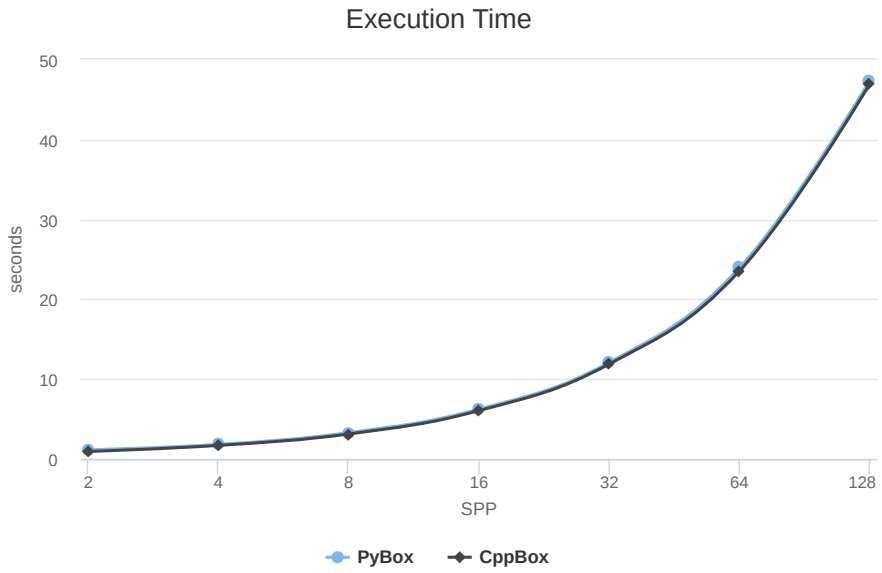
6.3.3 Python API Overhead

When developing a technique using our Python API, a performance loss is naturally expected when compared to a C++ implementation. Our Python API is just a thin wrapper (binding) over the native C++ one, so the loss in performance will come mainly from the technique's Python code itself, not from the API calls (inherent overhead).

To evaluate the inherent overhead, we compared the execution times of a simple box filter implemented in C++ vs one implemented in Python (Figure 6.11). The execution time overhead is negligible, and as expected, there is no difference in image quality.

As a technique becomes more complex, the performance hit is expected to become more noticeable, since more Python code is executed. A good rule-of-thumb to keep in mind when developing a technique in Python is to avoid raw loops and use *numpy* functions whenever possible. If a *numpy* function is not available for the desired purpose, and the overhead of a custom Python implementation is too high, it is possible to implement the

Figure 6.11: Execution time in seconds of a simple box filter technique written using our C++ API (CppBox) vs the Python version (PyBox). The overhead is negligible.



function in C/C++ and call it from Python using Python's *ctypes* mechanism.

6.4 Summary

This chapter presented a case study to demonstrate the use of our framework. We provided results from a good variety of denoising techniques for Monte Carlo Rendering, which we adapted to our framework, including a recent CNN-based technique (Section 6.1). To serve as rendering back-end for our system, we also adapted four rendering systems. The results visualization interface included with FBKSD was presented in Section 6.2, showing its image comparison tool and charting features. We finished with a discussion on types of overhead that can occur and how to avoid them (Section 6.3).

7 CONCLUSIONS AND FUTURE WORK

This thesis presented the FBKSD system — A Framework for Developing and Benchmarking Sampling and Denoising Algorithms for Monte Carlo Rendering — which allows developers to implement sampling and denoising algorithms in a general, renderer-agnostic way. This makes it straightforward to perform extensive benchmarks involving various algorithms across different renderers. Our system contains both a *local* version consisting of an SDK that allows users to develop and test their techniques in their own machines, and an *online* submission system that allows submitted techniques to be automatically benchmarked and published.

FBKSD initially supports three well known rendering systems (PBRT v3, PBRT v2, Mitsuba), as well as a custom procedural renderer. The system is flexible enough to allow the addition of new renderers in the future. Having a good selection of renderers increases the availability of scenes and also poses a more challenging test for techniques, specially learning-based ones, which are usually trained using only one rendering system.

To demonstrate the capabilities of our framework, we adapted several denoising techniques to our API (LWR (REN et al., 2013), NFOR (BITTERLI et al., 2016), LBF (KALANTARI; BAKO; SEN, 2015), RPF (SEN; DARABI, 2012), SBF (LI; WU; CHUANG, 2012), RHF (DELBRACIO et al., 2014), NLM (ROUSSELLE; KNAUS; ZWICKER, 2012), RDFC (ROUSSELLE; MANZI; ZWICKER, 2013), GEM (ROUSSELLE; KNAUS; ZWICKER, 2011), and KPCN (BAKO et al., 2017)), and presented a case study with an evaluation of the results produced. This significant set of techniques form a good representation of the techniques in the field, which demonstrates the versatility of our framework.

In combination with the online submission system, FBKSD greatly simplifies the task of developing new techniques and comparing existing ones, which is an invaluable addition to the rendering research community.

7.1 Future Work

The current result visualization page provided with our system has some limitations. The main one is that it does not provide a ranking of the techniques. This could be implemented in the future by ordering the techniques by error and assigning a score based on the relative position of each technique for each scene.

Another improvement to the results page would be to add a classification system for the scenes. The basic idea is to assign tags for each scene, denoting which characteristics each scene contain. Scene characteristics can include, for example, the presence of certain effects (e.g., motion blur, depth-of-field, caustics). This extra information would allow a better classification of the benchmarked techniques according the their performance for each effect. Implementing this feature requires changing both the FBKSD core and the visualization page. The core needs to read the tags from the scene metadata files and save them to the output files used to feed the visualization page. The visualization page needs to read this new data and implement a convenient way of searching and classifying the existing results according to the tags.

REFERENCES

ABADI, M. et al. Tensorflow: A system for large-scale machine learning. In: **Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2016. (OSDI'16), p. 265–283. ISBN 978-1-931971-33-1. Available from Internet: <http://dl.acm.org/citation.cfm?id=3026877.3026899>.

ANDERSON, L. et al. A Domain-Specific Language for Monte Carlo Sampling. **ACM Trans. Graph.**, v. 36, n. 4, jul 2017.

ANGULAR. 2019. Available from Internet: <https://angular.io>.

APPEL, A. Some techniques for shading machine renderings of solids. In: **Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference**. New York, NY, USA: ACM, 1968. (AFIPS '68 (Spring)), p. 37–45. Available from Internet: <http://doi.acm.org/10.1145/1468075.1468082>.

ARVO, J. The role of functional analysis in global illumination. In: **Rendering Techniques '95**. [S.l.]: Springer-Verlag, 1995. p. 115–126.

ARVO, J.; KIRK, D. Particle transport and image synthesis. In: **Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1990. (SIGGRAPH '90), p. 63–66. ISBN 0-89791-344-2. Available from Internet: <http://doi.acm.org/10.1145/97879.97886>.

BAKER, S. et al. **Middlebury Flow Accuracy and Interpolation Evaluation**. 2011. Available from Internet: <http://vision.middlebury.edu/flow/eval/>.

BAKER, S. et al. A database and evaluation methodology for optical flow. **IJCV**, Kluwer Academic Publishers, Hingham, MA, USA, v. 92, n. 1, p. 1–31, mar. 2011. ISSN 0920-5691.

BAKO, S. et al. Kernel-predicting convolutional networks for denoising monte carlo renderings. **ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2017)**, v. 36, n. 4, July 2017.

BARRON, J. L.; FLEET, D. J.; BEAUCHEMIN, S. S. Performance of optical flow techniques. **IJCV**, Kluwer Academic Publishers, Hingham, MA, USA, v. 12, n. 1, p. 43–77, feb. 1994. ISSN 0920-5691.

BAUSZAT, P. et al. General and robust error estimation and reconstruction for monte carlo rendering. **Computer Graphics Forum**, v. 34, n. 2, p. 597–608, 2015. ISSN 1467-8659.

BITBUCKET. 2019. Available from Internet: <https://bitbucket.org>.

BITTERLI, B. et al. Nonlinearly Weighted First-order Regression for Denoising Monte Carlo Renderings. **Computer Graphics Forum**, v. 35, n. 4, p. 107–117, jul 2016. ISSN 01677055.

BUCK, I. et al. Brook for gpus: Stream computing on graphics hardware. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 23, n. 3, p. 777–786, aug. 2004. ISSN 0730-0301.

CHAITANYA, C. R. A. et al. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 36, n. 4, p. 98:1–98:12, jul. 2017. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/3072959.3073601>.

CHIU, K.; SHIRLEY, P.; WANG, C. Graphics gems iv. In: HECKBERT, P. S. (Ed.). San Diego, CA, USA: Academic Press Professional, Inc., 1994. chp. Multi-jittered Sampling, p. 370–374. ISBN 0-12-336155-9. Available from Internet: <http://dl.acm.org/citation.cfm?id=180895.180927>.

CHOLLET, F. et al. **Keras**. 2015. <https://keras.io>.

COOK, R. L. Stochastic sampling in computer graphics. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 5, n. 1, p. 51–72, jan. 1986. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/7529.8927>.

COOK, R. L.; PORTER, T.; CARPENTER, L. Distributed ray tracing. In: **Proc. SIGGRAPH '84**. [S.l.: s.n.], 1984. p. 137–145. ISSN 0097-8930.

CROW, F. C. The aliasing problem in computer-generated shaded images. **Commun. ACM**, ACM, New York, NY, USA, v. 20, n. 11, p. 799–805, nov. 1977. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/359863.359869>.

DELBRACIO, M. et al. Boosting monte carlo rendering by ray histogram fusion. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 33, n. 1, p. 8:1–8:15, feb. 2014. ISSN 0730-0301.

DIPPÉ, M. A. Z.; WOLD, E. H. Antialiasing through stochastic sampling. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 19, n. 3, p. 69–78, jul. 1985. ISSN 0097-8930. Available from Internet: <http://doi.acm.org/10.1145/325165.325182>.

DOBKIN, D. P.; EPPSTEIN, D.; MITCHELL, D. P. Computing the discrepancy with applications to supersampling patterns. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 15, n. 4, p. 354–376, oct. 1996. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/234535.234536>.

DOBKIN, D. P.; MITCHELL, D. P. Random-edge discrepancy of supersampling patterns. In: **Graphics Interface '93**. [S.l.: s.n.], 1993. p. 62–69.

DOCKER. 2019. Available from Internet: <https://www.docker.com>.

DUNBAR, D.; HUMPHREYS, G. A spatial data structure for fast poisson-disk sample generation. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 25, n. 3, p. 503–508, jul. 2006. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/1141911.1141915>.

DUTRE, P. et al. **Advanced Global Illumination**. [S.l.]: AK Peters Ltd, 2006. ISBN 1568813074.

EBEIDA, M. S. et al. Efficient maximal poisson-disk sampling. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 30, n. 4, p. 49:1–49:12, jul. 2011. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/2010324.1964944>.

EROFEEV, M. et al. **VideoMatting**. 2014. Available from Internet: <http://videomatting.com/>.

EROFEEV, M. et al. Perceptually motivated benchmark for video matting. In: XIE, X.; JONES, M. W.; TAM, G. K. L. (Ed.). **BMVA Press Proc.** [S.l.: s.n.], 2015. p. 99.1–99.12. ISBN 1-901725-53-7.

FERWERDA, J. A. Three varieties of realism in computer graphics. In: ROGOWITZ, B. E.; PAPPAS, T. N. (Ed.). **Human Vision and Electronic Imaging VIII**. [S.l.]: SPIE, 2003. v. 5007, p. 290 – 297.

GHARBI, M. et al. Sample-based monte carlo denoising using a kernel-splatting network. **ACM Trans. Graph.**, ACM, v. 38, n. 4, p. 125:1–125:12, 2019.

GITHUB. 2019. Available from Internet: <https://github.com>.

GITLAB. 2019. Available from Internet: <https://gitlab.com>.

GOODFELLOW, I. et al. Generative adversarial nets. In: GHAHRAMANI, Z. et al. (Ed.). **Advances in Neural Information Processing Systems**. Curran Associates, Inc., 2014. p. 2672–2680. Available from Internet: <http://papers.nips.cc/paper/5423-generative-adversarial-nets.pdf>.

HACHISUKA, T. et al. Multidimensional adaptive sampling and reconstruction for ray tracing. **ACM TOG**, v. 27, n. 212, p. 1, 2008. ISSN 07300301.

HAGEMANN, L.; OLIVEIRA, M. M. Scene conversion for physically-based renderers. In: **2018 31st SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)**. [S.l.: s.n.], 2018. p. 226–233.

HECK, D.; SCHLöMER, T.; DEUSSEN, O. Blue noise sampling with controlled aliasing. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 32, n. 3, p. 25:1–25:12, jul. 2013. ISSN 0730-0301.

HECKBERT, P. S. Adaptive radiosity textures for bidirectional ray tracing. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 24, n. 4, p. 145–154, sep. 1990. ISSN 0097-8930. Available from Internet: <http://doi.acm.org/10.1145/97880.97895>.

JAKOB, W. **Mitsuba renderer**. 2010. Http://www.mitsuba-renderer.org.

JAROSZ, W. et al. Orthogonal array sampling for monte carlo rendering. **Comput. Graph. Forum**, v. 38, p. 135–147, 2019.

JONES, T. R. Efficient generation of poisson-disk sampling patterns. **Journal of Graphics Tools**, Taylor & Francis, v. 11, n. 2, p. 27–36, 2006. Available from Internet: <https://doi.org/10.1080/2151237X.2006.10129217>.

KAILKHURA, B. et al. Stair blue noise sampling. **ACM Trans. Graph.**, ACM, v. 35, n. 6, p. 248:1–248:10, nov. 2016. ISSN 0730-0301.

KAJIYA, J. T. The rendering equation. **SIGGRAPH'86**, ACM, New York, NY, USA, v. 20, n. 4, p. 143–150, aug. 1986. ISSN 0097-8930.

KALANTARI, N. K.; BAKO, S.; SEN, P. A machine learning approach for filtering Monte Carlo noise. **ACM Trans. Graph.**, v. 34, n. 4, p. 122:1–122:12, jul 2015. ISSN 07300301.

KALANTARI, N. K.; SEN, P. Fast generation of approximate blue noise point sets. **Comput. Graph. Forum**, The Eurographs Association & John Wiley & Sons, Ltd., Chichester, UK, v. 31, n. 4, p. 1529–1535, jun. 2012. ISSN 0167-7055. Available from Internet: <http://dx.doi.org/10.1111/j.1467-8659.2012.03149.x>.

KALANTARI, N. K.; SEN, P. Removing the noise in monte carlo rendering with general image denoising algorithms. **Computer Graphics Forum**, Blackwell Publishing Ltd, v. 32, n. 2pt1, p. 93–102, 2013. ISSN 1467-8659.

KENSLER, A. Correlated multi-jittered sampling. In: **Pixar Technical Memo 13-01, Pixar Animation Studios**. [S.l.: s.n.], 2013.

KETTUNEN, M.; HäRKöNEN, E.; LEHTINEN, J. Deep convolutional reconstruction for gradient-domain rendering. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 38, n. 4, p. 126:1–126:12, jul. 2019. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/3306346.3323038>.

KOPF, J. et al. Recursive wang tiles for real-time blue noise. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 25, n. 3, p. 509–518, jul. 2006. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/1141911.1141916>.

KUZNETSOV, A.; KALANTARI, N. K.; RAMAMOORTHI, R. Deep adaptive sampling for low sample count rendering. **Computer Graphics Forum**, v. 37, n. 4, p. 35–44, 2018.

LAFORTUNE, E. P.; WILLEMS, Y. D. Bi-Directional Path Tracing. In: **Proceedings of the 3rd international conference on Computational graphics and Visualization techniques (COMPUGRAPHICS '93**. [S.l.: s.n.], 1993. p. 145–153.

LAGAE, A.; DUTRÉ, P. An alternative for wang tiles: Colored edges versus colored corners. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 25, n. 4, p. 1442–1459, oct. 2006. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/1183287.1183296>.

LAGAE, A.; DUTRé, P. A comparison of methods for generating poisson disk distributions. **Computer Graphics Forum**, Blackwell Publishing Ltd, v. 27, n. 1, p. 114–129, 2008. ISSN 1467-8659. Available from Internet: <http://dx.doi.org/10.1111/j.1467-8659.2007.01100.x>.

LEE, M. E.; REDNER, R. A. Filtering: A note on the use of nonlinear filtering in computer graphics. **IEEE Comput. Graph. Appl.**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 10, n. 3, p. 23–29, may 1990. ISSN 0272-1716.

LI, T.-M.; WU, Y.-t.; CHUANG, Y.-y. SURE-based optimization for adaptive sampling and reconstruction. **ACM Trans. Graph.**, v. 31, p. 1, 2012. ISSN 07300301.

MARK, W. R. et al. Cg: A system for programming graphics hardware in a c-like language. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 22, n. 3, p. 896–907, jul. 2003. ISSN 0730-0301.

MITCHELL, D. P. Spectrally optimal sampling for distribution ray tracing. **SIGGRAPH Comput. Graph.**, ACM, New York, NY, USA, v. 25, n. 4, p. 157–164, jul. 1991. ISSN 0097-8930. Available from Internet: <http://doi.acm.org/10.1145/127719.122736>.

MITCHELL, D. P. Ray tracing and irregularities of distribution. In: **In Third Eurographics Workshop on Rendering**. [S.l.: s.n.], 1992. p. 61–69.

MITCHELL, D. P. Consequences of stratified sampling in graphics. In: **Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques**. New York, NY, USA: ACM, 1996. (SIGGRAPH '96), p. 277–280. ISBN 0-89791-746-4. Available from Internet: <http://doi.acm.org/10.1145/237170.237265>.

MITCHELL, S. A. et al. Spoke-darts for high-dimensional blue-noise sampling. **ACM Trans. Graph.**, ACM, v. 37, n. 2, p. 22:1–22:20, may 2018. ISSN 0730-0301.

MOON, B.; CARR, N.; YOON, S.-E. Adaptive rendering based on weighted local regression. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 33, n. 5, p. 170:1–170:14, sep. 2014. ISSN 0730-0301.

MULLAPUDI, R. T. et al. Automatically scheduling halide image processing pipelines. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 35, n. 4, p. 83:1–83:11, jul. 2016. ISSN 0730-0301.

NIEDERREITER, H. **Random Number Generation and quasi-Monte Carlo Methods**. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 1992. ISBN 0-89871-295-5.

OPENEXR. 2019. Available from Internet: <https://www.openexr.com>.

PASZKE, A. et al. Automatic differentiation in PyTorch. In: **NIPS Autodiff Workshop**. [S.l.: s.n.], 2017.

PERRIER, H. et al. Sequences with low-discrepancy blue-noise 2-d projections. **Comput. Graph. Forum**, v. 37, p. 339–353, 2018.

PHARR, M.; HUMPHREYS, G. **Physically Based Rendering, from Theory to Implementation**. 2. ed. [S.l.]: Morgan Kaufmann, 2010.

PHARR, M.; HUMPHREYS, G. **Physically Based Rendering, Second Edition: From Theory To Implementation**. 2nd. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN 0123750792, 9780123750792.

PHARR, M.; JAKOB, W.; HUMPHREYS, G. **Physically Based Rendering, from Theory to Implementation**. 3. ed. [S.l.]: Morgan Kaufmann, 2016.

RAGAN-KELLEY, J. et al. Decoupling algorithms from schedules for easy optimization of image processing pipelines. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 31, n. 4, p. 32:1–32:12, jul. 2012. ISSN 0730-0301.

REINERT, B. et al. Projective blue-noise sampling. **Computer Graphics Forum**, 2015. ISSN 1467-8659.

REN, P. et al. Global illumination with radiance regression functions. **ACM Trans. Graph.**, v. 32, p. 1, 2013. ISSN 07300301.

RHEMANN, C. et al. **Alpha Matting Evaluation Website**. 2009. Available from Internet: <http://www.alphamatting.com/eval_25.php>.

RHEMANN, C. et al. A perceptually motivated online benchmark for image matting. In: **CVPR**. [S.l.: s.n.], 2009. p. 1826–1833. ISBN 978-1-4244-3992-8.

ROUSSELLE, F.; KNAUS, C.; ZWICKER, M. Adaptive sampling and reconstruction using greedy error minimization. **ACM Trans. Graph.**, v. 30, n. 6, dec. 2011.

ROUSSELLE, F.; KNAUS, C.; ZWICKER, M. Adaptive rendering with non-local means filtering. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 31, n. 6, p. 195:1–195:11, nov. 2012. ISSN 0730-0301.

ROUSSELLE, F.; MANZI, M.; ZWICKER, M. Robust denoising using feature and color information. **Comput. Graph. Forum**, v. 32, n. 7, p. 121–130, 2013.

RUSHMEIER, H. E.; WARD, G. J. Energy preserving non-linear filters. In: **Proc. SIGGRAPH '94**. [S.l.: s.n.], 1994. p. 131–138. ISBN 0-89791-667-0.

RUST. 2019. Available from Internet: <https://www.rust-lang.org>.

SAMET, H. Sorting in space: Multidimensional, spatial, and metric data structures for computer graphics applications. In: **ACM SIGGRAPH ASIA 2010 Courses**. [S.l.: s.n.], 2010. (SA '10), p. 3:1–3:52. ISBN 978-1-4503-0527-3.

SCHARSTEIN, D.; SZELISKI, R. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. **IJCV**, Kluwer Academic Publishers, Hingham, MA, USA, v. 47, n. 1-3, p. 7–42, abr. 2002. ISSN 0920-5691.

SCHARSTEIN, D.; SZELISKI, R.; HIRSCHMüLLER, H. **Middlebury Stereo Vision Page**. 2002. Available from Internet: <http://vision.middlebury.edu/stereo/>.

SEN, P.; DARABI, S. On filtering the noise from the random parameters in Monte Carlo rendering. **ACM Trans. Graph.**, v. 31, n. 3, p. 1–15, may 2012. ISSN 07300301.

SHIRLEY, P. Discrepancy as a quality measure for sample distributions. In: **In Eurographics '91**. [S.l.]: Elsevier Science Publishers, 1991. p. 183–194.

SOBOL', I. On the distribution of points in a cube and the approximate evaluation of integrals. **USSR Computational Mathematics and Mathematical Physics**, v. 7, n. 4, p. 86 – 112, 1967. ISSN 0041-5553. Available from Internet: <http://www.sciencedirect.com/science/article/pii/0041555367901449>.

STEIN, C. M. Estimation of the mean of a multivariate normal distribution. **Ann. Statist.**, The Institute of Mathematical Statistics, v. 9, n. 6, p. 1135–1151, 11 1981.

VEACH, E. **Robust monte carlo methods for light transport simulation**. Thesis (PhD), Stanford, CA, USA, 1998. AAI9837162.

VEACH, E.; GUIBAS, L. J. Metropolis light transport. In: **Proc. SIGGRAPH '97**. [S.l.: s.n.], 1997. p. 65–76. ISBN 0-89791-896-7.

VOGELS, T. et al. Denoising with kernel prediction and asymmetric loss functions. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 37, n. 4, p. 124:1–124:15, jul. 2018. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/3197517.3201388>.

WEI, L.-Y. Parallel poisson disk sampling. **ACM Trans. Graph.**, ACM, New York, NY, USA, v. 27, n. 3, p. 20:1–20:9, aug. 2008. ISSN 0730-0301. Available from Internet: <http://doi.acm.org/10.1145/1360612.1360619>.

WEI, L.-Y.; WANG, R. Differential domain analysis for non-uniform sampling. **ACM Trans. Graph.**, v. 30, p. 50, 2011.

WHITTED, T. An improved illumination model for shaded display. **Commun. ACM**, ACM, New York, NY, USA, v. 23, n. 6, p. 343–349, jun. 1980. ISSN 0001-0782. Available from Internet: <http://doi.acm.org/10.1145/358876.358882>.

XU, B. et al. Adversarial monte carlo denoising with conditioned auxiliary feature. **ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH Asia 2019)**, v. 38, n. 6, p. 224:1–224:12, 2019.

YELLOTT, J. Spectral consequences of photoreceptor sampling in the rhesus retina. **Science**, American Association for the Advancement of Science, v. 221, n. 4608, p. 382–385, 1983. ISSN 0036-8075. Available from Internet: <http://science.sciencemag.org/content/221/4608/382>.

YUKSEL, C. Sample elimination for generating poisson disk sample sets. **Comput. Graph. Forum**, v. 34, n. 2, p. 25–32, may 2015. ISSN 0167-7055.

Zhou Wang et al. Image quality assessment: from error visibility to structural similarity. **IEEE Transactions on Image Processing**, v. 13, n. 4, p. 600–612, April 2004.

ZIMMER, L. H. et al. Path-space motion estimation and decomposition for robust animation filtering. **Comput. Graph. Forum**, v. 34, p. 131–142, 2015.

ZWICKER, M. et al. Recent Advances in Adaptive Sampling and Reconstruction for Monte Carlo Rendering. **Computer Graphics Forum**, v. 34, n. 2, p. 667–681, may 2015. ISSN 01677055.

APPENDIX A — FBKSD C++ API REFERENCE

In this chapter, we provide FBKSD C++ API. To access the complete documentation, visit https://fbksd.github.io/fbksd/docs/latest. The main repository for the FBKSD SDK is available at https://github.com/fbksd/fbksd.

A.1 Denoising Technique Example

For developing denoisers or samplers, see the `BenchmarkClient` class. It contains the main API used to communicate with the benchmark server. The code below shows a simple box filter implemented using the client API.

```
#include <fbksd/client/BenchmarkClient.h>
using namespace fbksd;
int main(int argc, char* argv[])
{
    // STEP 1: Instantiate the client object and get scene information.
    BenchmarkClient client(argc, argv);
    SceneInfo scene = client.getSceneInfo();
    const auto w = scene.get<int64_t>("width");
    const auto h = scene.get<int64_t>("height");
    const auto spp = scene.get<int64_t>("max_spp");
    // STEP 2: Set the sample layout.
    SampleLayout layout;
    layout("COLOR_R")("COLOR_G")("COLOR_B");
    client.setSampleLayout(layout);
    float* result = client.getResultBuffer();
    // STEP 3: Request samples.
    client.evaluateSamples(SPP(spp), [&](const BufferTile& tile)
    {
        for(auto y = tile.beginY(); y < tile.endY(); ++y)
        for(auto x = tile.beginX(); x < tile.endX(); ++x)
        {
            float* pixel = &result[y*w*3 + x*3];
            for(int s = 0; s < spp; ++s)
            {
                float* sample = tile(x, y, s);
                pixel[0] += sample[0];
                pixel[1] += sample[1];
                pixel[2] += sample[2];
            }
        }
    });
    // STEP 4: Reconstruct the image.
    const float sppInv = 1.f / (float)spp;
    for(int64_t y = 0; y < h; ++y)
    for(int64_t x = 0; x < w; ++x)
    {
        float* pixel = &result[y*w*3 + x*3];
        pixel[0] *= sppInv;
        pixel[1] *= sppInv;
        pixel[2] *= sppInv;
    }
    // STEP 5: Send result and finish.
    client.sendResult();
    return 0;
}
```

A.2 Sampling Technique Example

Samplers are also implemented using `BenchmarkClient` class, the differences is that a sampler needs to write random parameters for each sample. To accomplish this, the method `BenchmarkClient::evaluateInputSamples()` is used. It receives two callback functions, where the first callback is used to generate the sample positions, and the second one is used to consume the sample values (as in the denoiser example).

The example below shows an example of a sampling technique using our API. Note that besides the RGB color, the sample layout contains the random parameters `IMAGE_X`, `IMAGE_Y`, etc., which are all set as `INPUT`. These are the random parameters the sampler needs to generate.

```
1  #include <fbksd/client/BenchmarkClient.h>
2  using namespace fbksd;
3
4  // Function this sampler uses to generate random sample positions.
5  float randomFloat();
6
7  int main(int argc, char* argv[])
8  {
9      // STEP 1: Instantiate the client object and get scene information.
10     BenchmarkClient client(argc, argv);
11     SceneInfo scene = client.getSceneInfo();
12     const auto w = scene.get<int64_t>("width");
13     const auto h = scene.get<int64_t>("height");
14     const auto spp = scene.get<int64_t>("max_spp");
15     // STEP 2: Set the sample layout.
16     SampleLayout layout;
17     layout("IMAGE_X", SampleLayout::INPUT)
18           ("IMAGE_X", SampleLayout::INPUT)
19           ("LENS_U", SampleLayout::INPUT)
20           ("LENS_V", SampleLayout::INPUT)
21           ("TIME", SampleLayout::INPUT)
22           ("LIGHT_X", SampleLayout::INPUT)
23           ("LIGHT_Y", SampleLayout::INPUT)
24           ("COLOR_R")
25           ("COLOR_G")
26           ("COLOR_B");
27     client.setSampleLayout(layout);
28     float* result = client.getResultBuffer();
29     // STEP 3: Request samples.
30     client.evaluateSamples(SPP(spp),
31         [&](const BufferTile& tile)
32         {
33             // Generate sample positions
34             for(int64_t y = tile.beginY(); y < tile.endY(); ++y)
35             for(int64_t x = tile.beginX(); x < tile.endX(); ++x)
36             for(int s = 0; s < spp; ++s)
37             {
38                 float* sample = tile(x, y, s);
39                 sample[IMAGE_X] = randomFloat() + x;
40                 sample[IMAGE_Y] = randomFloat() + y;
41                 sample[LENS_U] = randomFloat();
42                 sample[LENS_V] = randomFloat();
43                 sample[TIME] = randomFloat();
44                 sample[LIGHT_X] = randomFloat();
45                 sample[LIGHT_Y] = randomFloat();
46             }
47         },
48         [&](const BufferTile& tile)
```

```
49          {
50              // Consume sample values
51              for(auto y = tile.beginY(); y < tile.endY(); ++y)
52              for(auto x = tile.beginX(); x < tile.endX(); ++x)
53              {
54                  float* pixel = &result[y*w*3 + x*3];
55                  for(int s = 0; s < spp; ++s)
56                  {
57                      float* sample = tile(x, y, s);
58                      pixel[0] += sample[0];
59                      pixel[1] += sample[1];
60                      pixel[2] += sample[2];
61                  }
62              }
63          }
64      );
65      // STEP 4: Reconstruct the image.
66      const float sppInv = 1.f / (float)spp;
67      for(int64_t y = 0; y < h; ++y)
68      for(int64_t x = 0; x < w; ++x)
69      {
70          float* pixel = &result[y*w*3 + x*3];
71          pixel[0] *= sppInv;
72          pixel[1] *= sppInv;
73          pixel[2] *= sppInv;
74      }
75      // STEP 5: Send result and finish.
76      client.sendResult();
77      return 0;
78  }
```

A.3 IQA Metric Example

FBKSD provides an API that allows implementing IQA metrics to be used by the system. The library is called `fbksd-iqa`, and is composed of two classes: `IQA` and `Img`. The code bellow is an implementation of MSE IQA metric using OpenCV.

```
1   #include <fbksd/iqa/iqa.h>
2   #include <opencv2/core/core.hpp>
3   #include <opencv2/highgui/highgui.hpp>
4   #include <opencv2/imgproc/imgproc.hpp>
5   using namespace cv;
6
7   static float computeMSE(const Mat& img1, const Mat& img2)
8   {
9       Mat s1;
10      absdiff(img1, img2, s1);
11      s1 = s1.mul(s1);
12      Scalar s = sum(s1);
13      double sse = s.val[0] + s.val[1] + s.val[2];
14      double mse  = sse / (double)(img1.channels() * img1.total());
15      return mse;
16  }
17
18  int main(int argc, char* argv[])
19  {
20      // STEP 1: Create an instance of fbksd::IQA, passing the required information.
21      fbksd::IQA iqa(argc, argv,                              // argc, argv from main()
22                     "My-MSE",                                // Metric's acronym
23                     "My Simple Mean squared error metric",   // Metric's full name
24                     "",                                      // Reference (optional)
25                     true,                                    // lowerIsBetter flag
26                     false);                                  // hasErrorMap flag
27      fbksd::Img ref;
```

```
28      fbksd::Img test;
29      // STEP 2: Load the input images.
30      iqa.loadInputImages(ref, test);
31      // STEP 3: Compute the IQA value comparing the two images.
32      Mat refMat(ref.height(), ref.width(), CV_32FC3, ref.data());
33      Mat testMat(ref.height(), ref.width(), CV_32FC3, test.data());
34      float error = computeMSE(refMat, testMat);
35      // STEP 4: Report the value (and the error map, if supported).
36      iqa.report(error);
37      return 0;
38  }
```

A.4 Renderer Example

Porting renderers to be used as rendering back-ends is a bit more complex. The main classes of interest are: `RenderingServer`, `SamplesPipe` and `SampleBuffer`.

The code below shows an example of simple renderer that always produces blue images. For examples of real renderers, see the renderers provided in the `fbksd-package` git repository.

```
1   #include <fbksd/renderer/RenderingServer.h>
2   #include <fbksd/renderer/samples.h>
3   #include <iostream>
4   #include <thread>
5   #include <cmath>
6   using namespace fbksd;
7   namespace
8   {
9   // Fixed image size and SPP.
10  // In a real renderer, this would be loaded from the scene file.
11  const int64_t g_width = 1000;
12  const int64_t g_height = 1000;
13  const int64_t g_spp = 1;
14  // Returns a SceneInfo with information about the scene.
15  SceneInfo getSceneInfo()
16  {
17      SceneInfo info;
18      info.set<int64_t>("width", g_width);
19      info.set<int64_t>("height", g_height);
20      info.set<int64_t>("max_spp", g_spp);
21      info.set<int64_t>("max_samples", g_width * g_height * g_spp);
22      return info;
23  }
24  void setLayout(const SampleLayout& layout)
25  {
26  }
27  void evaluateSamples(int64_t spp, int64_t remainder, int tileSize)
28  {
29      if(spp)
30      {
31          // break the image in tiles 16x16
32          int nTilesX = std::ceil(float(g_width) / 16);
33          int nTilesY = std::ceil(float(g_height) / 16);
34          for(int tileY = 0; tileY < nTilesY; ++tileY)
35          for(int tileX = 0; tileX < nTilesX; ++tileX)
36          {
37              int beginX = tileX*16;
38              int beginY = tileY*16;
39              int endX = std::min((tileX+1)*16, w);
40              int endY = std::min((tileY+1)*16, h);
41              int numPixels = (endX - beginX) * (endY - beginY);
```

```
42              SamplesPipe pipe({beginX, beginY}, {endX, endY}, spp * numPixels);
43              for(int y = beginY; y < endY; ++y)
44              for(int x = beginX; x < endX; ++x)
45              {
46                  pipe.seek(x, y);
47                  for(int64_t s = 0; s < spp; ++s)
48                  {
49                      SampleBuffer sampleBuffer = pipe.getBuffer();
50                      sampleBuffer.set(IMAGE_X, x);
51                      sampleBuffer.set(IMAGE_Y, y);
52                      sampleBuffer.set(COLOR_R, 0.0);
53                      sampleBuffer.set(COLOR_G, 0.0);
54                      sampleBuffer.set(COLOR_B, 1.0);
55                      pipe << sampleBuffer;
56                  }
57              }
58          }
59      }
60      if(remainder)
61      {
62          // divide the remaining samples in tiles of maximum size.
63          int nTiles = std::ceil(float(remainder) / tileSize);
64          for(int tile = 0; tile < nTiles; ++tile)
65          {
66              int numSamples = tile == 0 ? remainder % tileSize : tileSize;
67              SamplesPipe pipe({0, 0}, {w, h}, numSamples);
68              for(int i = 0; i < numSamples; ++i)
69              {
70                  SampleBuffer sampleBuffer = pipe.getBuffer();
71                  int x = drand48() * g_width;
72                  int y = drand48() * g_height;
73                  sampleBuffer.set(IMAGE_X, x);
74                  sampleBuffer.set(IMAGE_Y, y);
75                  sampleBuffer.set(COLOR_R, 0.0);
76                  sampleBuffer.set(COLOR_G, 0.0);
77                  sampleBuffer.set(COLOR_B, 1.0);
78                  pipe << sampleBuffer;
79              }
80          }
81      }
82  }
83  void finish()
84  {
85      std::cout << "Done." << std::endl;
86  }
87  } // namespace
88  int main(int argc, char* argv[])
89  {
90      // STEP 1: Crate rendering server object.
91      RenderingServer server;
92      // STEP 2: Set the callback functions or methods
93      // In our case, we'll break images in 16x16 tiles.
94      server.onGetTileSize([](){return 16;});
95      // getSceneInfo() will be called when the server need information
96      // about the scene being rendered.
97      server.onGetSceneInfo(&getSceneInfo);
98      // onSetParameters() will be called to inform the renderer about sample layout requirements.
99      server.onSetParameters(&setLayout);
100     // evaluateSamples() will be called each time the server requires samples.
101     // The sample data transfer between client and server is asynchronous.
102     // In this case we must run evaluteSamples() in another thread.
103     std::unique_ptr<std::thread> thread;
104     server.onEvaluateSamples([](int64_t spp, int64_t remainder, int tileSize){
105         if(thread && thread->joinable())
106             thread->join();
107         thread.reset(new std::thread(evaluateSamples, spp, remainder, tileSize));
108     });
109     // finish() will be called when the server is done with the current scene.
110     server.onFinish(&finish);
111     // STEP 3: Run the server.
112     // This is a blocking call that will return only when the benchmark server decides.
113     server.run();
114     return 0;
```

```
115 }
```

A.5 Classes

A.5.1 BenchmarkClient

```
1  /**
2   * \brief The BenchmarkClient class is used to communicate with the benchmark server.
3   *
4   * This class provides query methods that allows a technique to get information about the scene,
5   * request samples to be rendered, end send the final result.
6   *
7   * Only one instance of this class should be created in your program.
8   */
9  class BenchmarkClient
10 {
11 public:
12     using TileConsumer =
13         std::function<void(const BufferTile&)>; //!< Tile consumer callback function (SPP version).
14     using TileProducer =
15         std::function<void(const BufferTile&)>; //!< Tile producer callback function (SPP version).
16     using TileConsumer2 =
17         std::function<void(int64_t count, float* samples)>; //!< Tile consumer callback function (non-
       SPP version).
18     using TileProducer2 =
19         std::function<void(int64_t count, float* samples)>; //!< Tile producer callback function (non-
       SPP version).
20
21     /**
22      * \brief Connects with the benchmark server.
23      *
24      * The BenchmarkClient object should be instantiated at the beginning of the main function.
25      *
26      * If you pass the argc and argv parameters from main(), the following command-line options
27      * become available:
28      *
29      *     --fbksd-renderer "<renderer_exec> <renderer_args> ..."
30      *       Starts the renderer process with the given arguments.
31      *     --fbksd-spp <value>
32      *       Sets the sample budget available to client.
33      *
34      * This allows you to run your client program directly (for debugging purposes, for example).
35      */
36     BenchmarkClient(int argc = 0, char* argv[] = nullptr);
37
38     BenchmarkClient(const BenchmarkClient&) = delete;
39
40     BenchmarkClient(BenchmarkClient&&) = default;
41
42     ~BenchmarkClient();
43
44     /**
45      * \brief Get information about the scene being rendered.
46      *
47      * The more important information are:
48      * - image dimensions (width and height);
49      * - maximum number of samples available.
50      *
51      * This information allows you to configure your technique accordingly.
52      * See SceneInfo for more details.
53      *
54      * \return SceneInfo
55      */
56     SceneInfo getSceneInfo();
```

```
57
58      /**
59       * \brief Sets the sample layout.
60       *
61       * The layout is the way to inform the FBKSD server
62       * what data your technique requires for each sample, and how the data should
63       * be laid out in memory.
64       * Common data includes, for example, color RGB and (x,y) image plane position.
65       *
66       * This method should be called just once.
67       */
68      void setSampleLayout(const SampleLayout& layout);
69
70      /**
71       * \brief Returns a pointer to the buffer where the result image is stored.
72       *
73       * The user should write the final reconstructed image to this buffer before calling
74       * sendResult();
75       *
76       * The layout of the image in memory follows a scan-line pixel order:
77       * matrix of pixels with `width` columns, and `height` lines, with each pixel having
78       * R, G, and B values, in this order.
79       *
80       * @note BenchmarkClient owns the buffer: do not delete.
81       */
82      float* getResultBuffer();
83
84      /**
85       * @brief Request samples.
86       *
87       * The samples are computed by the renderer and sent in tiles. For each tile, the callback
88       * function BenchmarkClient::TileConsumer is called so you can consume the samples.
89       *
90       * @param spp
91       * Number of samples as a multiple of the number of pixels.
92       * @param consumer
93       * Callback function that will be called for each tile produced by the renderer.
94       */
95      void evaluateSamples(SPP spp, const TileConsumer& consumer);
96
97      /**
98       * @brief Request samples.
99       *
100      * This is an overload method that request a amount of samples that is not in SPP.
101      * Since the amount can be less then the number of pixels, there is no guarantee
102      * that all pixels are covered by samples.
103      *
104      * For each tile, the callback function BenchmarkClient::TileConsumer2 is called so
105      * you can consume the samples.
106      *
107      * @param numSamples
108      * Number of samples requested.
109      * @param consumer
110      * Callback function that will be called for each tile produced by the renderer.
111      */
112     void evaluateSamples(int64_t numSamples, const TileConsumer2& consumer);
113
114     /**
115      * @brief Request samples with input values.
116      *
117      * This is the method you'll use to implement adaptive techniques.
118      *
119      * As opposed to the evaluateSamples() methods, this method allows samples with
120      * INPUT random parameters. The producer callback is called for each tile to allow
121      * you to write the input values.
122      *
123      * Once the renderer computes the sample results, the tile is passed to the consumer
124      * callback so you can read them.
125      *
126      * @param spp
127      * Number of samples as a multiple of the number of pixels.
128      * @param producer
129      * Callback function that will be called for each tile required by the renderer.
```

```
     * @param consumer
     * Callback function that will be called for each tile produced by the renderer.
     */
    void evaluateInputSamples(SPP spp,
                              const TileProducer& producer,
                              const TileConsumer& consumer);

    /**
     * @brief Request samples with input values.
     *
     * This is an overload method that request a amount of samples that is not in SPP.
     */
    void evaluateInputSamples(int64_t numSamples,
                              const TileProducer2& producer,
                              const TileConsumer2& consumer);

    /**
     * \brief Sends the final result (rgb image)
     *
     * Calling this method is must be the last method you do before you exiting.
     */
    void sendResult();
};
```

A.5.2 SceneInfo

```
/**
 * \brief The SceneInfo class gives information about the scene being rendered.
 *
 * The SceneInfo class stores information about the scene being rendered.
 * You can use this information to tune the parameters of your algorithm.
 *
 * For example, to get the image size maximum number of samples,
 * you do the following:
 * \snippet SceneInfo_snippet.cpp 0
 *
 * The available query names are:
 *
 * | Query              | Type       |
 * | -------------------|------------|
 * | width              | int64_t    |
 * | height             | int64_t    |
 * | has_motion_blur    | bool       |
 * | has_dof            | bool       |
 * | max_samples        | int64_t    |
 * | max_spp            | int64_t    |
 * | shutter_open       | float      |
 * | shutter_close      | float      |
 *
 * @note Not all query names are available for all scenes.
 *
 * \ingroup Core
 */
class SceneInfo
{
public:
    /**
     *  @brief Return bool if a info with the given name and type exists.
     */
    template<typename T>
    bool has(const std::string& name) const;

    /**
     * \brief Get the a info with the given type and name.
     *
     * If the info doesn't exist, returns T().
     */
    template<typename T>
    T get(const std::string& name) const;
```

```
44
45      /**
46       * \brief Set the a value with type T, key name, and value.
47       */
48      template<typename T>
49      void set(const std::string& name, const T& value);
50
51      /**
52       * \brief Add all items from scene.
53       *
54       * If this SceneInfo and `scene` have items with the same key, they will be overwritten.
55       */
56      SceneInfo merged(const SceneInfo& scene) const;
57  };
```

A.5.3 SampleLayout

```
1   /**
2      \brief The SampleLayout class permits to specify the layout of the samples in memory.
3
4      The sample layout is the order in which the sample elements appear in memory.
5
6      To specify a layout, use the operator(), like this:
7      \snippet BenchmarkClient_snippet.cpp 0
8      In this case the samples will follow the specified layout in memory, e.g:
9      \a sample = [<IMAGE_X>, <IMAGE_Y>, <COLOR_R>, <COLOR_G>, <COLOR_B>, <repeat...>].
10
11     The elements can be random parameters, or features. The available random parameters and features
        are:
12
13     | Random parameters |
14     |-------------------|
15     | IMAGE_X           |
16     | IMAGE_Y           |
17     | LENS_U            |
18     | LENS_V            |
19     | TIME              |
20     | LIGHT_X           |
21     | LIGHT_Y           |
22
23     | Features            | Enumerable | Description                                 |
24     |---------------------|------------|---------------------------------------------|
25     | COLOR_R             | no         | Final sample radiance value                 |
26     | COLOR_G             | no         |                                             |
27     | COLOR_B             | no         |                                             |
28     | DIRECT_LIGHT_R      | no         | "Incident direct light value on the         |
29     | DIRECT_LIGHT_G      | no         | first intersection point"                   |
30     | DIRECT_LIGHT_B      | no         |                                             |
31     | DEPTH               | no         | Depth of the first intersection point       |
32     | NORMAL_X            | yes (0, 1) | World normal                                |
33     | NORMAL_Y            | yes (0, 1) |                                             |
34     | NORMAL_Z            | yes (0, 1) |                                             |
35     | TEXTURE_COLOR_R     | yes (0, 1) | Texture value (albedo)                      |
36     | TEXTURE_COLOR_G     | yes (0, 1) |                                             |
37     | TEXTURE_COLOR_B     | yes (0, 1) |                                             |
38     | WORLD_X_NS          | no         | "World position on the first                |
39     | WORLD_Y_NS          | no         | non-specular intersection point"            |
40     | WORLD_Z_NS          | no         |                                             |
41     | NORMAL_X_NS         | no         | "World normal on the first                  |
42     | NORMAL_Y_NS         | no         | non-specular intersection point"            |
43     | NORMAL_Z_NS         | no         |                                             |
44     | TEXTURE_COLOR_R_NS  | no         | "Texture value (albedo) on the first        |
45     | TEXTURE_COLOR_G_NS  | no         | non-specular intersection point"            |
46     | TEXTURE_COLOR_B_NS  | no         |                                             |
47     | DIFFUSE_COLOR_R     | no         | "Diffuse component of the final sample      |
48     | DIFFUSE_COLOR_G     | no         | radiance"                                   |
49     | DIFFUSE_COLOR_B     | no         |                                             |
50
51     When implementing adaptive techniques, you may want to generate your own random parameters. In
```

```
        this case,
52      random parameters can be given an optional SampleLayout::ElementIO flag, specifying the element
        as input
53      (the user generates the element and gives it as input to the rendering system)
54      or output (the rendering system generates the element). All features are always considered input.
55      You can set the SampleLayout::ElementIO flag when specifying the layout, or latter, using the
        setElementIO() method.
56
57      Some features can appear more than once, for example, in a path tracing renderer, the user may
        want access to
58      the world position of the first two intersections. These features are called enumerable (see
        Features table), and can be given an index specifying the corresponding intersection point (from
        0 to N). The total index N depends on the kind of scene and integrator being used.
59  */
60  class SampleLayout
61  {
62  public:
63      /**
64       * \brief Used to specify if a sample element is \e input or \e output.
65       */
66      enum ElementIO : bool
67      {
68          OUTPUT = false,
69          INPUT = true,
70      };
71
72      /**
73       * \brief Adds an element to the sample layout.
74       *
75       * \param name Name of the element
76       * \param io   Defines the element as OUTPUT or INPUT
77       */
78      SampleLayout& operator()(const std::string& name, ElementIO io = OUTPUT);
79
80      /**
81       * \brief Defines the number of the last added element.
82       */
83      SampleLayout& operator[](int num);
84
85      /**
86       * \brief Sets the element name as \a input or \a output.
87       */
88      SampleLayout& setElementIO(const std::string& name, ElementIO io);
89
90      SampleLayout& setElementIO(int index, ElementIO io);
91
92      /**
93       * \brief Returns the number of elements in the layout.
94       */
95      int getSampleSize() const;
96
97      /**
98       * \brief Returns the number of INPUT elements in the layout.
99       */
100     int getInputSize() const;
101
102     /**
103      * \brief Returns the number of OUTPUT elements in the layout.
104      *
105      * If you already have the input size, it's cheaper to call getSampleSize() - inputSize.
106      */
107     int getOutputSize() const;
108
109     /**
110      * \brief Checks if the element `name` exists and is INPUT.
111      */
112     bool hasInput(const std::string& name) const;
113
114     /**
115      * \brief Checks if any element is INPUT.
116      */
117     bool hasInput() const;
118
```

```
119     /**
120      * @brief Sets the Beckmann roughness threshold used to decompose the diffuse sample radiance
         values.
121      *
122      * The features `DIFFUSE_COLOR_{R, G, B}`, contain the diffuse part of the final sample radiance
         values (`COLOR_{R,G,B}`).
123      * A light-surface interaction is considered diffuse when the Beckmann roughness (alpha value) of
          the material
124      * is >= than the threshold.
125      *
126      * Increasing the threshold causes less energy to be included in the diffuse color features.
127      *
128      * @arg a Beckmann roughness value in the [0, inf) range.
129      */
130     void setRoughnessThreshold(float a);
131
132     /**
133      * @brief Returns the Beckmann roughness threshold value.
134      *
135      * The default value is 0.1.
136      */
137     float getRoughnessThreshold() const;
138 };
```

A.5.4 IQA

```
1  /**
2   * @brief Main API for implementing an image quality assessment (IQA) technique for fbksd.
3   *
4   * The IQA techniques are used to compute a value representing how close the results generated
5   * by the sampling and denoising techniques are from the reference images.
6   *
7   * A fbksd::IQA object should be constructed passing the `argc` and `argv`
8   * parameters from `main()`. Only one instance must be created.
9   *
10  * The following CLI becomes available in your program:
11  *
12  * ```text
13  * Usage: ./<iqa_exec> [options] <ref-img-path> <test-img-path>
14  *
15  * The IQA value is written to stdout in json format: `{"<acronym>":<value>}`.
16  * You can save the value in a file instead by using the option `--value-file`.
17  *
18  * If supported by the method, the error map can be saved by using the option `--map-file`.
19  * The file extension must be `.png`. Passing `--map-file` to a technique that doesn't provide
20  * the error map to report() has not effect.
21  *
22  * Arguments:
23  *   <ref-img-path>        Reference image path.
24  *   <test-img-path>       Test image path.
25  *
26  * Options:
27  *   -h, --help                        Displays this help message and exits.
28  *   --info                            Displays information about this IQA method and exits.
29  *   --value-file <path/value-file>    File name for the output value text file.
30  *   --map-file <path/map-file>        File name for the output error map image file.
31  * ```
32  *
33  * The two input images can be read using the loadInputImages() method.
34  * After you compare the two images and compute the IQA value, report the value by calling report().
35  *
36  * Note that there are two overloads for the report method: report(float) and report(float, const Img
      &).
37  * If your IQA method can produce an error map with values in the [0, 1) range, passing the error map
38  * is recommended. In this case, also pass the flag `hasErrorMap == true` when creating the IQA
      instance.
39  *
40  */
41 class IQA
```

```
42  {
43  public:
44      /**
45       * @brief Constructs a IQA object.
46       *
47       * If the images are not found, throws a std::runtime_error exception.
48       *
49       * @param argc
50       * argc from `main(int argc, char* argv[])`.
51       * @param argv
52       * argv from `main(int argc, char* argv[])`. Can not be null.
53       * @param acronym
54       * Acronym used as ID for this IQA technique (ex: "MSE"). Can not be empty.
55       * @param fullName
56       * Full name of the technique (ex: "Mean square error").
57       * @param reference
58       * Bibliographical reference for the technique.
59       * @param lowerIsBetter
60       * Flag indicating that lower values are to be considered better for this IQA method.
61       * @param hasErrorMap
62       * Flag indicating that this method provides an error map. If this flag is true,
63       * `report(float, const Img&)` should be called, passing the error map.
64       */
65      IQA(int argc,
66          char const* const* argv,
67          const std::string& acronym,
68          const std::string& fullName,
69          const std::string& reference,
70          bool lowerIsBetter,
71          bool hasErrorMap);
72
73      ~IQA();
74
75      /**
76       * @brief Loads the input images.
77       *
78       * The image files are read from disk.
79       *
80       * @param refImg  Reference image.
81       * @param testImg Test image.
82       */
83      void loadInputImages(fbksd::Img& refImg, fbksd::Img& testImg);
84
85      /**
86       * @brief Reports the IQA value.
87       *
88       * @note If the IQA instance was created with `hasErrorMap == false`, call this overload.
89       *
90       * @param value
91       * The IQA value obtained by comparing the reference and the test images.
92       */
93      void report(float value);
94
95      /**
96       * @brief Reports the IQA value and the error map.
97       *
98       * @note If the IQA instance was created with `hasErrorMap == true`, call this overload.
99       *
100      * @param value
101      * The IQA value obtained by comparing the reference and the test images.
102      * @param errorMap
103      * An image representing the error map, with pixel values in the [0, 1) range. Values outside
104      * this rage are clamped. The image must have the same size as the reference and test images.
105      * Also note that the map contains error values (e.g. larger is always worse), regardless of the
         flag
106      * `lowerIsBetter` passed to IQA::IQA().
107      */
108     void report(float value, const Img& errorMap);
109 };
```

A.5.5 Img

```cpp
/**
* @brief A simple three-channel image class with float data type.
*/
class Img {
public:
    /**
    * @brief Constructs an empty image.
    *
    * An empty image has zero width and height and not memory buffer allocated
    * (e.g. data() returns nullptr).
    */
    Img() = default;

    /**
    * @brief Constructs a uninitialized image of the given size.
    */
    Img(int width, int height);

    /**
    * @brief Constructs an image of the given size and initializes it with the value `v`.
    */
    explicit Img(int width, int height, float v);

    /**
    * @brief Constructs an image taking ownership of the given data.
    */
    explicit Img(int width, int height, std::unique_ptr<float[]> data);

    /**
    * @brief Constructs an image copying the given data.
    */
    explicit Img(int width, int height, const float* data);

    /**
    * @brief Returns the image width (number of columns).
    */
    int width() const;

    /**
    * @brief Returns the image height (number of rows).
    */
    int height() const;

    /**
    * @brief Returns a pointer to the internal data buffer.
    */
    float* data();

    /**
    * @brief Returns a pointer to the internal data buffer.
    */
    const float* data() const;

    /**
    * @brief Returns a pointer to pixel (x, y).
    *
    * The pointer contains the RGB data: Ex:
    * ```cpp
    * float* p = img(x, y);
    * float R = p[0];
    * float G = p[1];
    * float B = p[2];
    * ```
    *
    * @param x Pixel x coordinate in the [0, width - 1] range.
    * @param y Pixel y coordinate in the [0, height - 1] range.
    */
```

```
68      float* operator()(int x, int y);
69
70      /**
71      * @brief Returns a pointer to pixel (x, y).
72      */
73      const float* operator()(int x, int y) const;
74
75      /**
76      * @brief Returns the value of channel c for the pixel (x, y).
77      *
78      * @param x Pixel x coordinate in the [0, width - 1] range.
79      * @param y Pixel y coordinate in the [0, height - 1] range.
80      * @param c Channel c in the [0, 2] range.
81      */
82      float& operator()(int x, int y, int c);
83
84      /**
85      * @brief Returns the value of channel c for the pixel (x, y).
86      */
87      const float& operator()(int x, int y, int c) const;
88
89      /**
90      * @brief Convert the pixel values to the [0, 256) range.
91      *
92      * The algorithm is based on the "exrtopng" tool from the exrtools package (http://scanline.ca/
         exrtools/).
93      */
94      void toneMap();
95  };
```

A.5.6 RenderingServer

```
1   /**
2    * \brief Implements the server that provides samples and scene information to FBKSD.
3    *
4    * This class uses a callback mechanism. The renderer should provide the appropriate
5    * callback functions that will be called when the client makes the corresponding request.
6    */
7   class RenderingServer
8   {
9   public:
10      using GetTileSize
11          = std::function<int()>;
12      using GetSceneInfo
13          = std::function<SceneInfo()>;
14      using SetParameters
15          = std::function<void(const SampleLayout& layout)>;
16      using EvaluateSamples
17          = std::function<void(int64_t spp, int64_t remainingCount, int pipeSize)>;
18      using LastTileConsumed
19          = std::function<void()>;
20      using Finish
21          = std::function<void()>;
22
23      /**
24       * @brief Creates a rendering server.
25       */
26      RenderingServer();
27
28      RenderingServer(const RenderingServer&) = delete;
29
30      RenderingServer(RenderingServer&&) = default;
31
32      ~RenderingServer();
33
34      /**
35       * @brief Sets the GetTileSize callback.
36       *
37       * The callback is called just once when the renderer is started.
```

```
38        */
39       void onGetTileSize(const GetTileSize& callback);
40
41       /**
42        * @brief Sets the GetSceneInfo callback.
43        *
44        * The callback is called when the client asks for scene information.
45        * The callback should return a SceneInfo object.
46        *
47        * Callback signature:
48        * \code{.cpp}
49        * SceneInfo callback();
50        * \endcode
51        */
52       void onGetSceneInfo(const GetSceneInfo& callback);
53
54       /**
55        * @brief Sets the SetParameters callback.
56        *
57        * The callback is called when the client sets the sample layout.
58        * The sample layout is passed to the callback.
59        *
60        * Callback signature:
61        * \code{.cpp}
62        * void callback(const SampleLayout& layout);
63        * \endcode
64        */
65       void onSetParameters(const SetParameters& callback);
66
67       /**
68        * @brief Sets the EvaluateSamples callback.
69        *
70        * The callback is called every time the client asks for samples to be rendered.
71        * The parameters passed to the callback are:
72        * - spp: number of requested samples per pixel
73        * - remainingCount: an extra number of samples (not multiple of the number of pixels)
74        * - pipeSize: maximum number of samples allowable when creating a SamplesPipe.
75        *
76        * The callback should return true on success.
77        *
78        * Callback signature:
79        * \code{.cpp}
80        * bool callback(int64_t spp, int64_t remainingCount);
81        * \endcode
82        */
83       void onEvaluateSamples(const EvaluateSamples& callback);
84
85       /**
86        * @brief Sets the LastTileConsumed callback.
87        *
88        * The callback is called when the client consumes the last tile.
89        */
90       void onLastTileConsumed(const LastTileConsumed& callback);
91
92       /**
93        * @brief Sets the Finish callback.
94        *
95        * the callback is called when the client wants the renderer to finish and exit.
96        *
97        * Callback signature:
98        * \code{.cpp}
99        * void callback();
100       * \endcode
101       */
102      void onFinish(const Finish& callback);
103
104      /**
105       * @brief run
106       */
107      void run();
108  };
```

A.5.7 SamplesPipe

```
/**
* \brief The SamplesPipe class is the main way of transferring samples between client and server.
*
* You can thing of this as a big memory region where samples are saved by the renderer
* and read by the client.
*
* During the rendering process, a rendering thread typically:
* - instantiates a SamplesPipe
* - for each pixel to be rendered
*   - sets the position of the pipe using one of the seek methods
*   - calls getBuffer() to get a SampleBuffer for the current sample
*   - renders the sample and save the data into the SampleBuffer
*   - inserts the SampleBuffer into the pipe using the operator<<()
*
* Several rendering threads can work in parallel, each one having their own pipe.
* In this case, they have to make sure that a pipe position is not written by
* different threads.
*/
class SamplesPipe
{
public:
    /**
    * @brief Acquires a pipe for reading/writing samples.
    *
    * The constructor tries to acquires exclusive hold of a buffer tile. If no buffer tile is
     available,
    * the constructor blocks until one is available.
    *
    * @param begin, end
    * Define the window in the image where the pipe will cover.
    * @param numSamples
    * Number of samples that will be inserted in this pipe. The total number of samples
    */
    SamplesPipe(const Point2l& begin, const Point2l& end, int64_t numSamples);

    /**
    * @brief Releases the pipe, signaling that the it's ready to be consumed by the client.
    */
    ~SamplesPipe();

    /**
    * @brief Sets the pipe position using (x, y) pixel position.
    *
    * This is a more convenient whey of setting the pipe position
    * using a (x, y) pixel position.
    *
    * @param x, y
    * Pixel position.
    */
    void seek(int x, int y);

    /**
    * @brief Returns the current pipe position.
    */
    size_t getPosition() const;

    /**
    * @brief Returns the number of samples inserted so far.
    */
    size_t getNumSamples() const;

    /**
    * @brief Returns a SampleBuffer for the current pipe position.
    *
    * If the client specified a sample layout with input random parameters,
    * the returned sample buffer will contain the written by the client.
    */
```

```
67      SampleBuffer getBuffer();
68
69      /**
70      * @brief Inserts SampleBuffer in the current pipe position and
71      * advances to the next position.
72      */
73      SamplesPipe& operator<<(const SampleBuffer& buffer);
74  };
```

A.5.8 SampleBuffer

```
1   /**
2   * \brief A SampleBuffer stores data for one sample and is used to get/send data to a SamplesPipe.
3   *
4   * When the client sets the sample layout, it specifies two types of values:
5   * - Random parameters
6   * - Features
7   *
8   * Random parameters are values typically generated by a random sampler generator.
9   * These values can be specified as INPUT (provided by the client) or OUTPUT (provided by the renderer
        ).
10  *
11  * Features are the values computed by a rendering algorithm (e.g. color, depth, etc.)
12  */
13  class SampleBuffer
14  {
15  public:
16      SampleBuffer();
17
18      /**
19      * \brief Conditionally writes a random parameters value to the buffer.
20      *
21      * This method only writes the value to the buffer if the random parameter was specified as
22      * OUTPUT by the client, otherwise this call has no effect.
23      *
24      * @return The resulting value in the buffer.
25      */
26      float set(RandomParameter i, float v);
27
28      /**
29      * @brief Returns a random parameter value.
30      */
31      float get(RandomParameter i);
32
33      /**
34      * \brief Write a feature value to the buffer
35      *
36      * @return The value v.
37      */
38      float set(Feature f, float v);
39
40      /**
41      * \brief Write a numbered feature value to the buffer.
42      *
43      * Numbered features are features that have one instance for each intersection point
44      * in a path from the camera.
45      * For example, world position x can be for the first intersection point (number 0: WORLD_X) or
46      * the second (number 1: WORLD_X_1).
47      *
48      * @return The value v.
49      */
50      float set(Feature f, int number, float v);
51
52      /**
53      * @brief Return a feature value.
54      */
55      float get(Feature f);
56  };
```

APPENDIX B — FBKSD PYTHON API REFERENCE

In this chapter, we provide FBKSD Python API. To access the complete documentation, visit https://fbksd.github.io/fbksd/docs/python/latest. The main repository for the FBKSD SDK is available at https://github.com/fbksd/fbksd.

B.1 Denoising Technique Example

To create a denosing technique using FBKSD's Python API, import the module `fbksd.client`. It supports only non-adaptive denoising techniques for now. The workflow for is basically the same as in the C++ API:

1. create a `BenchmarkClient` object;
2. request information about the scene (`get_scene_info()`);
3. setup the sample layout (`get_sample_layout()`);
4. request the samples (`evaluate_samples()`);
5. reconstruct the result image, saving it to the result image buffer;
6. send the result (`send_result()`).

The code below shows a simple box filter written using the FBKSD Python API.

```
1  #!/usr/bin/env python3
2
3  import fbksd.client as fc
4  import numpy as np
5
6  result = None
7
8  # step 5
9  def process_samples(tile, offset):
10     (ny, nx, ns, ss) = tile.shape
11     (bx, by) = offset
12     res_window = result[by:by+ny, bx:bx+nx]
13     np.mean(tile, out=res_window, axis=2)
14
15 # step 1
16 client = fc.BenchmarkClient()
17
18 # step 2
19 info = client.get_scene_info()
20 spp = info.get('max_spp')
21
22 # step 3
23 client.set_sample_layout(['COLOR_R', 'COLOR_G', 'COLOR_B'])
24 result = client.get_result_buffer()
25
26 # step 4
27 client.evaluate_samples(spp, process_samples)
28
29 # step 6
30 client.send_result()
```

B.2 Classes

B.2.1 BenchmarkClient

```
class BenchmarkClient:
   """The BenchmarkClient class is used to communicate with the benchmark
   server
   """

   evaluate_samples((BenchmarkClient)self, (object)spp, (object)consumer) -> None:
   """Request samples.

      The samples are computed by the renderer and sent in tiles. For
      each tile, the consumer callback function is called so you can
      consume the samples.

      Parameters:
         **spp**
           number of samples per pixel to evaluate.

         **consumer**
           Consumer callback function. The callback should accept two
           parameters: "tile" and "offset".

           The "tile" is a numpy.ndarray containing the tile data. It
           has 4 dimensions, in this order: "height", "width", "spp",
           and "sample_size", where "sample_size" is the number of
           components of the sample specified in the
           "set_sample_layout()" method.

           The "offset" is a tuple containing the (x, y) pixel
           coordinates of the result image, corresponding to the
           upper-left corner of the tile.

           Note: Your callback function should return as fast as possible. So, try to avoid raw loops
     when consuming the samples whenever possible.
    """

   get_result_buffer((BenchmarkClient)self) -> numpy.ndarray:
   """Returns a pointer to the buffer where the result image is
      stored.

      The user should write the final reconstructed image to this
      buffer before calling "send_result()".

      Returns:
         A RGB image as a numpy.ndarray with shape "(height, width, 3)".
    """

   get_scene_info((BenchmarkClient)) -> SceneInfo:
   """
      Returns:
         Returns information about the scene being rendered.

      Return type:
         "SceneInfo".
    """

   send_result((BenchmarkClient)self) -> None:
   """Sends the final result."""

   set_sample_layout((BenchmarkClient)self, (list)layout[, (object)roughness=0.1]) -> None :
   """Sets the sample layout.

      The layout is the way to inform the FBKSD server what data your
      technique requires for each sample, and how the data should be
      laid out in memory. Common data includes, for example, color RGB
```

```
63        and (x,y) image plane position.
64
65        Parameters:
66           **layout** (*list*)
67             List of sample components.
68             A component can be a string (Ex: "'COLOR_R'"), or a pair with
69             a string and a number indicating    the enumeration value
70             (Ex: "('NORMAL_X', 1)", indication the x normal value for
71             the second intersection point). The list of available
72             components can be found in the C++ API reference.
73
74           **roughness** (*float*)
75             Beckmann roughness threshold used to decompose the diffuse sample radiance values.
76      """
```

B.2.2 SceneInfo

```
1   class SceneInfo:
2       """Contains information about the scene being rendered.
3
4       The list of available information can be found in the C++ API
5       reference.
6       """
7
8       get((SceneInfo)arg1, (str)key) -> object:
9       """Returns the value for a given key. Throws an error if the key does not exists."""
10
11      has((SceneInfo)arg1, (str)key) -> bool:
12      """Returns *True* if the given key exists."""
```

APPENDIX C — PROCEDURAL RENDERER

Our custom procedural renderer generates images from a scene consisting of pure mathematical expressions described using the ExprTk syntax[1], as opposed to geometry data used by traditional renderers. The mathematical expressions describe how the sample positions (input) translates into sample values (output).

C.1 Scene Description

The scene description is a text file with two sections separated by a # symbol. The first section sets the image dimensions (width × height), and number of samples per pixel (spp). The second section defines the mathematical expression used to compute the sample values.

The Listing C.1 defines a scene consisting of a 3D sphere centered in the $(0, 1]^3$ domain. It gets the `IMAGE_X`, `IMAGE_Y`, and `TIME` input random parameters, maps them to the sphere domain as a (x, y, z) point, and checks if the point is inside the sphere. If it is, the value 1.0 is written to the RGB output color. Figure C.1 shows the resulting image. Note that the image contains some noise related to the `TIME` random parameter.

The full list of available random parameters and features can be found in the `SampleLayout` class API documentation in Appendix A.

```
width := 500;
height := 500;
spp := 64;

#

var c[3] := { 0.5, 0.5, 0.5 };
var r_sq := 0.5^2;
var d := ((IMAGE_X/width) - c[0])^2 + ((IMAGE_Y/height) - c[1])^2 + (TIME - c[2])
    ^2;

COLOR_R := d <= r_sq ? 1.0 : 0.0;
COLOR_G := COLOR_R;
COLOR_B := COLOR_R;
```

Listing C.1: 3D Sphere scene example.

[1] http://www.partow.net/programming/exprtk/index.html

Figure C.1: Procedural 3D sphere result generated by our procedural renderer.