Improving Rendering Performance with Autotuning and Data-Driven Anti-Aliasing

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Killian Herveau

aus Karlsruhe (Baden-Württemberg)

Tag der mündlichen Prüfung: 27/01/2025
Erster Gutachter: Prof. Dr.-Ing Carsten Dachsbacher
Zweiter Gutachter: Prof. Dr. Michael Guthe

Contents

List of Figures

List of Tables

1 Introduction

Physically Based Rendering (PBR) is a method in computer graphics aiming at rendering images in a way that models the flow of light in the real world. It uses physics-based models to simulate the interaction of light with surfaces. PBR tries to achieve photorealistic, accurate and predictable representations of how materials will appear under different lighting conditions.

PBR has a wide range of applications. In the film industry, PBR is used in visual effects to integrate CG elements with live-action footage seamlessly. Architects and product designers use PBR for visualizations to give clients a better understanding of how the final product will look. It can also be used as substitute images for training neural networks for image segmentation, recognition, leading to technologies such as self driving cars. It is used in video games to create more immersive and realistic environments.

The realism achived with current techniques is so high that it can be difficult for people to distinguish between a PBR-rendered image and a real photograph. However, this realism comes at a cost. PBR is computationally intensive, requiring significant processing power and time. A single frame can take tens of hours to render, and movies usually require a bit less than 200000 frames.

The high computational cost of PBR is one of the reasons why it is a hot topic for research. As graphics hardware becomes more powerful and algorithms become more efficient, PBR is becoming more complex and sophisticated. This surge in interest thanks to new hardware capabilities is also democratizing the field, further increasing its outreach.

The process of PBR involves simulating photon paths, bouncing in the scene. Given that scenes can be arbitrarily complex, we often make simplifying assumptions to make the problem more tractable. For instance, light transport is considered instantaneous, not true for solar system scale scenes for instance. We also consider purely geometric optics, neglecting many phenomenon emerging from wave interactions such as diffraction, interference, polarization, frustrated total internal reflection, higher order frequency generation and more. However, these assumptions alone are not enough. We also use acceleration structures, such as treelets and hierarchical trees, to speed up the rendering process. These structures have parameters that we optimize depending on the scene while rendering is ongoing.

Mathematically, PBR is an integral problem. Originally formulated as a recursive integral, it captures the important terms for light transport: the material's reflection property over all angles, and the emissivity of materials. Monte Carlo (MC) methods, which are stochastic methods for computing the integral, are the most commonly used technique for this problem. The typical example of such a method is Path Tracing. Sending a ray from the camera and sampling the next ray according to the material's properties many times until a maximum depth is reached. At each step this algorithm computes the contribution of the relevant light sources to the path, and is accumulated. Finding a light can be challenging in certain scenarios where glass materials are involved. Good sampling is therefore crucial, and many methods have been developed over the years to improve this aspect.

One category of method approach this sampling issue from a different angle. Once a good path has been sampled, it is only slightly modified, trying to improve on it. This is done by mutating the path, for instance by applying a random small angle shift to the reflection off of a material. This random shift is characterized by one of several parameters that need to be adjusted. The new path generated, if it is deemed good enough, becomes the current state of the Markov chain. This process is called Metropolis Light Transport. In this thesis, we propose a way to optimize the path space parameters of such methods, such as mutation parameters during rendering for improved quality.

These algorithms are not practical for real-time rendering. For real-time scenarios, other strategies can be employed to improve image quality. One example is anti-aliasing. Anti-aliasing attempts to address the issue of imperfect pixel sampling. As images are computed with only a finite amount of samples, they also have a limited spatial resolution. This spatial sampling results in a certain category of artifacts, aliasing artifacts. Anti-aliasing consists in removing or alleviating these artifacts from the final images. It can be a post-process or start during the rendering phase, for instance by placing samples in smart locations. One improvement of this takes advantage of the data already computed at the previous. It is in fact highly likely that most computed pixels will still be on screen. Using this temporally accumulated data for spatial anti aliasing is misleadingly named temporal anti aliasing. Mainstream algorithms often use not only the image data but also depth data, sometimes material data. Recent improvements in this field include the massive use of deep learning, notably with AMD's FSR and Nvidia's DLSS. These methods use neural networks for achieving remarkable results. These methods do improve aliasing and are used as anti-aliasing in many real-time applications. In reality, they perform upscaling, i.e., they change the resolution, using lower resolution data to create higher resolution images. In this thesis, we propose another approach, for a true anti-aliasing solution. We use neural networks and propose the smallest high-performing network for anti-aliasing.

1.1 Original Contributions

In this section we briefly present the different contributions of this thesis in the context of optimizations in rendering including real time, offline and online methods. Offline rendering present a different set of tradeoffs that we can take advantage of by optimizing our models for different criteria. Costly optimization ahead of time can prove worthwhile when the optimized process is a very hot part of the code. We discuss the methods and implications of optimization in rendering in Chapter 3. Afterwards, we evaluate other forms of optimization in the context of Markov Chain Monte Carlo algorithms. Finally, we discuss our contributions regarding machine learning for Temporal Anti Aliasing using Machine Learning. Each of the following descriptions are tied to their respective published paper.

1.1.1 Model-based optimization of acceleration structures

Hybrid Online Autotuning for Parallel Ray Tracing [Her+19] This project started from the simple observation that more performance is hidden in the gap between acceleration structure parameters and scene specificity. We bridge the gap by creating a way to describe the scene in a way that suits our optimisation framework: the indicators. We use the indicators to create a model that will output an acceleration structure parameter configuration when given a scene's indicators as an input. This improved the overall performance, while minimizing the overhead of some strategies such as vanilla autotuning.

1.1.2 Analysis of acceleration structure parameter for autotuning purposes and method to find and constrain tunable parameters

Analysis of Acceleration Structure Parameters and Hybrid Autotuning for Ray Tracing [Her+21] This project stemmed from the complicated machinery behind acceleration structure builder, and their sometimes counter intuitive behavior. This leads to difficulties in choosing relevant parameters in the context of autotuning. We then proposed a method to select and analyze the parameters of acceleration structure, while also giving relatively general results thanks to our analysis being conducted on a start-of-the-art and very popular framework.

1.1.3 Online kernel size optimization for Markov Chain Monte Carlo Metropolis Light Transport

Out-of-the-Loop Autotuning of Metropolis Light Transport with Reciprocal Probability Binning [HOD23] Metropolis light transport (MLT) methods mutate an existing path instead of recomputing it from scratch. The specific mutation techniques employed can be very diverse. We choose to focus on the BSDF mutations to showcase our method allowing for the online optimization of mutation techniques parameters. We use Primary Sample Space MLT, which maps the mutations step to the unit hypercube of random numbers. In PSSMLT a normal distribution is chosen to sample the next random number, centered around the current sample, with a fixed standard deviation. We introduce a method separating this standard deviation into several values. The best value will be chosen according to the probability distribution of the surface hit. This allows to have larger mutations or smaller mutations according to the lighting scenario.

1.1.4 Minimal Temporal Anti Aliasing without artifacts

Minimal Convolutional Neural Networks for Temporal Anti Aliasing [HPD23] Temporal Anti Aliasing (TAA) is a spatial anti aliasing technique using data accumulated over the last few frames. We create a Convolutional Neural Network to solve this aliasing problem. We develop an architecture taking as input the depth, motion vectors and image data. The motion vectors are used to reproject the old pixels at their new location, increasing the amount of current information. This is then processed together with current frame data with a kernel predicting method, generating a clean output image. We use a

Recurrent Neural network (RNN) to take the newly processed output as an input for the next iteration.

1.2 Outline

This thesis is organized in four main chapters, not including the present introduction and the conclusion. Chapter 2 gives all the necessary background and theoretical considerations that this work requires. Chapter 3 presents a method to optimize acceleration structure parameters, both online and offline. It discusses the necessary tradeoffs and analyzes in details the parameter choice and its impact on optimization. Chapter 4 introduces a method for path-space optimization in Metropolis Light Transport. Chapter 5 discusses how recurrent neural networks combined to convolutional neural networks perform powerful operations with relatively light computation over the state of the art. Finally, Chapter 6 concludes the thesis' contribution in the optimization landscape, and proposes an outlook on future work in these domains.

2 Fundamentals

2.1 Ray Tracing

Ray tracing, at its core, is the simple intersection of a ray and a surface. We define a ray $\boldsymbol{R}$ by its direction $\boldsymbol{d}$, a unit vector, and origin $\boldsymbol{O}$. A scalar t is used to specify the endpoint of the ray, which can be at infinity. A ray is more precisely defined as the ensemble of all points satisfying $L = \{\boldsymbol{d}t + \boldsymbol{O} \in \mathbb{R}^3 : t \in \mathbb{R}^+\}$. We commonly use the relation:

$$\boldsymbol{R} = \boldsymbol{d}t + \boldsymbol{O}. \tag{2.1}$$

To intersect the ray with the scene we commonly use triangles, noted as a collection of 3 3D vertices $T = \{v_1, v_2, v_3\}$. In some cases, the intersected surface can be a function, we will not cover this here. The purpose of raytracing is to intersect the line L with the triangle T. To perform this intersection, several algorithms are possible. One of the most common is the Möller-Trumbore algorithm [MT05]. This algorithm returns the parameter t in the equation Equation (2.1) corresponding to the intersection with T. This value is always either a positive number or $+\infty$.

Among the settings for the camera, we give a few useful definitions for concepts that will be mentioned later. The up vector is a reference vector that is used to place the camera upright by default. The field of view is an angle that defines the visible span of the camera. In other words, a large field of view will cover more of the landscape while a lower field of view will appear zoomed in. The field of view is independent of the resolution.

In geometric optics, the propagation of light is reversible, i.e. the light would travel the exact same way in one direction or in another. This allows to consider the camera as

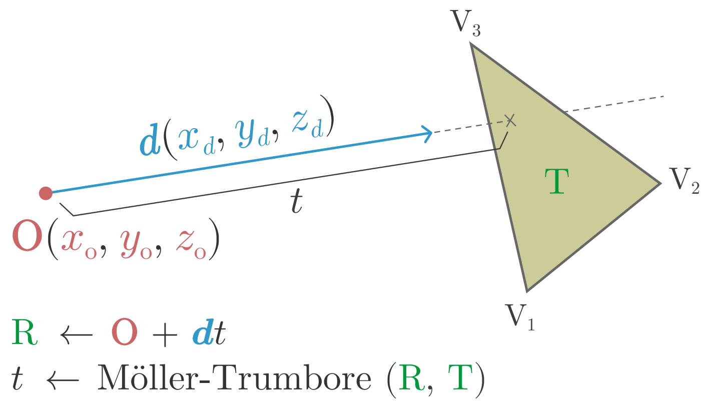

V_3
$\boldsymbol{d}(x_d, y_d, z_d)$
T
V_2
t
$O(x_o, y_o, z_o)$
V_1
$R \leftarrow O + \boldsymbol{d}t$
$t \leftarrow$ Möller-Trumbore (R, T)

Figure 2.1: Ray-Triangle intersection. The Möller-Trumbore algorithm takes the three vertices and the ray as input and outputs the t value.

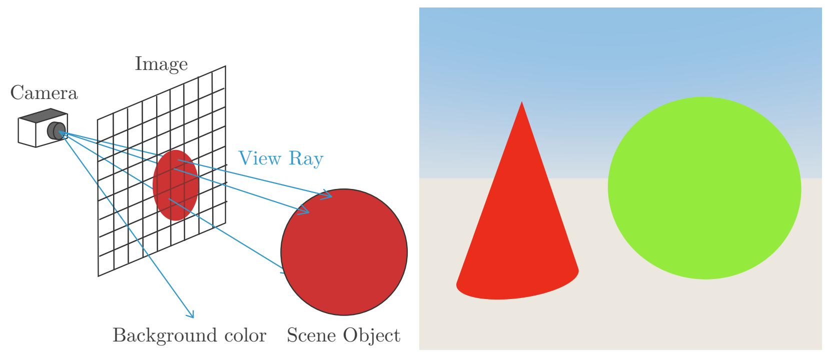


Figure 2.2: Left: The camera sends out many rays that create an image of what is behind. **Right:** The kind of image that can be produced by this simple version of raytracing.

a light source, and directly find what amount of light the camera receives from its own point of view. From it, rays of light spanning the field of view are created. Subdividing this field of view into small squares (pixels) allows for storing meaningful values into each one, such as t for a depth map from the point of view of the camera, or the color of the object, as can be seen in Figure 2.2.

Because we have multiplied the number of rays by the number of pixels N_p we subdivided the field of view into, we now have N_p intersection tests to perform. For reference, an image in $1080p$, the most common resolution for laptop or monitor screens, counts $N_p = 1080 \times 1920 = 2,073,600$ pixels.

To display complex shapes, artists will often use upwards of millions of triangles. Each ray requires its own intersection test to be certain we do find the closest surface. This means, one ray will need as many intersection tests as there are triangles in the scene. The total number of intersection tests is now $N_i = N_p N_t$, with N_t the number of triangles.

For a modest scene such as *sponza*, there are $300,000$ triangles, in 1080p, about $N_i = 6.10^{11}$ intersection tests are needed. For this, we only compute direct visibility from the camera. There is no lighting, no reflections. For these features, we need to send even more rays, which multiplies further the number of intersection tests.

2.2 Acceleration structures

This problem is alleviated through the use of acceleration structures. The main goal of acceleration structures is to remove the need for intersecting every primitive (general name for the surfaces to intersect) for every ray. The common approaches nowadays all rely on a divide and conquer basis. The foundational principle of these methods is to separate the empty space from the occupied space. Two main approaches have emerged and are now considered standard. One approach is to recursively segment the space, dividing the extent of the considered space into recursively smaller parts, segmenting only when needed. Grouping primitives whose extent is fully contained in a given space allows for testing only against the space segmentation. This segmentation is usually a grid like structure, which lends itself well to optimizations. This is the k-D tree approach. Another approach is object grouping. One can group close primitives and define a bounding volume

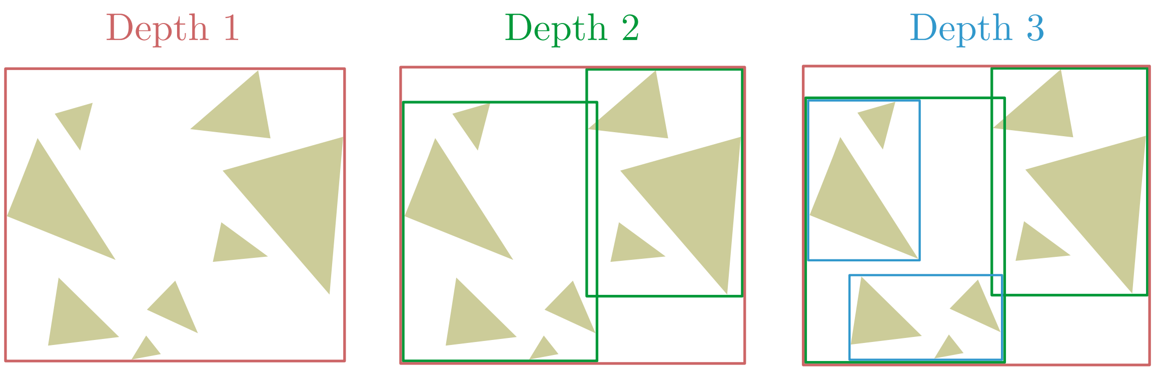


Figure 2.3: Building a BVH involves recursively separating the primitives. The bounding boxes can overlap. In our case, there is a maximum leaf size of 4.

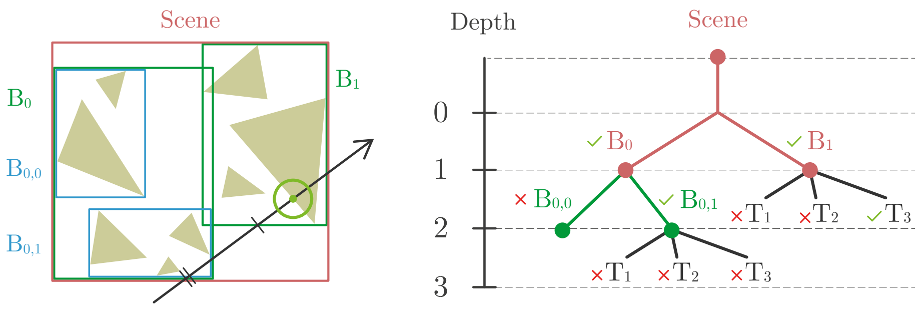


Figure 2.4: Intersecting a BVH is equivalent to searching through a tree. All intersected nodes are checked, including overlapping nodes, and when multiple valid connections are found, the shortest one is kept.

around, fully containing the primitives. This is the BVH (Bounding Volume Hierarchy) approach. We will develop this approach further.

2.2.1 Bounding Volume Hierarchy

A BVH, as we said, groups together the primitives that are close in a node. In practice, we start by subdividing all primitives into two nodes, and then subdivide each node into two nodes again, until there are only a handful of triangles per node, then called a leaf. In the 2D example in Figure 2.3, we show one possible splitting. The choice of the splitting axis (x or y) and its location should in general minimize the sizes of the bounding volumes. Splitting strategies are covered in more detail in Section 2.2.2. When tracing a ray, this allows for only $log(N_t)$ tests on average, given N_t primitives. The shape of the bounding volume chosen can vary, but most modern tools use axis aligned bounding boxes (AABBs) as these are very fast to intersect. We will consider AABBs as the bounding volume. The illustrations in Figure 2.3 summarizes the process of BVH building and Figure 2.4 summarizes the intersection part. The benefits of using an acceleration structure scale exponentially, but for smaller problem sizes like the one shown, the improvement is only marginal.

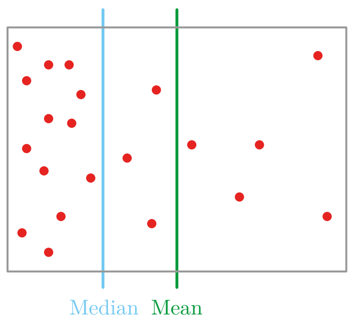


Figure 2.5: Two naive BVH splitting strategies. The mean clearly misses the cluster while the median can identify it. Note that this is a good case for the median.

2.2.2 The splitting problem

Subdividing a node, and hence selecting which primitives go into which child node, is a non-trivial process.

Among the naive approaches presented in Figure 2.5 are the mean split, simply splitting the box at its geometric center along the widest dimension, and placing all primitives whose centroid is on one side into their own bounding volume. A better approach is the median split, using some form of closeness information at the cost of an $O(n)$ computation for obtaining the median accross the largest dimension. Note that this cost is most often paid during building. This is a significant improvement over the mean split. To go beyond these two simple approaches, as different splitting strategies can lead to large differences in performance, metrics have been developed to compare acceleration structures.

The acceleration structure has one goal: make ray tracing as fast as possible. A number of elements can improve the situation:

- Reduce overall number of intersection tests
- Reduce probability to hit a leaf node without intersecting a primitive
- Reducing the overlap between bounding volumes
- Fast intersection algorithm for the bounding volume

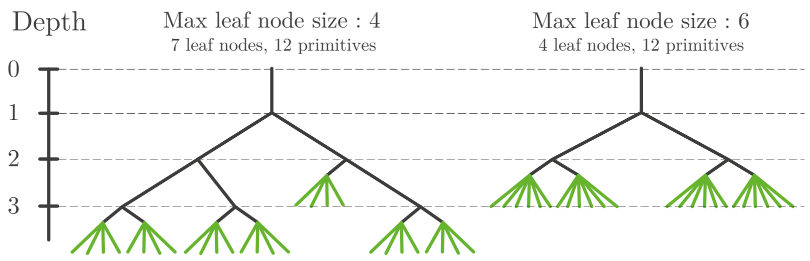


Figure 2.6: Representation of two different structures built for the same underlying set of primitives. Bigger leaf size lowers the depth of the tree.

The algorithm can also be constrained by an user parameter such as minimum or maximum leaf size, meaning the minimum or maximum number of primitives in a leaf node, as presented on the Figure 2.6. The maximum depth can also be specified. If they are incompatible (the maximum depth requiring more primitives per leaf node), it is up to

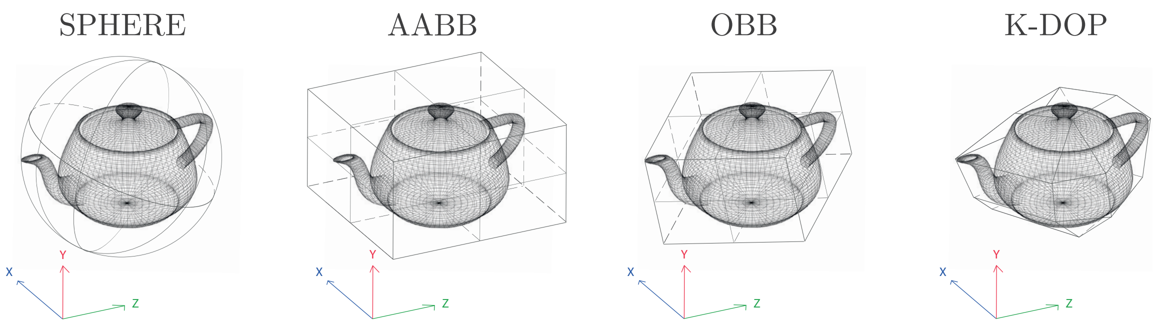


Figure 2.7: The different bounding volumes present increasingly more complexity and computational cost and memory.

the implementation whether to return an error, or proceed while breaking one, or the two parameter requirements. They should therefore not be relied upon before proper analysis. They have an impact on the considerations mentioned above, but we will expand on them in the Section 3.2.

2.2.3 Bounding Volumes

One first approach is to change the shape of the bounding volume. Indeed, many shapes are possible, and theoretically, it would be better to intersect a hull that perfectly fits the primitives inside. Experiments on different bounding volumes have yielded marginal improvements, or downright worse performance compared to Axis Aligned Bounding Boxes. The key point being that more complex shapes such as in Figure 2.7 are also more complicated to intersect. This usually improves all the points discussed above except for the high intersection cost. This cost being incurred at every intersection test, complex bounding volumes are generally slightly better for simplistic splitting strategies. When AABBs offset such a high degree of hardware optimizations, the complicated bounding volumes offer little potential at a high cost. It is an approach that can only used in very specific circumstances. The other usual bounding volume is the Oriented Bounding Box. These are often used for fur for instance. Some other shapes have specific use cases too. There is more to say on bounding volumes, for which we refer to the literature as it is not a focus of this work [Klo+98; Mei+21].

2.2.4 Surface Area Heuristic

The second approach to improving acceleration structure is to better split the primitives. This approach has seen many variations up until now, but one specific metric and its associated strategy has lead to considerable improvements, and is to this day a staple in acceleration structure comparisons. The Surface Area Heuristic (SAH) [MB90] allows the computation of the estimated cost of performing intersection tests for a specific configuration of the nodes.

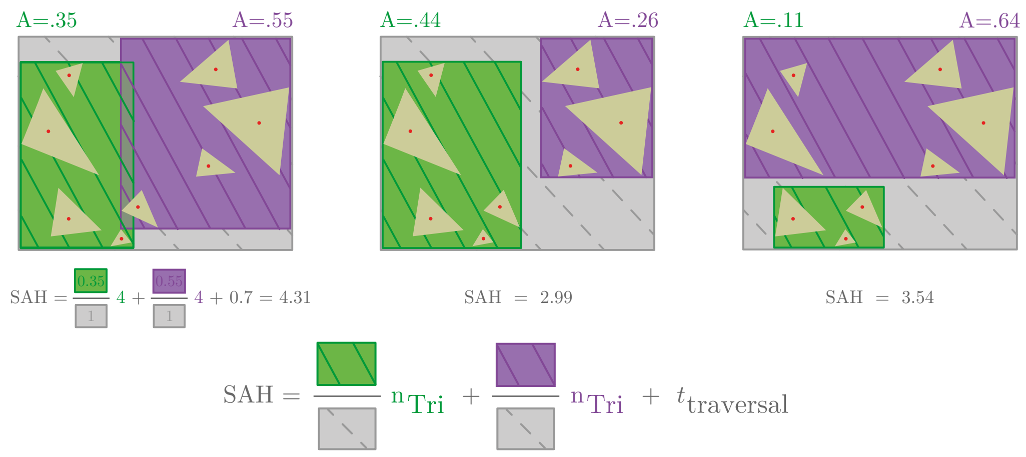

A=.35
A=.55
SAH = 0.35/1 4 + 0.55/1 4 + 0.7 = 4.31
A=.44
A=.26
SAH = 2.99
A=.11
A=.64
SAH = 3.54
SAH = nTri + nTri + ttraversal

Figure 2.8: The SAH criteria is computed for each split position and the configuration with the smallest SAH criteria is chosen.

The SAH model takes its name from a geometric observation. When a uniformly distributed random ray intersect a box of surface S_P, the probability that it also intersects the fully contained child box of surface S_C is

$$P(C|P) = \frac{S_C}{S_P}. \tag{2.2}$$

This means we can estimate the total cost C_P of subdividing a node into two nodes in terms of raytracing computation:

$$c_P(C_1, C_2) = t_{traversal} + p_{C_1} t_{C_1} + p_{C_2} t_{C_2}, \tag{2.3}$$

with $t_{traversal}$ being the cost of traversing the node P, i.e. the intersection test of the bounding volume, with the computation of the probabilities p_{C_1} and p_{C_2}. t_{C_1} is the cost intersecting all the primitives in C_1:

$$t_{C_1} = \sum_{i=0}^{N_1} t_{intersect}(i) \tag{2.4}$$

$$t_{C_1} \approxeq N_1 t_{intersect}, \tag{2.5}$$

with $t_{intersect}(i)$ the intersection cost of the ith primitive. N_1 is the number of primitives contained in C_1.

In practice, $t_{intersect}(i)$ is very similar for most kind of primitives. This cost $C_P(C_1, C_2)$ is to be compared to all possible ways to split P into different dimensions, or number of primitives as shown in Figure 2.8. The cost of not splitting the node is also to be compared, simply $\sum_{i=0}^{N_P} t_{intersect}(i)$. In our work, we will focus on the traversal cost and intersection cost values.

More sophisticated solutions use predefined split locations and binning. Indeed, subdividing the space into a fixed number of equally spaced splitting planes can reduce the cost of SAH computation when primitive counts are high. The complexity of SAH is usually $O(nlog(n))$ using the binning strategy [WG17].

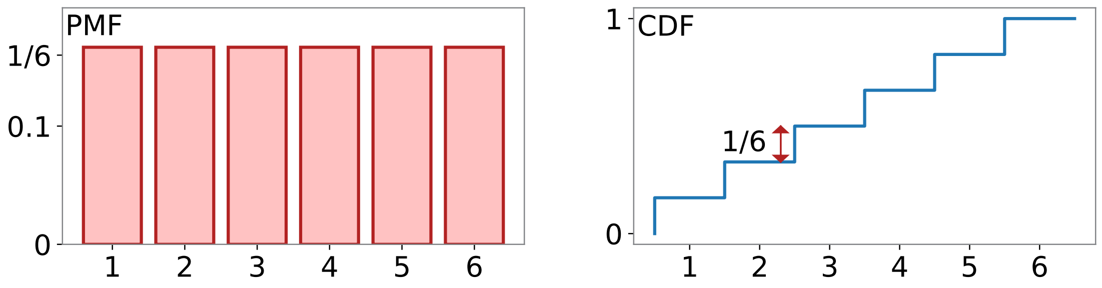


Figure 2.9: Left: Probability Mass Function (PMF), **Right:** Cumulative Distribution Function (CDF) of a 6 sided fair die.

2.3 Sampling

2.3.1 Probability density function

When an event has to be simulated, it is necessary to know its distribution of possible outcomes. When this distribution is weighted by the probability of the outcome, we call it a probability Mass function (PMF) for a discrete number of outcomes, and a Probability Density Function (PDF) for a continuous distribution.

We call sampling the act of selecting one (or several) such outcome(s), which then becomes a sample.

Discrete Probabilities

As an example, a roll of a fair six sided die would have six possible outcomes. From these outcomes, we can define events E whose ensemble is noted Ω. Each individual valid outcome denoted by their respective value $X = 1$ to $X = 6$, with equal probability $1/6$ as seen in Figure 2.9. The PDF is constant only because the die is fair. We call X the random variable.

We use the term event E to refer to any (non empty) subset of Ω that is closed under fundamental set operations. In other words an event could not be "the value of the die is even" as it breaks the closeness, but it could be "the value of the die is even and between 1 and 6". We note the collection of all valid subsets the σ-algebra $\mathcal{F}$ of the measurable space where $X \in (\Omega, \mathcal{F})$.

We usually compute the probability of events. For ease of writing and conciseness, it is common to talk directly in terms of probability of a value, i.e. $p(x)$. This makes it seem as if the probability is about x, when we are really talking about the probability of the events corresponding to each value of X, i.e., $P(X = x)$. The union of events correpond to an OR operation on the events.

If one were to simulate a six sided die with a uniform pseudorandom number generator whose output is $X \in [0, 1]$, a function mapping the $[0, 1]$ interval to the die value is needed. This is the cumulative distribution function, CDF. It is the primitive of the PDF. On the right part of the Figure 2.9 the CDF shows a step behavior. In practice, one would write an algorithm such that when $X \in [\frac{i}{6}, \frac{i+1}{6}]$, then the value of the die is $i+1$.

A PMF is a function that satisfies the Kolmogorov's axioms:

- the PMF must be non negative, i.e. no event can have 0 or negative probability:

$$P(X \in E) \geq 0, \forall E \in \Omega$$

- There is always at least one event happening: $P(\Omega) = 1$
- The union of disjoint events' probabilities is the sum of their probabilities :

$$P(\bigcup_i E_i) = \sum_i P(E_i)$$

Events are disjoints when they do not contain the same values of X: $X \in E_a \rightarrow X \notin E_b$ and $X \in E_b \rightarrow X \notin E_a$.

In our work, we encounter mostly continuous distributions. The continuous case has slightly different restrictions and definitions, which we will see in details in the following section.

2.3.2 Probabilities on the continuum

The random variable $X \in \Omega$ with Ω the (typically continuous) set of outcomes. The probability P of X being in an interval $[a, b]$ is given by its PDF f_X:

$$P(a \leq X \leq b) = \int_a^b f_X(u)du.$$

With f_X being non-negative:

$$f_X(a) \geq 0, \forall a \in \Omega.$$

And we define the CDF F_X as :

$$F_X(a) = \int_{\infty}^a f_X(u)du.$$

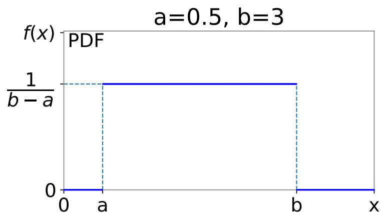
Assuming continuity, it also gives the very useful relationship using the derivative:

$$f_X(x) = \frac{d}{du} F_X(u).$$

Uniform distribution

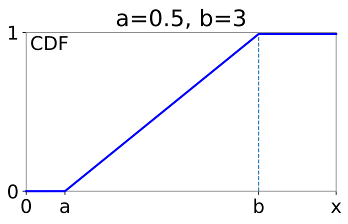
As we saw with the fair die example, its continuous counterpart is the uniform PDF. A one dimensional PDF with $x \in A$ and $A \subset \mathbb{R}$ has a PDF of:

$$f(x) = \begin{cases} \frac{1}{b-a} & \forall x \in [a, b] \\ 0 & \text{otherwise.} \end{cases} \quad (2.6)$$



The associated CDF is a simple clamped linear function. It is expressed as:

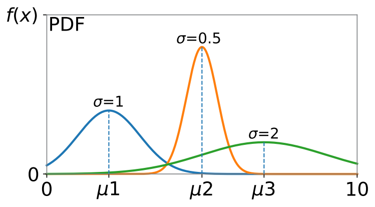
$$F(x) = \begin{cases} 0 & \forall x < a \\ \frac{x-a}{b-a} & \forall x \in [a, b] \\ 1 & \forall x > b. \end{cases} \tag{2.7}$$



Normal distribution

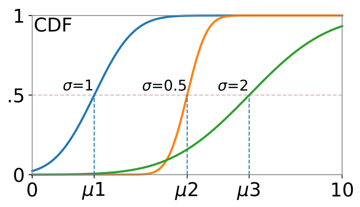
The normal distribution is ubiquitous in many fields, and our work makes extensive use of them. This distribution is noted with its parameters $\mathcal{N}(\sigma, \mu)$ respectively the standard deviation and the mean. Its PDF is:

$$f_X(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)^2}. \tag{2.8}$$



We use this PDF extensively in Chapter 4, along with the associated CDF. Note that in this example, the functions have support $\mathbb{R}$. What is shown is therefore only a part of the PDF. The CDF involves the error function:

$$F_x(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right], \tag{2.9}$$



with $\text{erf}\, x$ defined as the primitive of a normal distribution $\mathcal{N}(\sigma = \frac{1}{2}, \mu = 0)$:

$$\text{erf}(x) = \frac{1}{\pi}\int_{-x}^{x} e^{-t^2} dt.$$

This CDF graph is computed only on what is seen on the PDF graph, which implies it does not start at 0 and end at 1 vertically. This is a very important distinction as we sometimes need to define a different support on which these functions are no longer valid PDFs if the equations are unchanged. Indeed, they do not integrate to 1 anymore, as seen by the topmost value of the green curve. Encountering this issue is rare since one can evaluate probabilities quite easily on any finite subsets of $\mathbb{R}$. In Chapter 4, we encounter such an issue: evaluating an infinite number of PDFs. The issue is twofold. Computing only a finite sum is an approximation, and the terms get quickly below the numerical precision

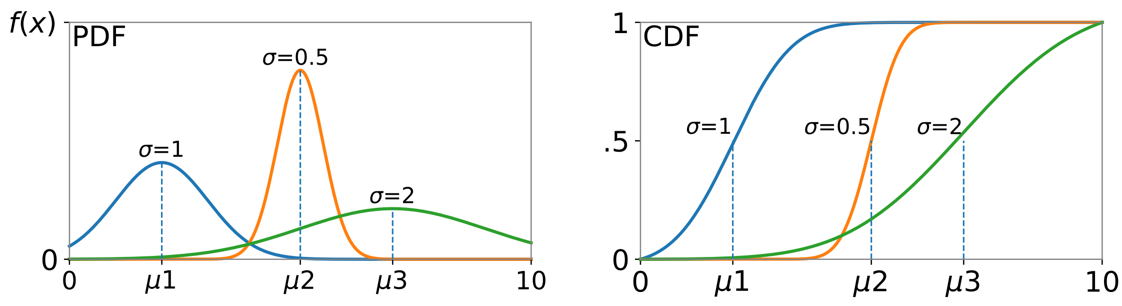


Figure 2.10: Truncated distributions of normal distributions. Note that the functions are now starting properly at 0 and ending on 1

of even double precision floating point. One way to effecively circumvent this issue is to use a truncated distribution.

Note that $\operatorname{erf}(x)$ not being expressed as elementary functions is not a problem as computers implement it very efficiently, it is less costly than a cosine call on our machines.

Truncated Distribution

It is a version of the original distribution restricted to another support, and satisfying the conditions of a probability density function.

Let X be a random variable distributed according to its PDF $f\colon \mathbb{R} \longrightarrow \mathbb{R}_0^+$, having a CDF $F(x)\colon \mathbb{R} \longrightarrow [0,1]$. Assume $[a,b]$ an interval in $\mathbb{R}$ defining the bounds of the truncated distribution $t_f : [a,b] \longrightarrow \mathbb{R}_0^+$. The general form of such a distribution is, $\forall y \in [a,b]$:

$$g(x) = f(x)I(a < x \leq b) \tag{2.10}$$

$$t_f(y) = f(y \mid a < X \leq b) = \frac{g(y)}{F(b) - F(a)} \tag{2.11}$$

$$T_f(y) = \frac{F(y) - F(a)}{F(b) - F(a)} \tag{2.12}$$

It is common to use an indicator function $I\colon \mathbb{R} \longrightarrow 0,1$ such that $I(a < x \leq b) = 1$ otherwise 0 to formally introduce a reduction of the support of f. In practice, we simply ignore anything outsite of the domain $[a,b]$.

We apply the truncated distribution to the Normal distribution seen previously. We define $t_{\mu,\sigma}$ the truncated distribution with support $[a,b]$ of a normal distribution $\mathcal{N}(\mu,\sigma)$. We show in Figure 2.10 the graph of $t_{\mu,\sigma}$ for the same values of μ and σ as in Section 2.3.2. This time, the distributions have a finite length.

These truncated distributions allow to use only a segment of a PDF, which can be very convenient.

Now, using these definitions, we will see how to generate series of random numbers following such a PDF.

2.3.3 Computing randomness

In practice, we obtain random numbers using Random Number Generation (RNG). Indeed, there is no proper source of randomness accessible from a computer where everything is deterministic. External devices can be used to generate high quality random numbers, but we focus here on algorithmic solutions.

There are different methods to generate random numbers, and many ways to evaluate their performance. We only focus on the relevant parts for our purpose. Indeed, in rendering, we mostly require very fast RNG. We describe the two main approaches used in rendering.

Pseudorandom number generation

This class of algorithms leverages the chaotic nature of certain mathematical operations (e.g. modulus) to guarantee certain properties such as their period (the number of iterations before the sequence repeats). Usually, these algorithms are quite close to true randomness and are good for most applications. The most widely used pseudorandom number generation algorithm is the Mersenne Twister. It is in the core random library of Python. Another widely used one is the PCG64, which is used in the numpy library of Python.

Quasirandom number generation

This class of algorithms sacrifices randomness quality for a better stratification and discrepancy. A low discrepancy sequence has, in all subsequences, a proportion of points proportional to the measure (size) of the subsequence. These algorithms attempt at spanning the intervals better than the pseudorandom algorithms that tend to cluster.

In rendering, the Halton sequence is very popular because it is very fast and produces decent results, especially after scrambling. The scrambling process assumes the sequence is a binary tree, and swaps the leaves. Then moving upwards to the root, swapping the subtrees at each step. Another such sequence is the Sobol sequence, having better stratification but not as easy to compute as the Halton sequence.

We compare the different RNG methods using a 2D scatter plot showing the positioning of 1024 couples $(\xi_1, \xi_2) \in [0, 1[^2$. This allows identifying methods that tend to cluster from those producing a more even distribution. We can see in Figure 2.11 different sequences produced by the Mersenne Twister and PCG64. These two algorithms perform very similarly and are both very close to true random noise. One widespread problem using random numbers is Monte Carlo integration (which we will detail later). Some of these applications benefit a lot from not having large empty spots (low discrepancy) as can be seen on the Mersenne twister or on PCG64.

An ideal sequence would always try to fill the widest gap between all samples. This is the idea behind the blue noise in Figure 2.11. This is a very low-discrepancy sequence with outstanding properties, and it is slowly becoming a standard in the industry. The biggest issue is that computing such a sequence is extremely expensive, especially in higher dimensions. Sometimes it is much simpler to use other sequences such as Sobol or Halton, as can be seen in Figure 2.12. These sequences suffer from very noticeable correlation artifacts, but scrambling helps in reducing it drastically.

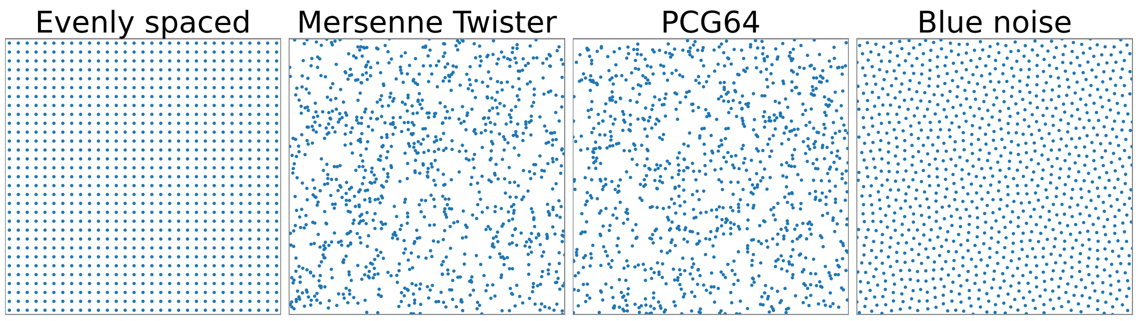


Figure 2.11: Many other RNG algorithms exist, some with an emphasis on certain characteristics such as guarantees on the maximum spread accross dimensions.

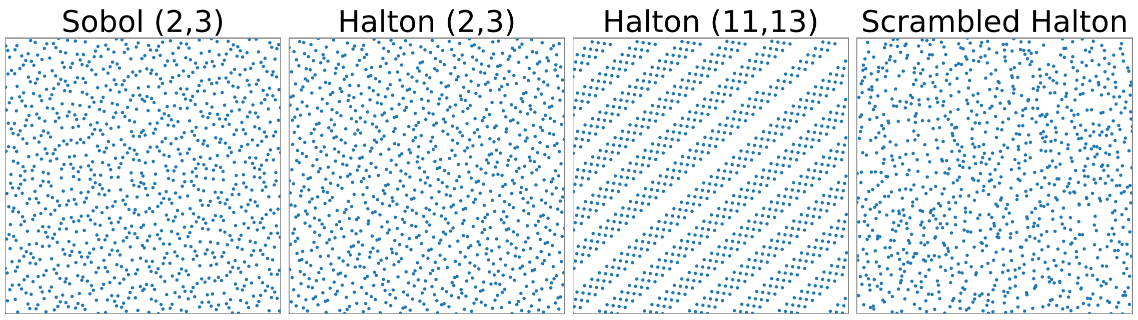


Figure 2.12: These low-discrepancy sequences show good properties in certain dimensions, but can present obvious correlation artifacts in others. Performing Owen scrambling greatly alleviates the artifacts. The Scrambled Halton presented here is (11,13), to show that scrambling does improve certain characteristics.

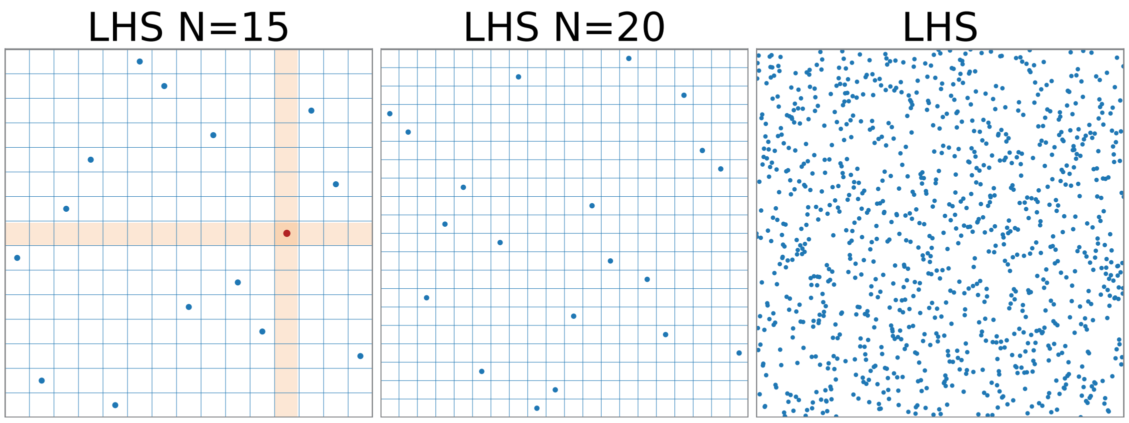


Figure 2.13: Left: Latin Hypercube sampling with centered samples. The highlight shows that no other sample is in the same row or column. **Center:** LHS wth more samples **Right:** The LHS with as many samples as the previously seen sequences. Scrambling can remove certain mathematical properties that we may want to keep.

2.3.4 Latin Hypercube sampling

Lating Hypercube Sampling is a method to stratify samples such that the space is relatively evenly explored. This requires subdividing the space in N partitions, randomly sampling withing each of these partitions. To generalize this algorithm to any dimension, each sample must be chosen such that it is the only sample in the axis-aligned hyperplane. One may consider the samples as rooks on an K dimensional hypercube. The rooks must be placed such that no one rook threatens any other.

These samples are still chosen according to the K dimensional CDF of the underlying distribution. Each sample is uniformly selected on their cell of the hypercube, then projected onto the CDF. This segmentation is done in the space of random numbers.

The Latin Hypercube is also useful for initializing a many parameters of a structure with a stratified set of parameters, spanning the entire space. This guarantees there are only a minimal amount of empty pockets in the sampling space.

2.3.5 The inverse CDF method

This section deals with creating random variables that follow a specific probability distribution. These random variables can be used for modelling certain processes involved in simulations. This is notably the case is rendering: when a ray hits a mirror surface, it can only go one way: its reflection, but for a rough surface, it can be re-emmited in many directions, and some materials have a specific distribution. We often model this behavior by considering this distribution a PDF, and then generating a random variable following this distribution. We will see the specifics of this particular case later in Section 2.5.3.

We use the first example to introduce the method, and detail the different steps. We then briefly go over different examples.

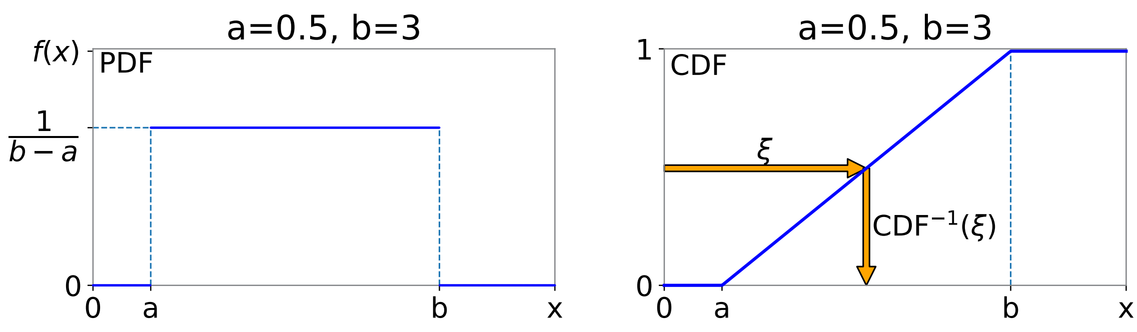


Figure 2.14: Illustration of the CDF^{-1} creating a mapping between a $[0, 1]$ and the support of the PDF.

Uniform distribution in an interval

Let $f : [a, b] \longrightarrow \mathbb{R}_0^+$ a PDF of a uniform distribution. In Figure 2.14, focusing on the CDF, one can see a mapping from the $[0, 1]$ interval to the $[a, b]$ interval. This can be used in general to create a random variable is explained by the properties of the CDF, notably that it is in $[0, 1]$ and that it is a non decreasing function. Let the corresponding CDF of f be F which, by definition is:

$$F(x) = \int_a^x \frac{1}{b-a} \tag{2.13}$$

$$F(x) = \frac{x-a}{b-a}, \tag{2.14}$$

The inverse of the CDF gives us a draw in $[a, b]$ from a draw in $[0, 1]$, as shown by Figure 2.14. Let $\xi \in [0, 1]$ be a uniform random number. The inverse CDF is computed by replacing F by the random number ξ, and x by the inverse CDF as follows:

$$F(x) = \xi = \frac{x-a}{b-a} \tag{2.15}$$

$$x = a + (b-a)\xi \tag{2.16}$$

$$F^{-1}(\xi) = a + (b-a)\xi \tag{2.17}$$

Using 1000 samples, we show that we obtain a uniform distribution on $[a, b]$, as seen of Figure 2.15.

This method has one interesting advantage : the calculation of the CDF can be performed numerically from the PDF without the need of an explicit closed form equation. This is very practical as sometimes the PDF is either too complicated to integrate, or is not conveniently expressible in terms of readily available functions.

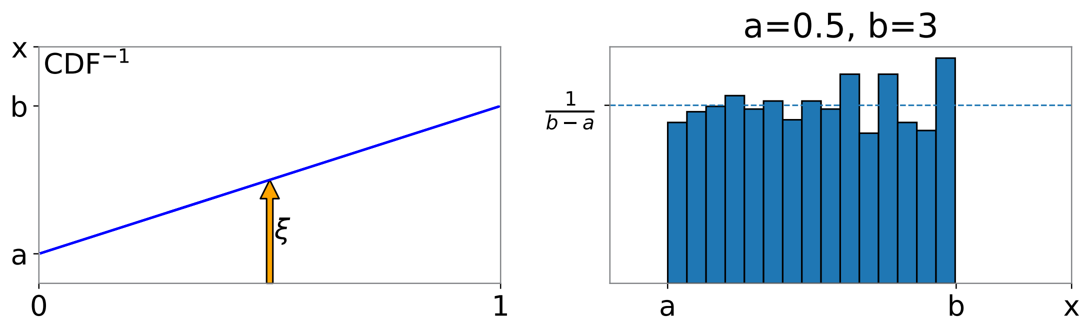


Figure 2.15: The inverse CDF produces the expected inverse distribution.

Normal Distribution

For a Normal distribution $\mathcal{N}(\sigma, 0)$, we have the PDF f_σ, the CDF F_σ and its inverse F_σ^{-1} such that:

$$F_\sigma(x) = 0.5 + 0.5\,\mathrm{erf}(\frac{x-\mu}{\sqrt{2}\sigma}) \tag{2.18}$$

$$F_\sigma^{-1}(\xi) = \mu + \sigma\sqrt{2}\,\mathrm{erf}^{-1}(2\xi - 1) \tag{2.19}$$

Using a Normal distribution comes with a few caveats that can be problematic in certain circumstances. Indeed, because the support of the normal distribution is infinite, the CDF^{-1} also goes to infinity, at 0 and at 1. Because the normal distribution is an exponential, numerically computing the value of the inverse CDF close to 0 or 1 quickly returns *Inf*. To alleviate this problem, a finer resolution may be used, as shown in the comparison between the top right and bottom left of Figure 2.16. This cost may not be realistic and/or practical, and sometimes this imprecision does not matter much as very little happens in these regions. A better way to handle this situation is to truncate the distribution, as we will see in the next paragraph.

Truncated Normal Distribution

The general formula for sampling from a truncated distribution is:

$$T^{-1}(\xi) = F^{-1}(F(a) + \xi(F(b) - F(a))), \tag{2.20}$$

with $a, b \in \mathbb{R}, a < b$. In this case, a and b are the bounds of truncation. T^{-1} is the inverse CDF of the truncated distribution, ξ is the random number and F and F^{-1} the CDF and its inverse of the untruncated distribution.

Applying this to the previously mentioned Normal distribution:

$$F_\sigma(x) = 0.5 + 0.5\,\mathrm{erf}\left(\frac{x-\mu}{\sqrt{2}\sigma}\right), \tag{2.21}$$

we obtain:

$$F_\sigma^{-1} = \mu + \sqrt{2}\sigma\ \mathrm{erf}^{-1}(2(F_\sigma(a) + \xi(F_\sigma(b) - F_\sigma(a))) - 1), \tag{2.22}$$

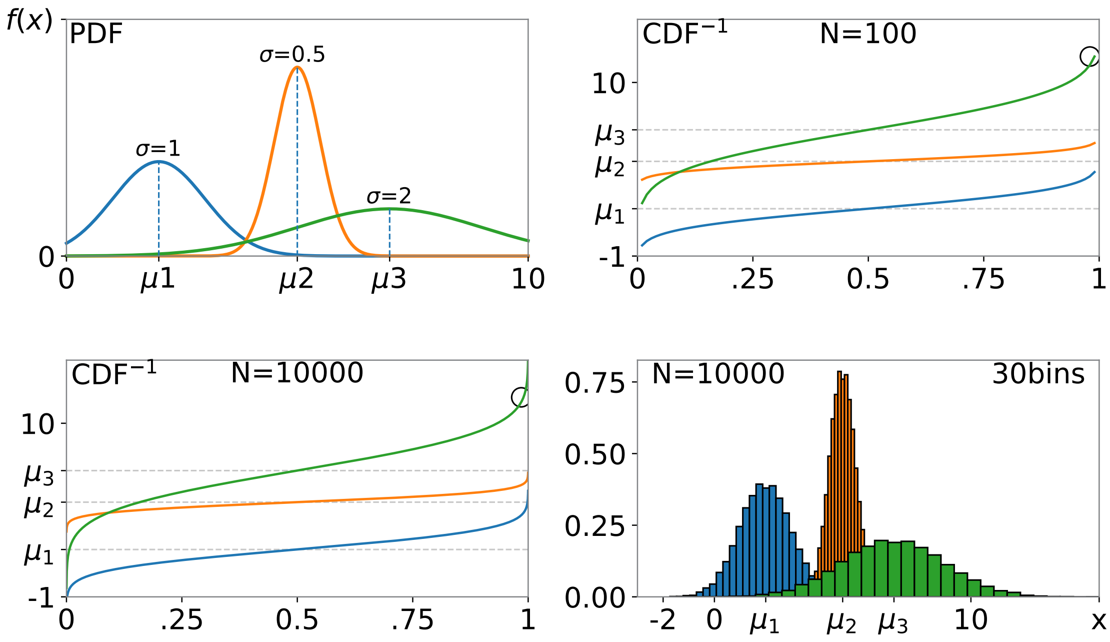


Figure 2.16: Top Left: The (relevant part of the) PDF of a Normal distribution. **Top Right:** The CDF^{-1} of this distribution. **Bottom Left:** The CDF^{-1} with more resolution. These three graphs diverge to infinity at 1 and 0 mathematically, but numerically evaluates to *Inf*. **Bottom Right:** The histogram obtained using the inverse CDF shown before.

with $\xi \in [0, 1]$ being a uniform random number and the F_σ^{-1} inverse CDF of the truncated distribution.

The truncated distribution guarantees all the desirable properties of the Normal distribution and a much more practical and numerically stable way to generate samples and compute their probabilities. The interesting point here is that the inverse CDF is not in bijection with an infinite support anymore but only between the decided bounds (0 and 10 for our example), as is shown in Figure 2.17. On the top right, one can see that the blue line suddenly jumps to 10, or close to it. This behavior can be preferred over simply outputting *NaN*. Note that now, there can not be any stray sample generated far from the region of interest, all samples are guaranteed to be in 0 and 10. This property can often reduce the amount of sampling needed, and this method also gives a way to mask a portion of a distribution.

2.4 Monte Carlo Integration

In Section 2.3, we have discussed the creation of random variables following specific distributions. One of the main application of this in rendering is for Monte Carlo integration. When we try to compute an integral, the Monte Carlo integration technique makes use of random samples whose distribution have a high impact on the convergence speed and accuracy. In this section, we discuss the use of Monte Carlo integration in rendering, and

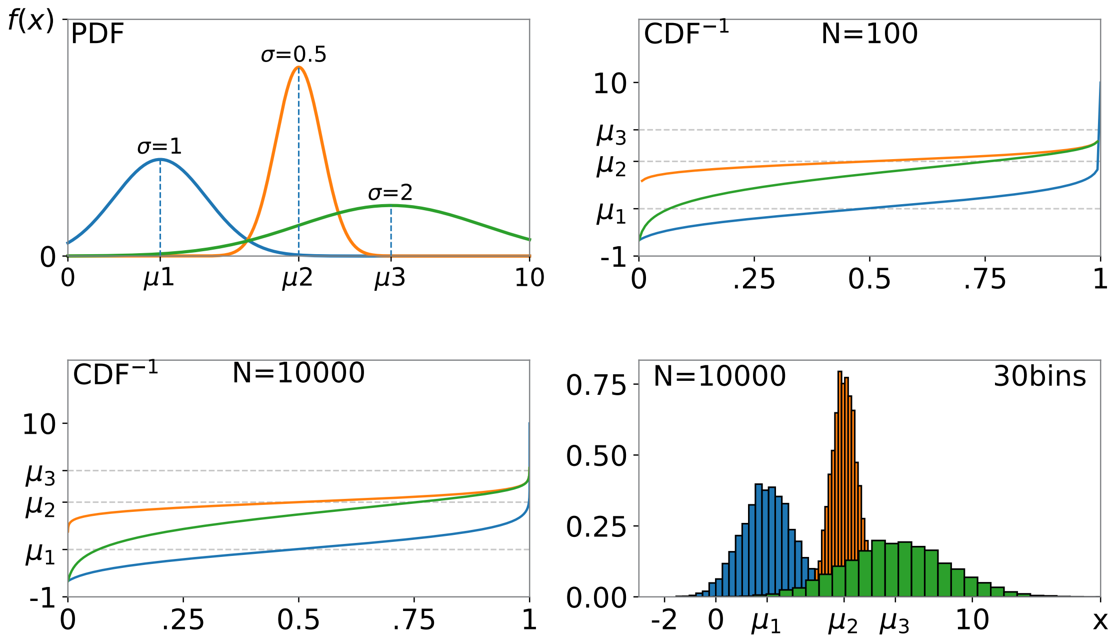


Figure 2.17: Top Left: The (relevant part of the) PDF of a Normal distribution. **Top Right:** The CDF^{-1} of this distribution. **Bottom Left:** The CDF^{-1} with more resolution. These three graphs diverge to infinity mathematically, but numerically evaluates to *Inf*. **Bottom Right:** The histogram obtained using the inverse CDF shown before.

in particular the use of Multiple Importance Sampling (MIS), a very popular technique taking fully advantage of the different PDFs used to model different visual phenomena.

2.4.1 Integration problem

Integrating a function can be done a variety of ways, typically in 1D and for simple and smooth functions, a rectangle or trapezoidal approach can be used. This works relatively well in one dimension. For multidimensional integration problems, such as the ones encountered in rendering, grid approaches quickly meet their limits as the dimensionality increases. Monte Carlo integration offers a general framework for solving integration problems. Let f be the function of interest defined over Ω and its integral be F such that:

$$F = \int_{\Omega} f(\bar{\mathbf{x}}) d\bar{\mathbf{x}}.$$

To estimate this integral, we use the estimator:

$$F_N = \frac{1}{N} \sum_{i=1}^{N} \frac{f(X_i)}{p(X_i)}, \tag{2.23}$$

with X_i a random variable distributed according to the PDF p.

One important property of this estimator is that as long as p is non zero over Ω when f is non zero, then $lim_{N\to\infty} F_N = F$.

The choice of p is free in the sense mentioned above, but not all p are equally good. An optimal choice can be theoretically derived from the variance calculation. We note σ^2 the variance of f and σ its standard deviation:

$$\mathrm{Var}(F_N) = \frac{1}{N^2}\sum_{i=1}^{N}\mathrm{Var}(f_n) \tag{2.24}$$

$$\mathrm{Var}(F_N) = \frac{1}{N}\sum_{i=1}^{N}\sigma^2. \tag{2.25}$$

We note that Monte Carlo algorithms improve the variance linearly with the number of samples.

Using an optimal density distribution is analogous to setting the variance to 0.

$$\mathrm{Var}(F_N) = 0 = \frac{1}{N}\left(\int_\Omega \frac{f(\bar{x})}{p}d\bar{x} - I\right) \tag{2.26}$$

$$\implies I = \int_\Omega \frac{f(\bar{x})}{p}d\bar{x} \tag{2.27}$$

$$\implies p = \frac{f}{I} \tag{2.28}$$

In practice one cannot use the result of the integration to compute the integration of course, but it tells us that having a probability density that is as close as possible will help reduce the variance of the estimator. Importance sampling is a great tool to model a PDF that is well adapted to the use case, which will reduce the variance of our estimator.

The PDF of a complex process can be difficult to formulate in one single expression. Using several PDFs, each specialized in one part of the process can make this task much simpler. When the process is better described with multiple PDFs, we use Multiple Importance Sampling (MIS) [Vea98]. This is the mathematical framework used to combine multiple PDFs. One such use case is light sampling and BSDF sampling. Indeed, they are two competing processes with their respective PDFs, and standard importance sampling cannot account for this.

To generate one sample from n PDFs $p_1, \ldots, p_n$, the MIS process uses a corresponding set of probabilities summing to one $c_1, \ldots, c_n$ to select the PDF that will be used. Let $w_1, \ldots, w_n$ be the set of weighing functions associated to the estimator F called the one-sample estimator of f. We note X_i a sample drawn according to the distribution p_i. The MIS estimator is then:

$$F = \frac{w_i(X_i)f(X_i)}{c_i p_i(X_i)}. \tag{2.29}$$

The functions w_i are typically chosen following the balance heuristic:

$$w_i(x) = \frac{p_i(x)}{\sum_{j=1}^{n} p_j(x)}, \tag{2.30}$$

but the $w_1, \ldots, w_n$ only need to sum to one when $f \neq 0$ and be zero when $f = 0$. Conversely it means that there needs to be at least one PDF capable of sampling f when f is non zero.

2.5 Physically Based Rendering

Raytracing as we have seen previously is not a simulation of any phenomenon per se, and physical units are not used in this case. Physically based rendering exploits real-world data and accurate physical simulation for delivering a correct depiction of light transport. Within the common assumptions of this field, such as instantaneous light transport, we develop techniques to estimate these physical quantities. These quantities are then used to model the interactions between light and medium. The study of the physical quantities related to light is called Radiometry, which we will detail in the next section. We introduce the rendering integral, relating the radiometric quantities into a recursive expression. We introduce the material models relevant for our project. Path tracing, one of the first algorithms that enabled PBR, is detailed and we also provide the mathematical ground on which it is built. Finally, we propose another perspective on solving the rendering integral with Metropolis Light Transport.

2.5.1 Radiometry

To study light transport we first define the units and the physical meaning behind several different concepts. Properly defining these units allows for a rigorous validation of the entire rendering process, as shown by the famous Cornell-box experiment [Gor+84].

The fundamental property of light related to our goal is the time average of the Poynting vector $\langle \boldsymbol{S} \rangle = \langle \boldsymbol{E} \times \boldsymbol{H} \rangle$, with $\boldsymbol{E}$ the eletric field, and $\boldsymbol{H}$ the magnetic field strength. The Poynting vector characterizes the energy per unit of time in the direction of propagation per unit area of the wave, in W/m^2. In the case of a plane wave, an assumption only broken for diffraction rendering techniques, the Poynting vector is : $\langle S \rangle = \frac{1}{\eta_0} \langle |E|^2 \rangle$. η_0 is the impedence of free space (i.e. to be replaced with the medium's impedence when appropriate). One important quantity is the energy carried through a surface per unit of time. This quantity is called the radiant flux, which involves the radiant energy Q,

$$\phi = \frac{dQ}{dt} \quad \text{[W]} \tag{2.31}$$

$$Q = \int_{\Sigma} \frac{\langle S \rangle}{\eta_0} \cos \alpha \, dA. \quad \text{[J]} \tag{2.32}$$

α is the angle between the normal at the considered position and the direction of propagation of the light —more specifically the angle between the Poynting vector and the surface normal. Q is the radiant energy of the field passing through the surface Σ. This result derives from the computation of the time average of the Poynting vector, the carrier of electromagnetic energy in the case of a plane wave in free space. We omit its derivation as this work focuses on the rendering aspects. Certain values depend on the spatial frequency of the wave, the inverse of the wavelength. For more generality, we can write the frequency dependent radiant flux:

$$\phi_\nu = \frac{\partial \phi}{\partial \nu} \quad [\mathrm{W\,m^{-1}}] \tag{2.33}$$

In the rest of this work, we do not focus on frequency dependent phenomenon. Indeed, several phenomenon are ignored in most rendering settings for the sake of simplicity. One simple instance is the use of Red Green Blue (RGB) values to represent a color. This choice makes very tedious the implementation of a simple prism effect, where the dependency on the frequency of the outgoing angle at the surface would involve a lot of additional computation. Of course, spectral rendering is the standard when such accuracy is required.

Irradiance is the measure of the energy coming from the hemisphere onto a differential surface element dA:

$$E_e = \frac{d\phi}{dA}. \quad [\mathrm{W\,m^{-2}}] \tag{2.34}$$

The irradiance can also be called radiant exitance, depending on the point of view of the surface emitting or receiving the radiant flux.

The radiance L is analogous to the irradiance but it is directional in that it involves the solid angle Ω into which we consider the radiant flux:

$$L = \frac{d^2\phi}{d\omega d(A\cos\theta)}. \quad [\mathrm{W\,m^{-2}\,sr^{-1}}] \tag{2.35}$$

This expression is the general case where the differential surface element dA is not necessarily aligned with ω, which is compensated by the $\cos\theta$ term, with θ the angle between the surface normal and ω.

2.5.2 Rendering integral

The physical quantities of radiometry can be written as a conservation law: what goes in goes out. Absorption and emission can be accounted for directly by adding certain terms, allowing to create a set of equations relating the useful quantities. This relation is called the rendering equation, introduced by Kajiya [Kaj86]. We compute the integral L_o, the outgoing radiance in direction ω_o at position x in function of the emitted radiance L_e and the reflected radiance L_r. Intuitively, the reflected radiance must be a function of the properties of the surface hit, of the incoming light and of the angles involved.

Indeed, this recursive equation involves the Bidirectional Scattering Distribution Function f (BSDF) of the surface at x parametrized by the outgoing and incoming direction ω_o and ω_i. We will describe this term in more detail in the next section. L_r also involves the incoming radiance L_i from ω_i at x and the surface normal $\mathbf{n}$. The surface normal is defined as a unit vector orthogonal to the tangent plane of the surface at position x. This equation is illustrated in Figure 2.19.

$$L_o(x, \omega_o) = L_e + L_r \tag{2.36}$$

$$L_r(x, \omega_o) = \int_\Omega f(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot \mathbf{n}) d\omega_i \tag{2.37}$$

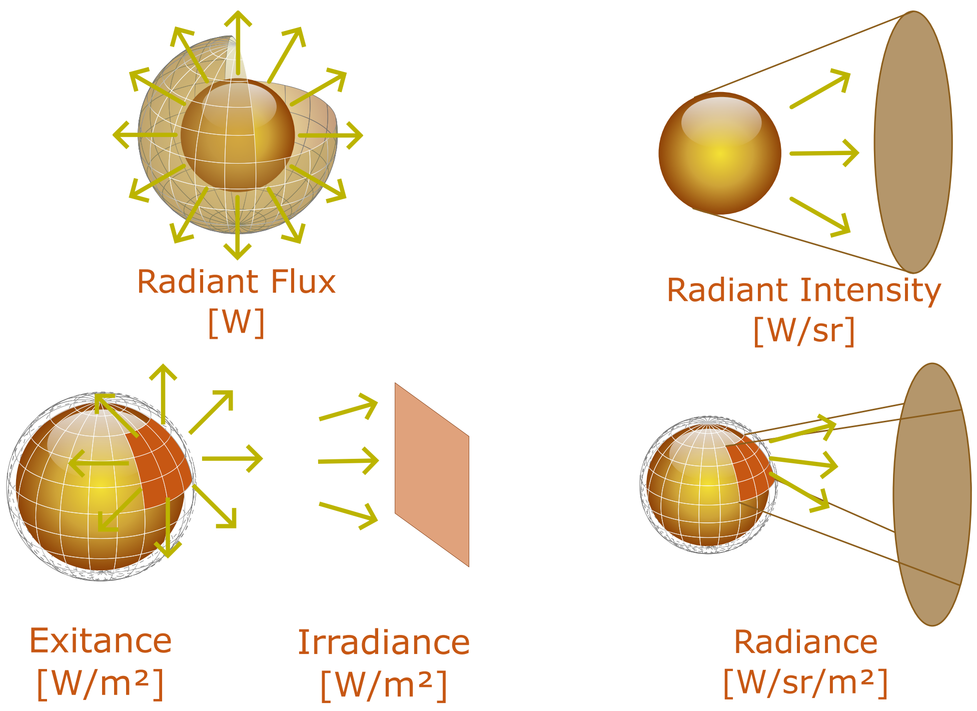


Figure 2.18: Illustration of the different radiometric quantitites involved. These quantities are better understood as integrals over surfaces.

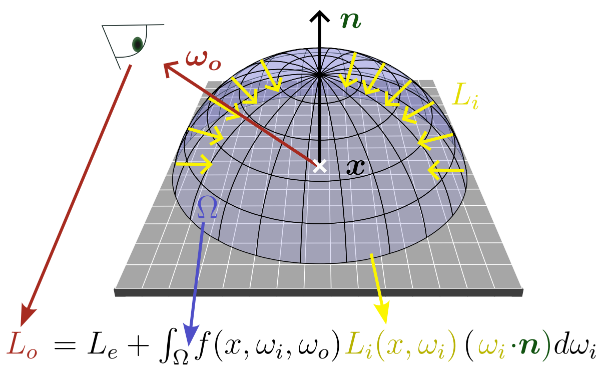

$\boldsymbol{n}$
$\boldsymbol{\omega_o}$
L_i
$\boldsymbol{x}$
Ω
$L_o = L_e + \int_\Omega f(x, \omega_i, \omega_o) L_i(x, \omega_i)\,(\omega_i \cdot \boldsymbol{n}) d\omega_i$

Figure 2.19: $L_i(x, \omega_i)$ is the radiance from ω_i received at x, while Ω is the hemisphere. The BSDF f_r represent the material's ability to reflect light in different directions, it can be different for each ω_i and ω_o.

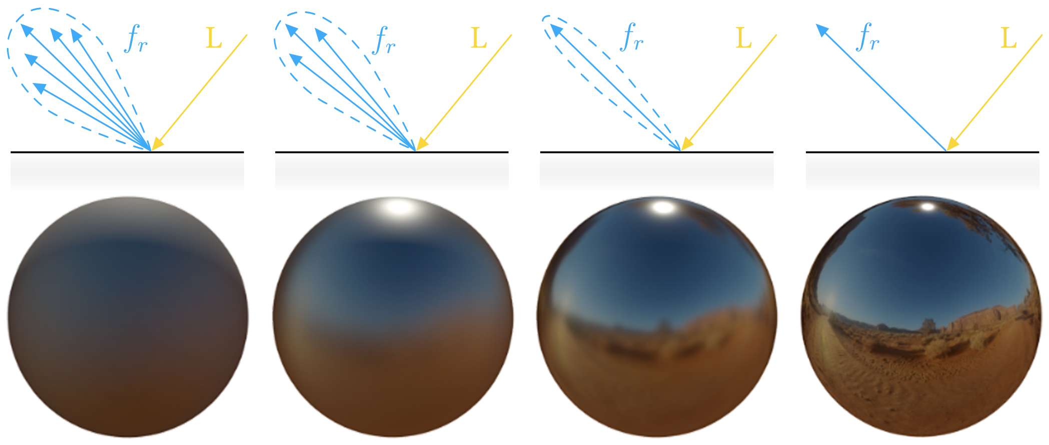


Figure 2.20: The BSDF parameter roughness changes the aspect of the material. Rougher to smoother from left to right.

This equation is recursive because computing the L_i term involves computing the L_o at each position where L_i is non zero, which itself uses L_i. Additionally, the only way for the integral to be non zero as a whole is to have non zero emission $L_e > 0$ somewhere, i.e. a light.

2.5.3 Materials

Materials can be described a number of ways. Shaders are extensively used in the video game industry to give sometimes very elaborate properties to surfaces, without necessarily a regard for light transport accuracy. In the PBR industry, material descriptions can be complex enough that entire pipelines are dedicated to them. Each aspect of the material is measured with a dedicated apparatus and all these measurements are combined to create a final material. Color, texture, 3D aspects such as bumps on a wall, or pores on the face of a human (bump map, normal map), the scattering of light in the volume of the material (subsurface scattering), everything can be measured and integrated as large texture files. This data-based approach is very costly but very effective. In the PBR communitiy, aproaches have been developed to approximate the behavior of many common materials. Materials such as dielectrics, metals, fabrics have dedicated functions allowing to represent from water to diamond, from copper to gold, and from velvet to silk. The convenience of these approaches is that only certain parameters of an equation need to be changed to use one material or another. We detail the general concept behind these functions, starting with the BSDF as shown in Figure 2.20. Note that BSDFs can be generalized further with spatially-varying BSDFs and even with subsurface scattering.

The BSDF is in fact composed of two parts: the BRDF and the BTDF where the R and T stand for Reflectance and Transmittance respectively. Intuitively, the BSDF gives a distribution of outgoing intensity over outgoing angles for every possible incoming direction of the light. This distribution corresponds to the intensity of any one outgoing direction. One can imagine pointing a directional light source toward the material, and pointing a camera at the intersection point from all directions and recording the outgoing brightness of each direction. Doing so for every angle of the light source is the BSDF. This description corresponds to a real-life apparatus called a gonio-spectrophotometer. Note that when the light and eye lie in the same hemisphere, it is the BRDF, otherwise it is the BTDF.

The BRDF and BTDF are formally defined as:

$$f_r(\omega_i, \omega_r) = \frac{dL_r(\omega_r)}{dE_i(\omega_i)} \tag{2.38}$$

$$f_t(\omega_i, \omega_r) = \frac{dL_t(\omega_r)}{dE_i(\omega_i)}. \tag{2.39}$$

L_x is the radiance and E_x the irradiance. Note that the irradiance term can be replaced by $dE_x = L_x \cos\theta d\omega_x$. Integrating these definitions yields the Equation (2.37).

Equations were proposed to model different material types. The microfacet model considers the surface as patches of surface all with different orientations. These models use a statistical description of the surface to account for properties such as roughness. Popular models are the Beckmann distribution [BS87] and the GGX distribution [Wal+07], the latter of which is used in the course of this work. Since the underlying mathematical description of these models is not explicitly used in our projects, we omit it here. We refer to the PBRT book for a more in-depth description [PJH17], to the Oren-Nayar model for diffuse reflections [ON94], and the Cook-Torrance model for specular reflections [CT82].

2.5.4 Path Tracing

As seen in the rendering equation, only lights give value to the integral. The problem now is how to make sure our random sampling will hit the lights. Indeed, Path Tracing is a technique that constructs paths one vertex at a time. After sampling an initial direction from the sensor, path tracing recursively samples new directions according to the BSDF, a process called BSDF sampling. These new directions are then used to find the next intersection. If nothing is intersected, then one considers the emission of the environment. This can be a constant value or an environment map, defining what the background should be for each angle. The process stops when it hits an emissive surface. This makes impossible the sampling of point light sources, and in general very improbable to hit lights, especially smaller ones. In practice, we explicitly sample lights at each vertex in a process called Next Event Estimation (NEE) and use a technique called Russian Roulette (RR) to choose whether to continue the path or to stop probabilistically when the contribution becomes too small compared to a user defined threshold.

Path tracing's model rewrites the rendering equation as an integral over all different paths. One can also subdivide these paths by their length, and then consider the sum of the contribution of all paths. This can be convenient to compute only paths up to a certain length for instance. We write:

$$L(x_0) = \int_{\mathcal{P}} f(\bar{x}) d\mu(\bar{x}) \tag{2.40}$$

$$d\mu(\bar{x}) = \prod_{i=0} dA(x_i), \tag{2.41}$$

where $\mathcal{P}$ is the *path space*, the set of all transport paths of all possible lengths. An individual *path* $\bar{x} \in \mathcal{P}$ is defined by a sequence of (surface) points. $d\mu$ is called the product area measure. The measurement contribution function $f(\bar{x})$ is the contribution of an individual path.

In practice, solving this equation happens as outlined in Algorithm 1, and (with some more optimizations, like NEE and MIS) produces results as shown in Figure 2.21.

```
Input:
  • $r_{o,\omega}$: contains the origin $o$ and direction $\omega$
  • Scene: geometry of the scene
  • maxDepth: maximum recursion depth

Function Radiance($r_{o,\omega_i}$, Scene, currentDepth):
    if currentDepth $\geq$ maxDepth then return $0$

    $x, \boldsymbol{n} \leftarrow$ intersection(Scene, $r_{o,\omega_i}$)
    // Russian roulette if currentDepth > 3 then
        $q \leftarrow 0.5$
        // Termination probability $\xi$
            $\leftarrow$ Random($[0, 1)$)
        if $\xi < q$ then return emittance

    end
    $\omega_o \leftarrow$ BSDFSample($x, \omega_i$)
    $p_{\omega_o} \leftarrow$ BSDFProbability($x, \omega_i, \omega_o$)
    $I \leftarrow$ Radiance($r_{x,\omega_o}$, Scene, currentDepth $+1$)
    // Apply correction factor for Russian roulette if currentDepth > 3 then
        $I \leftarrow I/(1-q)$
    end
    return emittance $+I$ BSDF($x, \omega_i, \omega_o$)$(\boldsymbol{\omega_i} \cdot \boldsymbol{n})/p$
```

Algorithm 1: Pseudo code for the Path Tracing algorithm.

Bidirectional Path Tracing (BDPT)

An extension of path tracing is Bidirectional Path Tracing. This allows the sampling of difficult light paths in situations where the lights are partially or heavily occluded. A simple glass wall could prevent Next Event Estimation from working as intended. Bidirectional Path Tracing methods start the path from both the light and the camera, checks that they can connect and computes the contribution of all the ways to sample this path. This path

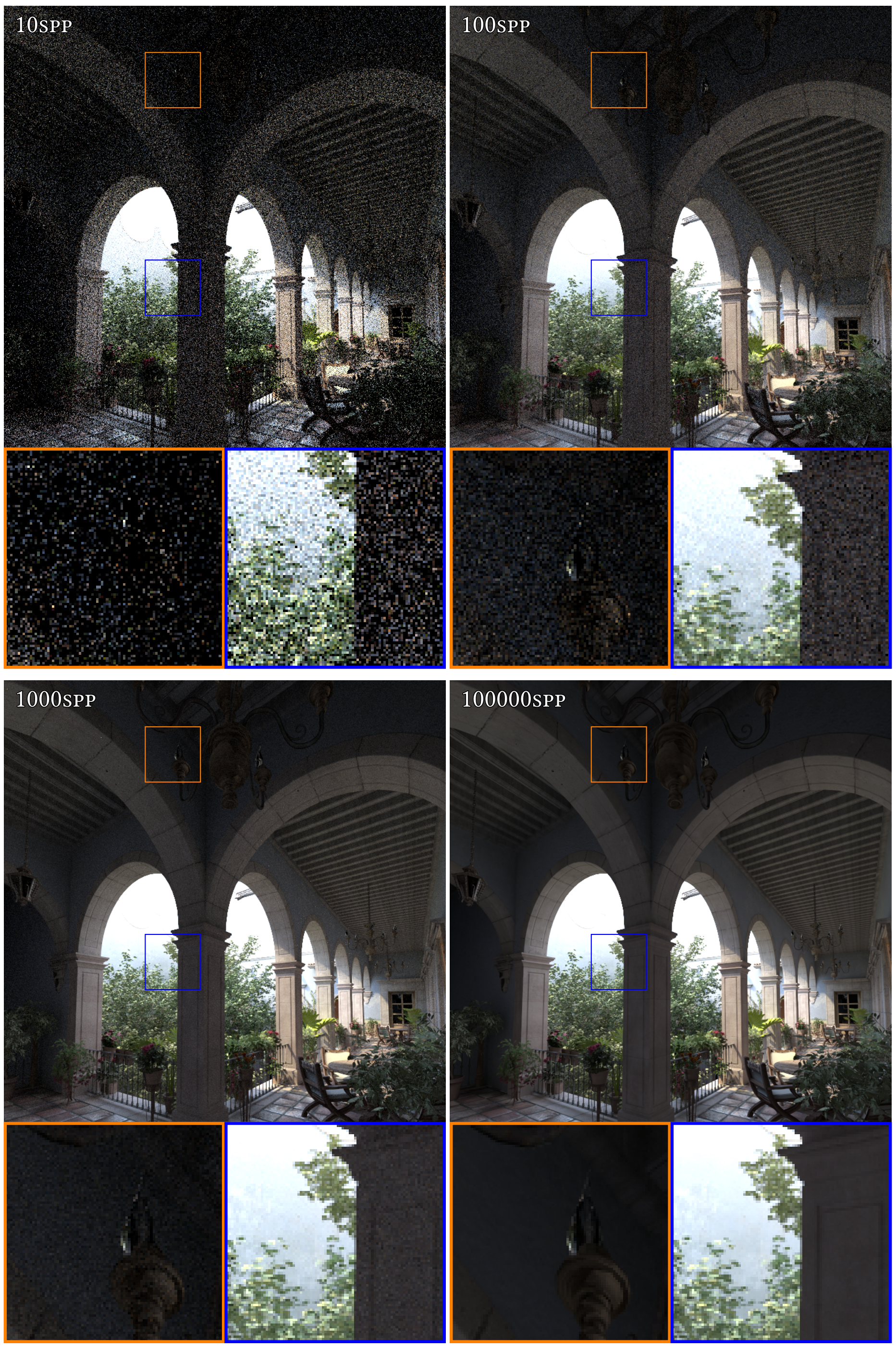


Figure 2.21: The radiance of the *san miguel* scene rendered with different numbers of samples per pixels. Many samples are needed to achieve a smooth final appearance, which we call converged.

can be sampled from the camera with 2 vertices and 5 from the light, or 3 for the camera and 4 for the light. BDPT techniques compute all the $n-1$ possibilities and combine them with the MIS balance heuristic.

This technique provides great improvements for scenes where lights that are difficult to hit from a camera ray. That being said, BDPT struggles with highly specular surfaces and with participating media (specifically inhomogeneous) as the connections cannot work in such a straightforward way. It can also significantly more difficult to implement as the number of sampling strategies grow, because they need to be properly accounted for in both directions of light travel.

2.6 Metropolis Light Transport

Metropolis Light Transport techniques are built upon any other traditional technique creating a whole path, such as Path Tracing or Bidirectional Path Tracing. MLT methods try to adress a shortcoming of the Path Tracing paradigm. Indeed, in Path Tracing, every path is fully independent, but when one high-contribution path is found, one would rather explore around this initial path, as it will likely lead to an important feature to resolve well (e.g. caustics).

The key idea is to generate a seed path with Path Tracing or BDPT, and to slightly adjust the path (mutate) in a way that favors high contribution areas. Different methods exist to slightly adjust the path and these methods are called mutation techniques (where sampling can occur). The sequence of mutations form a chain and if the mutation techniques only uses the current path to construct the next path, it is then called a Markov chain. Following the chains of the Markov Chain leads us to computing the intended integrand.

2.6.1 Overview of the method

Metropolis Light Transport of Markov Chain Monte Carlo, as its name implies, is a collection of methods working together. This section gives a high level summary.

Markov Chain

Markov Chain processes are a class of stochastic processes in which the computation of the next step depends only on the current state. From a given state, the next state is chosen according to a probability distribution. The behavior is not predictable on a chain by chain basis (as a stochastic process), but their global behavior is determined statistically. Indeed, after having explored a number of states, the Markov Chain reaches an equilibrium distribution, irrespective of the starting state. This equilibrium distribution describes the probability of each state.

The most important feature of Markov Chains is their transition probabilities from one step to the next, which in turn will define an equilibrium.

While writing this document, every morning I had to choose between two types of tea, green and black, but I don't like having the same same all the time, and I have a preference for black tea. We model this choosing process as a Markov Chain. Let green tea G and black tea B be two possible states of a Markov process, whose set of states is

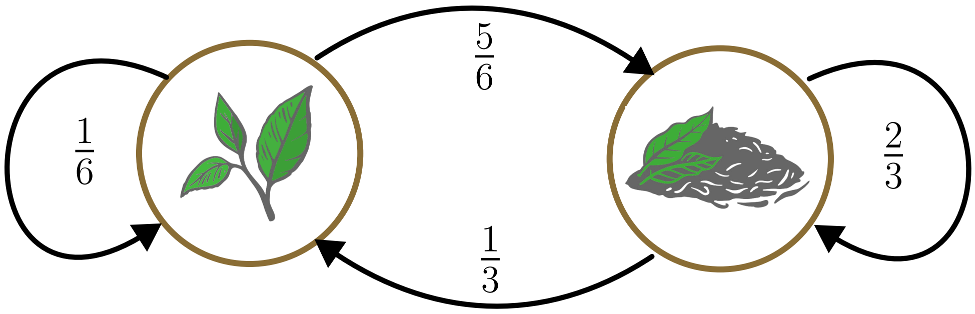


Figure 2.22: In this example, the probability to choose green tea depends only on the previous drink I had. If I had green tea before, the probability is $\frac{1}{6}$, and if I had black tea, the probability is $\frac{1}{3}$.

$S = \{G, B\}$. On Figure 2.22, the different transition probabilites are displayed on the arrows. The probabilites are not necessarily symmetric, i.e. the probability $P(G \to B)$ of choosing black tea from green tea is not the same as the probability $P(B \to G)$ of choosing green tea from black tea. More rigorously, they are analogous to conditional probabilites, written $P(G \to B) = P(B \mid G)$ and conversely $P(B \to G) = P(G \mid B)$, which is arguably confusing. These probabilities are more conveniently read as (for the first one): "the probability of B given the current state is G. Symmetry can be an important consideration as we will see in Section 4.5, but it does not affect the general structure of the process. However, it is necessary that all outgoing probabilities from a node sum to one, in the case of green tea G:

$$\sum_{X \in S} P(G \to X) = \frac{1}{6} + \frac{5}{6} = 1.$$

This example reaches quickly an equilibrium, as in, simulating the probability of encountering one or the other tend to a fixed value. In this case, the equilibrium distribution is $G = 0.2858$, $B = 0.7142$. These values answer questions regarding the long term of the chain such as: "What is the probability of choosing green tea in one year from now?".

In our case, the transition probabilities are defined by the mutation technique employed, whose description is completely decoupled from the Markov Chain itself. Contrary to the example, we are dealing with continuous variables, for which continuous distributions can be defined. In other words, the state of the chain can be the list of vertices, the mutation strategy tells us how to perform one step of the chain via the transition probabilities.

The mutation strategies correspond to the expert knowledge of the field, integrated into the simulation. The goal of these mutation strategies is to improve the convergence rate toward the desired equilibrium distribution.

The Markov Chain's progression is known on average, but there is no mechanism to guide it toward the regions we prefer. Steering the Markov Chain as well as integrating mutation strategies is the subject of the next technique, the Metropolis-Hastings Update.

Metropolis-Hastings Update

Once one step of the Markov Chain has been performed, a new set of vertices becomes the current state, and the radiance can be computed. From the radiance, which is a vector, we want to compute a scalar metric, allowing us to compare a sort of quality of the step. This scalar metric can be the average or max of the vector, or the luminance, as long as it represents something we likely want to maximize. This step quality that we can call scalar contribution can be used to steer the Markov Chain toward regions of higher contribution. By (stochastically) rejecting the step, when deemed too bad, and throwing the dice again for a better new state, the chain will find high contribution neighbourhoods. Note that the computation is not wasted even if rejected, the computed contribution is still accumulated.

This method [Has70] gives the probability of accepting the mutated state $\bar{v}$ compared to keeping the current state $\bar{u}$. This probability, called acceptance ratio, is computed using the scalar contribution function $C(\bar{u})$ of the radiance $L(\bar{u})$. Sampling for a distribution like $C(\bar{u})$ means we affect a probability density function p. Within the context of the path integral formulation for the ith pixel, p can be written as:

$$L_i(\bar{u}) = \int_{\mathcal{P}} f(\bar{u}) h_i(\bar{u}) d\mu(\bar{u}) \tag{2.42}$$

$$p(\bar{u}) = \frac{C(\bar{u})}{\int_{\mathcal{P}} C(\bar{u}) d\mathcal{P}}. \tag{2.43}$$

$$\tag{2.44}$$

The term $b = \int_{\mathcal{P}} C(\bar{u}) d\mathcal{P}$ is a constant which corresponds the average brightness of the entire scene. This can be estimated using BDPT or PT as a pre-process. MLT would still output a correct image but it would not have the correct absolute brightness.

The N-sample estimate of the path integral then becomes

$$I_i = \frac{b}{N} \sum_{i=1}^{N} \frac{L_i(\bar{u}) h_i(\bar{u})}{C(\bar{u})}. \tag{2.45}$$

This is what we want to compute by accumulating the samples given by the Markov Chain. The Metropolis-Hastings update suggests to accept or reject the proposed path given an acceptance ratio that depends on the relative contributions and probabilities. This acceptance ratio is noted:

$$a(\bar{v} \mid \bar{u}) = \min\left(1, \frac{C(\bar{v}) T(\bar{u} \mid \bar{v})}{C(\bar{u}) T(\bar{v} \mid \bar{u})}\right). \tag{2.46}$$

If accepted, $\bar{v}$ becomes the next state; otherwise, $\bar{u}$ is kept as the next state, as shown in Figure 2.23. In all cases, the contribution of the path $\bar{v}$ is accumulated in the estimator.

Mutation techniques

The original publication [Vea98] suggests several different mutation techniques. These techniques include changing the path length (i.e. adding or removing a vertex), changing the position on the sensor, and perturbations. They also use other strategies to stratify the samples over the sensor. The most relevant mutation strategy in our case is BSDF

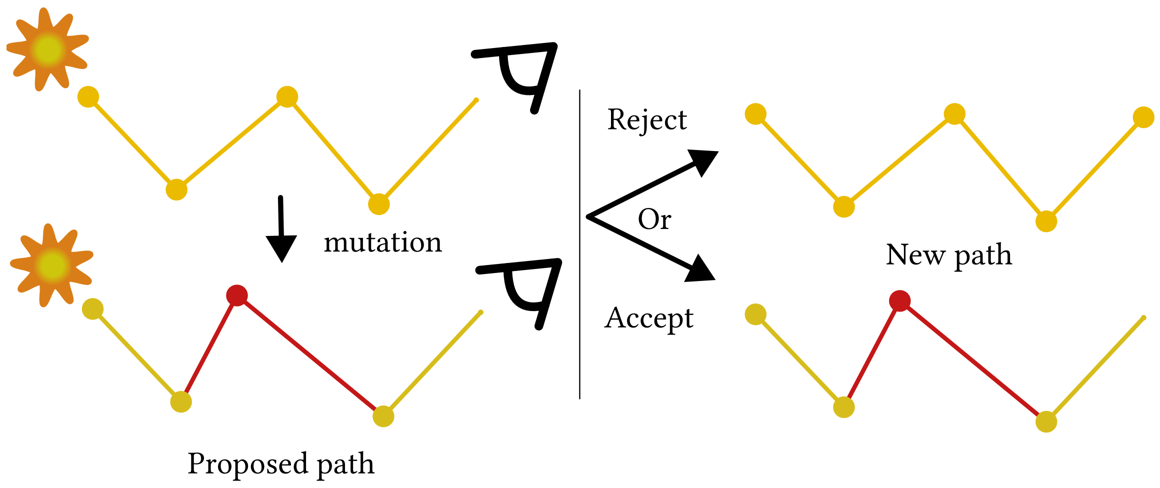


Figure 2.23: The mutation creates a proposed path. If its acceptance test ends up succeeding (it is stochastic), it is accepted and becomes the new path. Otherwise, the previous path is used.

perturbation. This mutation strategy exploits the sampling of an outgoing direction after the intersection with a surface. This sampling is performed through the BSDF of the material intersected. This BSDF directly gives a probability distribution over the outgoing angles in function of the incoming angle and surface properties. The idea behind the BSDF perturbation strategy is to sample the BSDF again in a small angular region around the previously sampled path, as can be seen on the Figure 2.24. Apart from the changed vertex, all other vertices stay in place and are reconnected, i.e. an intersection test is performed.

2.6.2 PSSMLT

Each vertex in the path can be described by the all the random numbers that generated them, hence to a path $\bar{u} = u_0, \dots, u_n$ one can associate a primary sample $s = n_0, \dots, n_k$ containing all the random numbers used to generate $\bar{u}$. Primary Sample Space MLT [Kel+02] makes the process of implementing different mutation strategies very simple by approaching the problem from a different perspective. Instead of working directly in the path space, mutating angles or sampling procedures, it uses Primary Sample Space. Note that it is very likely that $k > n$ as just a BSDF sampling requires two random numbers. The large-step mutation reinitializes the primary sample uniformly. The small-step mutation slightly changes each number in the primary sample. By operating on the primary sample directly one can decouple the actual mutation from the sampling technique.

The transition probabilities used in this approach are simple normal distributions $\mathcal{N}(\mu = x, \sigma = 0.01)$ centered on the current sample. In other words, one mutation correspond to sampling this normal distribution for all primary sample, as is illustrated in Figure 2.25. Several of these events form a (Markov) chain. In PSSMLT, it is important to use symmetric

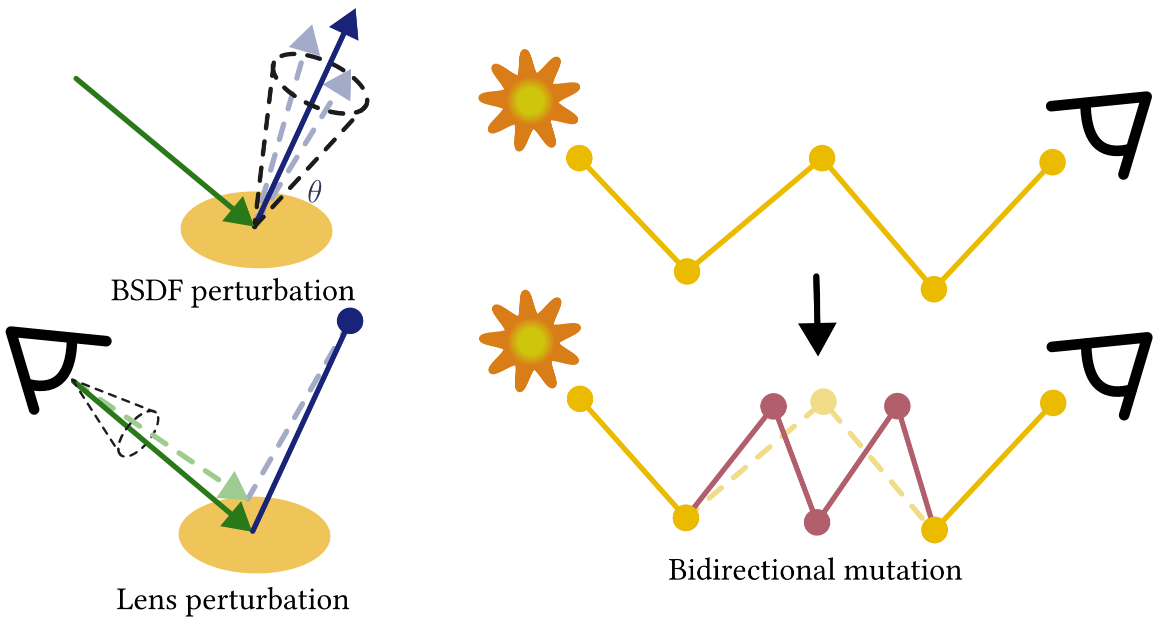


Figure 2.24: A few common ways to mutate the path. BSDF perturbation is used extensively in our work.

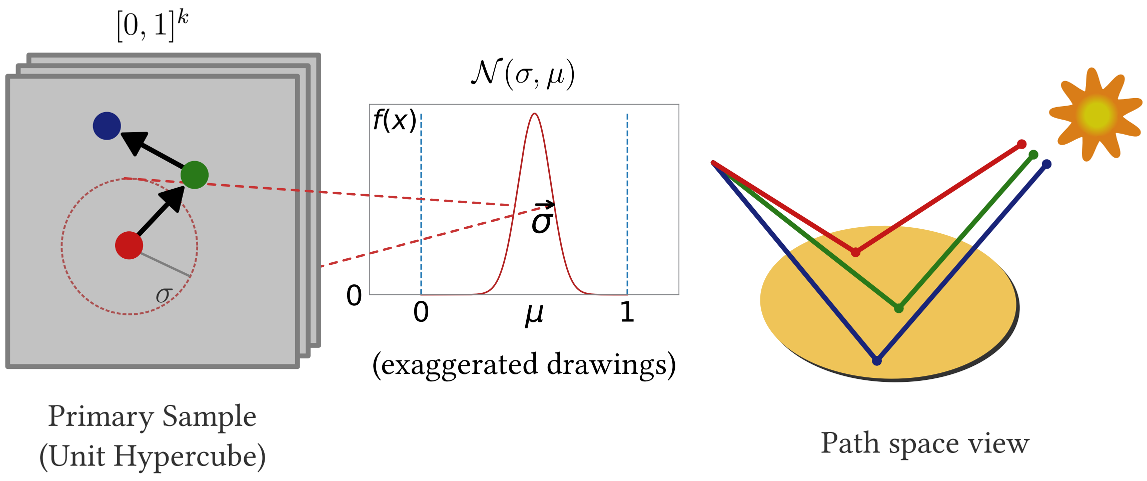


Figure 2.25: The starting point (red) is mutated in the primary sample space using the normal distribution (middle). The new point (green) has a different value in $[0, 1]$, which translates to a different path (right side). This chain continues on with the blue point.

distributions, i.e. $P(\bar{u} \mid \bar{v}) = P(\bar{v} \mid \bar{u})$. Then the acceptance ratio simply becomes the ratio of contributions, which saves some computations:

$$a(\bar{v} \mid \bar{u}) = min\left(1, \frac{C(\bar{v})}{C(\bar{u})}\right).$$

MLT techniques are a balancing act between using existing information, introducing correlations, and exploration of new areas of the image plane. To introduce a better ability to explore, MLT uses a large step mutation probability, which is a probability of sampling uniformly, which corresponds to a step of typical PT/BDPT. This introduction of randomness is necessary as otherwise, the chain would only explore the local neighbourhood of a high contribution area.

2.7 Improving Performance

Rendering in the industry can be a substantial part of the cost. Big computing clusters are used to render images, sometimes using dozens of hours for a single frame. Reducing the time spent rendering for the same quality is therefore an environmentally and economically interesting goal.

There are two main strategies to reduce the time spent rendering. The first is improving the structures that hot code depend on, mainly acceleration structures. The second is to improve sampling. Most of the sampling is performed locally, with very little scene awareness. Markov Chain Monte Carlo and related approaches include global information into the sampling scheme.

Acceleration structures

Optimizing acceleration structures involves changing the approach of building the tree compared to the tried and true greedy SAH method. Some approaches aim at reducing the cost of building the acceleration structure using the cheapest operations such as Morton codes.

LBVH [Lau+09] uses a space filling curve like the Morton codes but it also uses the SAH criteria to create treelets along this curve. This approach was later improved by Pantaleoni et al. [PL10] by clustering primitives using heuristics in the mesh data. Treelets methods employ a build fast, improve later approach. They first build small optimized leave trees and then proceed to optimize the placement of the treelet nodes. This approach has many advantages such as allowing the quick rebuilding of only a part of the geometry as other treelets can be left untouched. The method allows an iterative approach to building acceleration structures. Bittner et al. [BHH15] proposed an iterative approach with a quality comparable to the full greedy top-down approach.

The field slowly moves away from the SAH metric as the only gold standard measure and developed other metrics for improving and measuring the quality of the builders. For instance Wodniok et al. [WG17] make use of a combination of the SAH and End-Point Ovelap (EPO) metric and describe an approach improving its reliability as a performance predictor.

The decade of improvement in this field still left us with the issue of finding good parameters for these techniques. Indeed, most of them can change some threshold parameter for splitting, or the number of treelets, or the maximum number of primitives per leaf node. For instance, the previously discussed approach by Bittner et al. [BHH15] presented an incremental data structure where the number of modified nodes and the fraction of nodes to be updated are free parameters. More recently, Meister et al. [MB18] expose a configurable parameter for the density of search nodes in their reinsertion strategy.

Some if not most of these parameters are scene dependent, i.e. optimal values for the parameters can change from scene to scene. Several works established the link between scene properties and acceleration structure quality. Properties such as the distribution and overlap of triangles in a scene [SFD09] or visibility [VHS12] were found to be relevant. Dammertz et al. [DHK08] evaluated the impact of the number of triangles per leaf node of several BVH implementations and found substantial variations.

In our work, we use the existing high-quality approaches for building acceleration structures, the SAH BVH, combined with leveraging the scene awareness. We model and learn these specificities using reinforcement learning in Chapter 3. Reinforcement learning is not commonly used in BVH constructions, but it is a widely adopted framework in other rendering stragies, which we now explain in more details.

Global Sampling

The second approach is to improve convergence by performing better sampling. Path guiding is one area of research aiming at improving the local sampling using global information. Markov Chain Monte Carlo methods also use information from previous iterations to guide the new path, introducing correlations in the samples. This makes the exploration of high-contribution areas more likely, which helps in solving difficult light-transport situation. For instance, MCMC methods are known for rendering caustics particularly well. MCMC methods have been a fruitful avenue for improved sampling schemes as they lend themselves well to modifications thanks to an easily extendable mathematical framework.

The information about the scene can be leveraged to guide paths. One example is using the local geometry and their derivatives to update the path vertices, a strategy explored by Jakob et al. [JM12] known as manifold exploration.

The information about the materials can be leveraged too, as Geogiev et al. [Geo+13] uses joint PDFs to samples multiple vertices at once.

MCMC approaches can also be augmented with caches to store radiance and represent visibility information [Gru+16]. These caches can be 2 or 4 dimensional if they store directional information as well as the marginal distribution.

Several works established the benefits of machine learning for rendering. Vorba et al. [Vor+14] and later Reibold et al. [Rei+18] used Gaussian Mixture Models (GMM) to represent sampling distributions. They learn a parametrization of the GMMs using reinfocement learning. They show improvements in convergence rate for path tracing, bidirectional path tracing, and Metropolis light transport.

Our approach employs a concept related to GMMs but uses it to optimize acceleration structure parameters without modifying the underlying image synthesis algorithm.

Dahm et al. [DK17] improved the sampling scheme by using Q-learning, a model-free reinforcement learning technique. They perform training during rendering and learn to find light source paths. They successfully reduce the number of zero contribution paths and reduce the average path length.

We make use of the MCMC framework in Chapter 4 to demonstrate that performance improvements are possible within an existing strategy, by changing the parametrization of these algorithms. We employ an autotuner to select relevant parameter values for rendering, which we now explain in more details.

Input-Sensitive Autotuning

Autotuning as a tool to optimize program parameters automatically has been around for two decades. It was made popular by the ATLAS library [WD98], which optimizes BLAS primitives during installations using training examples shipped with the software. Today, the most widely used general purpose autotuning tools are ActiveHarmony[ŢCH02] and OpenTuner [Ans+14]. Both use a heuristic search to navigate the configuration search space and are oblivious to changing program inputs. ActiveHarmony uses the Nelder-Mead algorithm, whereas OpenTuner employs an ensemble of different search methods concurrently.

The PetaBricks language [Ans+09] is an example of a tuning scenario where input sensitivity is a first class property. In essence, application developers specify multiple implementations to solve a specific problem. An autotuner uses heuristical search to determine thresholds for the input size. Based on the thresholds the appropriate algorithm implementation is selected. Kicherer et al. [KBK11] present an online-learning scheme used to select an execution unit in a heterogeneous system. For a given code section, they decide dynamically at runtime whether the code should be executed on a GPU or the host CPU. Based on programmer-defined features of the code sections (e.g., “matrix size”) they train a manually built linear classifier online during program execution.

An idea close to ours in design was developed by Bergstra et al. [BPC12]. They combine a heuristic search and regression trees to optimize the parameters of a CUDA implementation of a numerical kernel. Their combination approach, however, operates in a different manner than the one we present: The regression tree model is built on training data and then serves as a surrogate for an empirical search. *Active learning* is the term referring to autotuning approaches that build and refine prediction models online. The basic principle of active learning is to use the prediction model itself to recommend samples to refine the model[Set09]. A recent instance of this method was presented by Balaprakash et al. [BGW13]. They build a performance model using dynamic trees. As part of a compiler, the model is applied to loop optimization of numerical kernels and to predict the performance of MPI codes. The Nitro system [Mur+14] is an approach to select optimal code variants in an input-dependent manner. Programmer-defined features are computed from program inputs and are used to query an SVM model. The model is built during an offline-training phase from input data. Bao et al. [Bao+16] apply model-based tuning to minimize energy consumption by selecting an optimal CPU frequency at program runtime. They use a compiler to determine relevant features for the program and create a set of profiling benchmarks to stress individual features. From this data, their approach builds a decision tree.

These related works build their models offline or online. In the former case, a-priori training data is required. In the latter case, the model is used to select training data. The performance of the configurations sampled while a model is still learning are subpar. Our approach provides a target-oriented way to train the model and to optimize the runtime performance of acceleration structures even during training.

2.8 Optimization, Autotuning and Machine Learning

Efficiency is increasingly placed at the core of new projects, from finance and travel to engineering datastructures and networks. Optimization then naturally became an indispensable tool. Rendering for instance has optimization problems at its core, as we saw in Section 2.2.2 that minimizing the Surface Area Heuristic criteria is key to obtain a well performing acceleration structure. These structures are used in varying contexts that may require different optimal solutions over time.

Parallel to this, machine learning is a related field that also involves optimization, notably for finding the best parameters of a model to accurately represent data. As we combine these approaches in our work, after giving a short overview of the fundamental optimization technique, the gradient descent, we explain the autotuning's core principles and detail the machine learning techniques relevant to our works.

2.8.1 Gradient descent

The optimization problems we face are related to the example of the gradient descent. Gradient descent is a very simple algorithm that can be improved with domain specific knowledge or additional heuristic that we will touch on further down below. It is a fundamental technique that has been built upon for decades, and is still in use today.

The gradient descent algorithm is an iterative optimization algorithm for finding the minimum of a function.

```
Input:
    • $f$: Function to optimize
    • $\theta$: Starting point
    • $\alpha$: Learning rate
    • max_iter: Maximum number of iteration
Output: Optimal point $\theta$
________________________________________
Function GradientDescent($f$, $\theta$, $\alpha$, max_iter):
    for $i$=1 to max_iter do
        $\theta = \theta - \alpha$ grad$f$ ($\theta$)
    end
    return $\theta$
```

Algorithm 2: Pseudo code for the Gradient descent algorithm

In this pseudocode:

f is the function we want to minimize. θ is the initial point or the initial parameters. α is the learning rate, which determines the step size at each iteration while moving toward a minimum of a loss function. max_iter is the maximum number of iterations. $compute\_gradient$ is a function that computes the gradient of f at the point θ. The algorithm iteratively adjusts the parameter θ in the opposite direction of the gradient of f at θ, scaling the step size by α. The procedure continues until it reaches the maximum number of iterations in this case. It is also common to stop when the gradient is small enough. The final output of the algorithm is the optimal parameter θ that minimizes the function f.

This is a simple version of the gradient descent algorithm. There are many variants of gradient descent that include additional techniques to improve convergence speed and accuracy, such as momentum, adaptive learning rates, and second-order methods. These can be particularly useful when dealing with complex, high-dimensional optimization problems, such as those encountered in machine learning.

This method can be very susceptible to local minima if no countermeasures are taken. To tackle these issues, a variety of heuristics or even specific algorithms and techniques are employed. When they are bundled together, it is common to name this program an autotuner.

2.8.2 Autotuning

Autotuning is an optimization machine. It often includes but is not limited to gradient descent. Autotuning is typically done to improve or normalize performance on different target machines that may require different parameters. For instance, one machine could have an old hard drive, which makes memory swapping (using disk space as RAM) very slow. To alleviate this issue, the autotuner could select a configuration that uses less memory at the cost of a slightly higher compute cost if it improves the overall user experience. It could also switch to using a streaming approach instead of a one-time approach to data loading, or change a data chunk size parameter.

In rendering, a small change to the architecture can lead to dramatic difference in runtime. Notably, a CPU for which the algorithm is not optimized for regarding vectorization or cache utilization could struggle even if it is newer. Not only the algorithms run on varying hardware, they also run on very different input scenes. This varying context lends itself well to the use of autotuning to adjust the parameters, optimizing the correspondance between algorithmic parameters and hardware and rendering scenario.

Nelder-Mead

In our works, we use an autotuner based on a variant of the Nelder-Mead algorithm [NM65]. This algorithm alleviates many issues of the naive gradient descent. The Nelder-Mead algorithm features 6 major steps for minimizing a function f:

1. Create $N+1$ test points $x_1, ..., x_{n+1}$
2. Sort the set $S = f(x_1), ..., f(x_{n+1})$
3. Compute the centroid $x_0 = \frac{1}{N}\sum_{i=1}^{N} x_i$. Note $N+1$ is not included.
4. Compute the reflection point $x_r = x_0 + \alpha(x_0 - x_{N+1})$
5. Expand or Contract depending of the ranking of $f(x_r)$ in the S

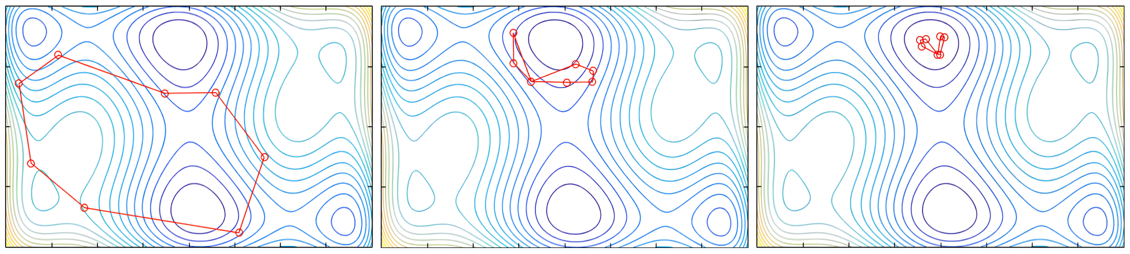


Figure 2.26: Nelder-Mead algorithm on a heightmap, converging towards the global minimum.

6. Shrink when no improvements are found
7. loop back to 2

This algorithm is itself parameterized by α, γ, ρ and σ, respectively reflection, expansion, contraction and shrink coefficients. The convergence can be viewed in Figure 2.26. The expansion phase helps in getting out of local minima, while the contract and shrink phase home in on the optimum. This algorithm is quite robust and stable, but sensitive to scale difference. This means having a large range of value in one dimension but not in another can lead to unstable convergence. This issue is addressed in our particular case by choosing appropriate bounds and normalizing our inputs between zero and one.

Commercial autotuners are typically quite complex, regrouping many different heuristics, custom algorithms, local minimum detection, best value caches and even some brute force when appropriate. Because we want to limit these effects in our work, we use a simpler version of an autotuner, consisting mostly of an improved version of the Nelder-Mead algorithm [Cha12].

2.8.3 Machine learning

Machine learning is comparatively to a decade prior, a very active field. The topics range from finance to computer vision to social studies. Machine learning offers techniques to find model parameters, find and separate clusters in data, detect outliers and much more. The basic principle is to create a model, such as an affine function $f = ax + b$. The machine learning algorithm use a set of training data to adjust the parameters (in this case, a and b) to minimize the difference between the model's predictions and the actual outcomes. This process is the training of the model, and it stops when the desired quality is achieved or when further iterating yield no improvements. Once the model is sufficiently trained, it can take previously unseen new data and make predictions.

This is the essence of supervised learning, one of the main categories of machine learning. Other categories include unsupervised learning, where patterns are found in the data without pre-existing labels, and reinforcement learning, where an (algorithmic) agent learns to make decisions by performing actions and receiving rewards.

Machine learning can be applied to specific functions or more complicated models with many parameters. An entire field of optimization sprouted from the need for the relevant mathematical expressions. Indeed, distinguishing between a parabola or the trough of a sine can be obvious in one or two dimensions, but much more difficult when they also comprise exponentials and other polynomials. Complex Curve fitting algorithms will try to model the data with increasingly complex expressions. Though this process can generate a meaningful expression where the terms and constants can represent real-world quantities, sometimes this is not at all what look for, or sometimes this solution does not scale up.

This is how the Neural Networks being universal function approximators arrive on the scene.

2.8.4 Neural Networks

Artificial Neural networks are maximally versatile. They can be used to approximate any continuous function up to any accuracy. This capability stems from the interconnectivity of the neurons. Neurons are small functions that can take an arbitrary number of inputs. They apply a weighted sum to their inputs and this goes through the transfer function ϕ.

$$n_i = \phi \sum_{j=0}^{m} w_{i,j} x_j$$

ϕ is typically a threshold function, but it can take specific shape for specific purposes, like a rectified linear function. $w_{i,j}$ is the weight affected by the neuron i to the input j. x_j is the jth input value. Note that it is common for neurons to have a bias b, implemented as $x_0 = 1$ and $w_{i,0} = b_i$. This bias allows the neuron to limit its firing rate, i.e. the likelyhood of having a non-zero output.

These weights are the most crucial element of the network. Training a network essentially consists of modifying the weights until the predicted outcome matches with the data. This matching is determined by a loss function, quantifying the distance between the network's output and the data. The weights of the network are changed using back-propagation, an efficient application of the chain rule allowing to update the networks' weights.

Overfitting

The same problems as with all other machine learning algorithms can happen, and neural networks are particularly prone to overfitting. Overfitting corresponds to learning the noise in the data, or the specific data itself, instead of generalizing. Because they are universal function approximators, if there are enough degrees of freedom, they will end up overfitting given enough training. When overfitting, a network will typically overperform on known data and vastly underperform on unseen data. Stopping the training early is one of the easiest ways to prevent overfitting. An abundance of very varied training data is also very effective. Other techniques worth mentionning are regularization and dropout. Regularization adds a penalty to the network, by either adding the sum of weight values to the loss function, or the sum of the square of the weight values. The former encourages overall lower weights while the other tend to spread out values. Dropout randomly sets a fraction of the input to 0. This prevents neurons from developing co-adaptations, making the network more robust.

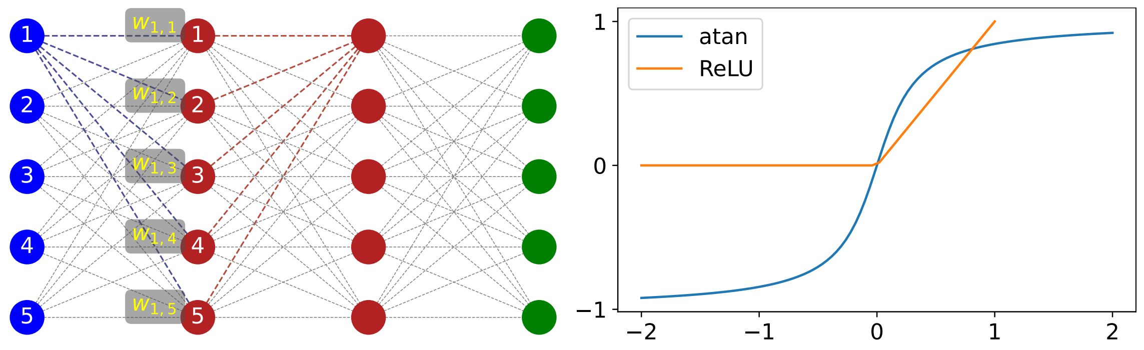


Figure 2.27: Left: Fully connected layers **Right:** A sigmoid and a rectified linear function, two widely used activation functions in machine learning.

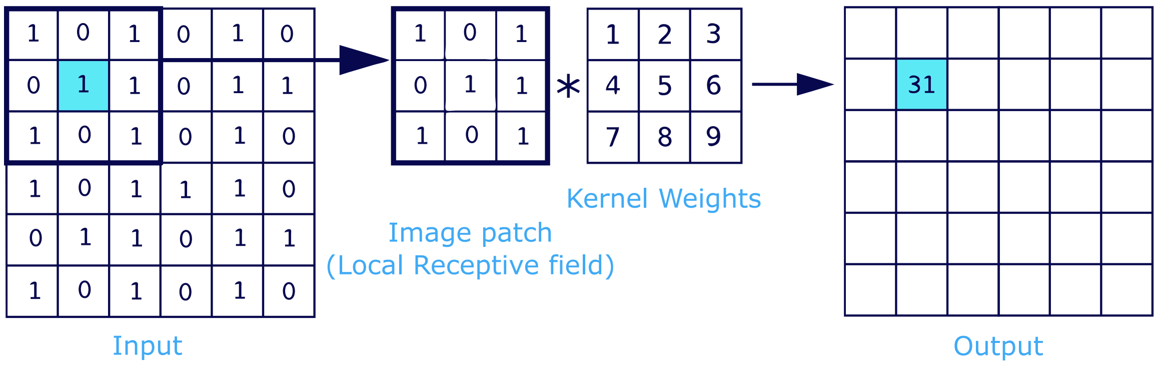


Figure 2.28: Convolution process for a Convolutional Neural Network. The filter weights are the parameters to be found by training. It is common to divide the final output by the sum of the kernel weights to guarantee normalization.

Layers

Neural networks are typically organized in layers, i.e. matrices of neurons. These layers are put next to each others and connected. These connections can differ greatly depending of the type of operations needed. Fully connected layers have each output of the first layer connect to all inputs of the second layer. For an $N \times N$ layer, there are $N \times N \times N$ weights, this can be seen in Figure 2.27. Other types of layers have been developed such as attention layers, normalization layers, and most importantly in our case, convolutional layers and recurrent layers.

Convolutional layers perform a convolution operation using a filter. The filter can be of arbitrary size, the larger the more expensive the operation. The filter represents the recipe for combining the pixels together. It is an $a \times b$ matrix, for a filter of size $(2a+1) \times (2b+1)$, the summation indices range from $-a$ to a and from $-b$ to b.

The convolution of the image I and filter K (this notation comes from the word Kernel):

$$(I * K)(x, y) = \sum_{i=-a}^{a} \sum_{j=-b}^{b} I(x+i, y+j) \cdot K(i, j) \tag{2.47}$$

This filter is applied on each pixel of the image, and the filter components corresponds to the fraction of each pixel that will be included in the result.

The filter only uses the local neighbourhood of each pixel to determine the output value, as can be seen in Figure 2.28. The learning process in this case consists in learning the values of the filter. This convolution operation has a number of benefits. It has a low amount of parameters but can generalize well. It is also agnostic to the position of the features contrary to fully connected layers. It guarantees to treat the entire image with the same operations.

Convolutional Neural Networks (CNNs) are widely used for image processing, in areas such as segmentation, noise removal, visual search, image recognition and much more. In Chapter 5 CNNs are the backbone of our method for temporal anti-aliasing.

3 Hybrid online autotuning

In this chapter, we discuss a method for improving performance in a raytracing context by mitigating the drawbacks of an existing technique: autotuning. Autotuning is a live optimization scheme that allows for efficient exploration of the search space. Our new method allows storing, reusing and even interpolating the existing data, spending less time searching and allowing more time on exploiting an already optimal solution. Finding the intersection between ray and scene is the very core of the ray tracing. This operation must be repeated billions of times to produce even a single frame, especially for path tracing, the current standard for production renderers [Chr+18; Bur+18; Fas+18]. Acceleration structures reduce the complexity of the operation from $O(n)$ to $O(log(n))$ in the number of primitives to intersect. However this complexity is still compounded by the immense number of rays to cast, which makes even the slightest gain worthwhile. Current state-of-the-art acceleration structure such as Embree, use BVHs with different construction strategies. These structures and strategies all come with parameters that should be adjusted to provide noticeable performance improvements. Acceleration structures are specific to each scene and their optimal parameter values may change from scene to scene. Autotuning proved to be a good way to find good parameters, as Tillmann et al. [Til+16] showed. One drawback is that this method will always have to search, have a "warm up time" before a good configuration is reached. Autotuning each scene may not be necessary when optimal parameters for similar scenes are known.

To recognize similar scenes, we developed indicators, computed from the scene data. These indicators serve as input to a machine learning model that outputs a set of acceleration structure parameters. This machine learning model is trained using a collection of different scenes and viewpoints. This process allows for near optimal configurations when inputs are similar to known configurations, and efficient autotuning search for totally new inputs. We report an improvement in rendering time, acceleration structure building time, or combined time, depending on the autotuning optimization target, compared to standard configurations. Autotuning on the tested scenes yields configurations that outperforms ones recommended by the literature by up to 11% median. We achieve 95% of the autotuning result while reducing its overhead by 90%.

We then dive deeper in the acceleration structure parameter behavior. The parameter space being too large to be explored at runtime, we explored it offline, leading to more global recommendations for parameter constraints during optimization. We establish a short list of good practices for setting constraints, depending on the optimization target and the type of scene.

This project led to the two publications [Her+19; Her+21]. This chapter is an extended version of these articles, with additional data and insight that could not be included due to paper length limitations. [1] In this chapter, we first detail the different components for online autotuning, namely our renderer and our autotuner.

We then perform a parameter exploration, and show the acceleration structure parameter's influence on the building time of the acceleration structure, and the rendering time. We show the benefits of using an autotuning setup to improve performance automatically and during runtime. We then develop indicators for our machine learning model which will anticipate which parameters are optimal. Finally, we test our models against baseline autotuning and default parameter configuration.

3.1 Autotuning Parallel Raytracing

In this section we will present our rendering environment, most notably its acceleration structure, and their integration within the autotuning framework. Acceleration structure building methods are very diverse, one can choose between partitioning space or partitioning objects, with kD-trees and BVHs as their respective notable family of methods. Their use cases vary but in many cases a BVH will be the default choice. Within the family of Bounding Volume Hierarchies, a large variety of methods have been developed. Newer methods now focus on iterative strategies to offload building cost over a longer period but with a quicker time to first ray. This variety brings with it a large amount of parameters, most of which have non-trivial values to determine.

Acceleration structures are ultimately what makes rendering possible at all for complex scenes. Sitting on top of it is the renderer itself. We now detail the renderer we use for our experiments.

3.1.1 Renderer

Ray tracing [Kaj86] and especially path tracing are now a standard of the movie industry [Chr+18; Bur+18]. It is increasingly used for interactive applications [NHD10] to help rasterization with global illumination [TO12]. Interactive ray tracing is also available on CPUs thanks notably to OSPray [Wal+17], an Intel open source ray tracing engine. In real-time applications, it is common to use one sample per pixel (spp) and then process the image with denoising techniques [Sch+17; SPD18]. As denoising techniques are not the focus of this dissertation, we will present the images as they are out of the renderer.

We implemented a Monte Carlo progressive path tracer in C++. The renderer is fully accessible from Python which is how we orchestrated the experiments. A renderer is called progressive when it performs a relatively small number of iterations before returning a noisy image. The combination of enough of these noisy images produces a more converged, noise-free image. This is very convenient to record progress and statistics on every step of the image generation. This is essentially what allows our autotuner to perform parameter modification.

[1] This expanded version features additional details on the autotuning process and management of the experiments. Other additions include figures and discussions relative to the `sahBlockSize` parameter in Section 3.2.1.

The renderer we built supports Multiple Importance Sampling [Vea98] for area lights for improved convergence. We also support dielectrics, specular, diffuse, and glossy materials. The diffuse materials are implemented with the Oren-Nayar model [ON94]. Glossy materials are implemented using the anisotropic GGX distribution [Wal+07]. These are typical, physically based BSDF models that are standard in most research renderers. To stay within real-time constraints, we limit the path length to three bounces. A shorter path length is detrimental to the quality of the measurements since it overemphasizes the importance of primary rays. In rendering, primary rays are coherent but subsequent rays are not. Coherence represents how nearby rays will spread out in the scene. Coherent rays result in similar accesses and traversal orders of the acceleration structures, which can be taken advantage of. Notably, with coherent rays, the cache utilization is much higher, and even some other technique can optimize and treat ray bundles, possibly using certain vectorization operations not possible on incoherent rays.

The literature proposed solutions to optimize acceleration structures based on known heuristics, like hard shadows directions and primary rays origin [HM08]. While it is interesting to evaluate how autotuners could optimize specifically for coherent rays, we target use cases of global illumination with path tracing. We require enough bounces for good quality global illumination, but few enough to represent real-time use cases.

Our renderer uses pixel-wise parallelism for scalability, implemented with OpenMP.

3.1.2 Tuning Parameters

In this section we discuss the choice of the acceleration structure. We further detail its parameters.

We adopt Embree [Wal+14] as our acceleration structure building and tracing framework. Embree is Intel's open source implementation of state-of-the-art BVHs, and it is representative of optimized BVH implementations. Embree targets high performance and provides a large set of parameters to tailor the BVH to any need. It includes algorithms for low, medium, and high quality BVHs. Embree exposes eleven tunable parameters which we summarize in Table 3.1. Optimization requires at minimum the valid range of each parameters. Further limiting the range allows for faster tuning convergence but the optimum might be outside of the limited range. We tackle this issue in more details in Section 3.2. These parameters will affect the construction of the BVH, and they are interdependent. For instance, reducing the number of triangles per leaf (`min/max leaf size`) is linked to the maximum depth of the tree (`maxDepth`). The tree will deepen significantly more if the leaf size is small. The `quality` parameter selects the subdivision algorithm. Its 3 values are respectively: Morton codes, binned SAH or binned SAH with primitive splitting. Binned SAH with primitive splitting computes whether it is advantageous to split certain triangles in the scene, allowing for tighter bounding boxes and a better SAH value. When `quality=1` (Morton codes) the `sahBlockSize`, `travcost` and `intcost` parameters are ignored. Indeed they refer to the costs in the SAH computation (cf: Section 2.2.4). They are used to discriminate between deepening the tree (lower traversal cost) or leaving it as it is when the minimum/maximum leaf size allows it (intersecting primitives in the node). Since Morton Codes do not rely on SAH computation, they are simply ignored. The node branching factor (`branchingFactor`) has 2, 4 and 8 as possible values. This parameter allows for its value as the number of node children. Indeed with vectorized instructions, it can be advantageous to have more than 2 node children to process them in parallel. This makes `branchingFactor` a hardware dependent parameter. These eleven parameters will be modified by the tuner to optimize a metric based on rendering and building time. Embree supplies a recommended configuration for these parameters. For our analysis it is important to have a starting configuration. As we want to compare to expert knowledge, we use the recommended values of Embree.

Table 3.1: Embree parameters

Name	Default Value	Description
quality	2	1: Morton codes 2: binned SAH 3: primitive split
branchingFactor	2	Max # of child nodes
maxDepth	32	Max depth of BVH tree
sahBlockSize	1	Optimization AVX/SSE
minLeafSize	1	Min # triangles per leaf
maxLeafSize	32	Max # triangles per leaf
travcost	1	Cost node traversal
intcost	1	Cost triangle intersect°
static	false	Optimize for static scene
compact	false	Optimize memory usage
robust	false	Robust intersection algorithms

3.1.3 The Autotuner

Our autotuner uses the Nelder-Mead [Cha12] algorithm for optimization. We collaborated with the developers of the autotuner to create an intermodal optimization scheme. This means the autotuner can optimize floating point values, boolean and nominal (integer) values at the same time. This is performed by keeping the Nelder Mead optimization progress and translating it to a new nominal value, remembering the state it left. When continuing to optimize with the first value, it can resume the previous optimization where it left it, essentially having several concurrent optimizations.

We created a Python wrapper around the used features to allow for improved automation using *pybind*. We pass the parameters to the autotuner with JSON format. The parameters can be seen in Table 3.1.

We tested different initialization methods for the parameters. We tried uniform distribution, centered normal distribution, and Latin Hypercube. The Latin Hypercube sampling [MBC79] turned out to be the most consistent initialization for our use case.

3.1.4 Tuner integration

The autotuner is quite easily integrated into any iterative process whose parameters need optimization.

```
{'parameters': [
  { 'name': 'maxBranchingFactor',
    'possibleValues': [2, 4, 8],
    'type': 'NOM',
    'value': 4},
  { 'name': 'dynamic',
    'possibleValues': [False, True],
    'type': 'NOM',
    'value': True},
  { 'name': 'sahBlockSize',
    'value': 2,
    'type': 'INT',
    'max': 32,
    'min': 1},
  { 'name': 'intcost',
    'value': 2.35
    'type': 'FLOAT',
    'max': 10.0,
    'min': 0.1}
]}
```

Figure 3.1: Example of the datastructure used to communicate with the autotuner. This structure shows an example of each type of parameter: INT for integers, FLOAT for floating point numbers, and NOM for nominal values. We show 2 different uses of Nominal values.

Figure 3.1 provides how to select the parameters as appropriate, and Figure 3.2 shows the few commands required for the autotuning process.

In the rendering loop, one only needs the current parameter values of the autotuner, for them to be used by the builder and renderer. Then one measures the different metrics required such as building time, rendering time,... to be processed by the tuner. This action triggers the optimizer to update the current parameter values for the next iteration. Note that for the first iteration, the "current" parameter values are the ones supplied to the autotuner at initialization.

3.1.5 Autotuning in action

Once the autotuner is connected to the renderer properly, we can obtain a graph such as the one in Figure 3.3 which we analyze in the following.

The solid horizontal line is what we compare to, an already decent configuration, but not optimal. The other lines correspond to different tuning targets. In one case we optimize for the rendering time only -the green line-, not considering building time. In the other case, we optimized for total time -the red line-, being acceleration structure building time added to the rendering time. The lines jump around as the optimizer tries to find good configurations, and then settles into the optimum it found. It takes around 30 iterations to converge.

The green line dipping well below the solid line on the leftmost graph indicates a noticeable improvement in rendering time. In this case, about 15%. This comes at the cost of total time of course, it is well known that spending more time building an acceleration structure, potentially lowering the SAH criterion further, improves on the rendering speed. However, the red line shows that we can do better on both sides. The left graph shows a

Figure 3.2: Pseudo code for the integration of the autotuner in the rendering process. One can easily keep only the best configuration so far as the best configuration, in case the algorithm explores a bad configuration at the last iteration

Input:

- TP: Array of Tuning Parameters [min, max, starting value]
- PV: Array of current Parameter Values
- N_iterations: Number of times the autotuner will change parameters

Result: Parameter configuration found by the autotuner after N_iterations

```
Tuner = Autotuner(TP) // Creates the Autotuner object
for $i \leftarrow 0$ to $N\_iterations$ do
    PV ← Tuner.current_PV
    t_build ← BuildBVH (PV) // measure time
    t_rendering ← Rendering (PV)
    // Choose the feedback function
    current_feedback ← t_build + t_rendering // or just ← t_rendering
    Tuner.feedback ← current_feedback
end
return Tuner.current_PV
```

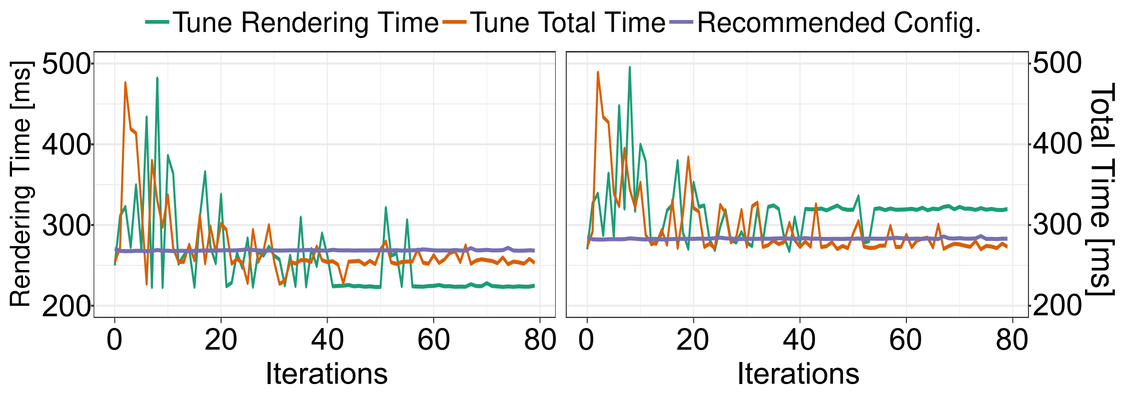


Figure 3.3: Example of autotuning improving performance on the *sponza* scene.

better rendering time is achieved, although not as improved as on the green line, and at the same time, a better total time is achieved.

These two different lines can represent scenarios where one has to rebuild the acceleration structure every frame, and one where it does not need rebuilding. Note that for scenes that require frequent updates, it is common to use iterative approaches that usually allow for partial update without a full rebuild, maintaining or improving performance on static parts of the scene, while allowing insertion of new objects or the movement of existing ones with minimal overhead.

We have described how the autotuner functions. We will now discuss about the parameters that are to be modified by the autotuner. These parameters are specific to each use case. We will provide a detailed example using the Embree acceleration structure parameters. This can serve as a reference for the analysis of any other autotuning use case for improving and understanding the expected reliability of the autotuning.

3.2 Parameter analysis

The parameters of the acceleration structure have an impact on its behavior and its performance. This section is dedicated to an in-depth analysis of the behavior of each BVH parameter. This also serves as a guide in case this study has to be replicated for another datastructure. Indeed, we provide a method for the investigation of relevant parameters.

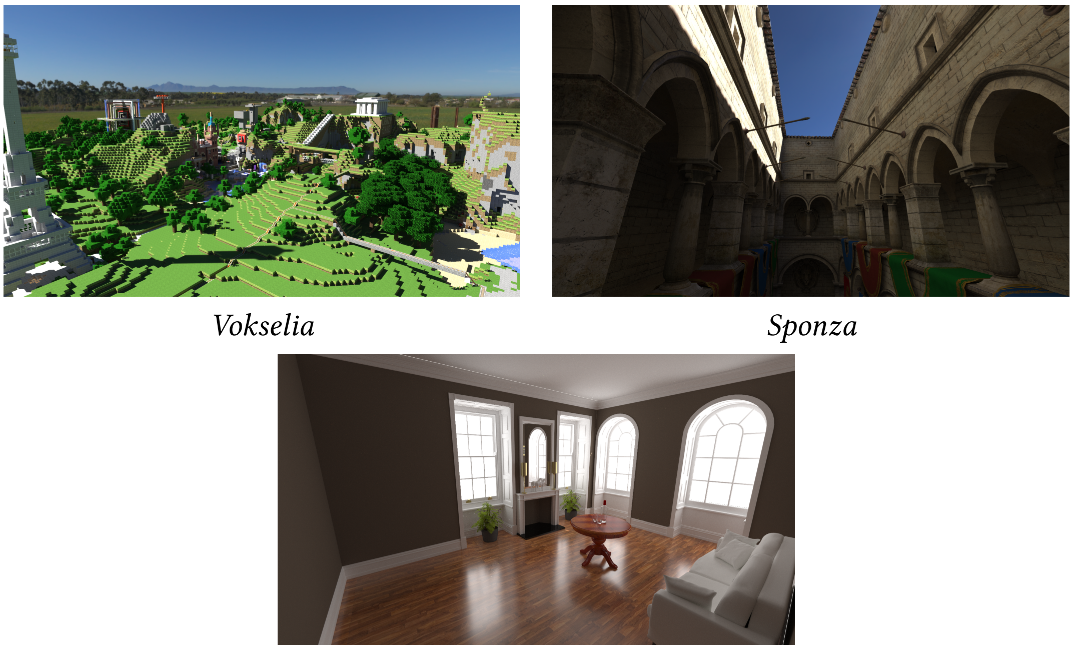
To test any given configuration of parameter values, we need a scene. We chose 3 different scenes: *vokselia*, *sponza*, and *fireplace_room*. These are three very different scenes. *vokselia* is a large, open map, entirely voxelized (scene from the Minecraft game). *sponza* is well known for its complex lighting being semi-outdoors, its somewhat elaborate geometry. Finally, *fireplace_room* is a fully indoor scene, presenting difficult light transport challenges, notably due to the variety of materials, being glossy, diffuse, specular with a mirror and a glass bottle, and a difficult combination of a metallic glossy material viewed on a mirror. These scene can be seen in Figure 3.4

3.2.1 Correlations

First we need to figure out which parameters are relevant at all, as in, which ones influence rendering and/or acceleration structure building time. For this purpose, we used $800,000$ (eight hundred thousand) random configurations of BVH parameters per scene. The gathered data contains all information related to timings, scene data and metrics, and BVH parameters.

As a first insight, we map out the correlations in the dataset. After computing it on seven different scenes, we noticed that the correlation magnitudes were different (up to 10 points) but the pattern remained the same. For this reason, and for conciseness, we only show them for *fireplace_room* and *sponza* in Figure 3.5.

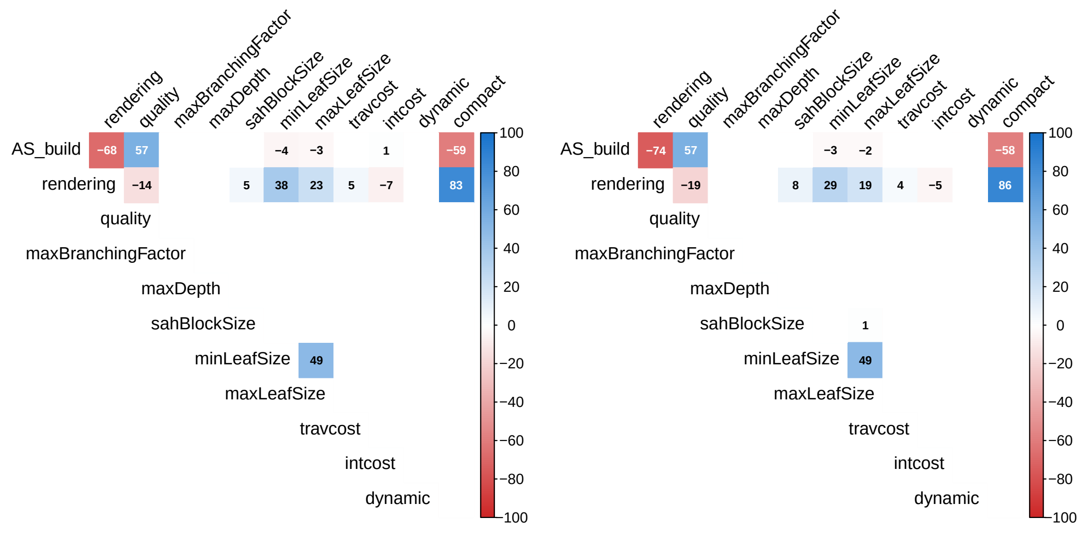
This correlation matrix represents the link between BVH parameters and timing. This matrix shows how the change in one parameter affects the change in other parameters. there is not enough information yet for a causal link, but it is interesting enough to warrant further investigation.



Vokselia *Sponza*

Fireplace room

Figure 3.4: The three scenes used in the parameter exploration.



Fireplace room *Sponza*

Figure 3.5: Correlation matrices. Blank boxes do not reach the 0.01 P-value criteria. `quality`, `compact`, `minLeafSize` show high correlation. `sahBlockSize` and `intcost`/`travcost` present small correlation but are well above the P-value.

We used random sampling of configurations of parameter values to create this matrix. This means there should be no correlation between parameters. The only correlation between parameters is with `minLeafSize` and `maxLeafSize`, due to the necessary condition `minLeafSize` $<$ `maxLeafSize`.

Because we use random sampling, all parameters show no correlation with other parameters, but they can show correlation with rendering time or build time. First we address the larger values of correlation. We start with the `compact` parameter. It has a large correlation value with rendering time, and anti correlates with the building time. We noticed that `compact` $= 1$ entirely disables all other parameters. A separate measurement showed about 30% decreased performance overall. This parameter is only useful to prevent a memory swapping scenario. Because all scenes fit in memory, and cases where the acceleration structure are a limiting memory constraint are rare in this analysis, the following analysis is conducted assuming `compact` disabled.

Parameter: Quality

We first note a very large impact of the `quality` parameter with 57% correlation with the build time and -19% for rendering time. These values are explained by a radical change in the algorithms used for building the structures. It impacts acceleration structure building time rendering time in both value and variance. Triangle splitting creates a better datastructure, improving rendering time at the cost of build time. The efficiency of this feature depends on the application-specific rendering load. To illustrate this, we compare two common resolutions on the *sponza* scene. The difference of rendering load between 480p and 720p is 432000 pixels. The total time speedup achieved by choosing to split triangles is -5.8% for 480p and $+8\%$ for 720p. `quality` is obviously a key parameter.

Parameter: Minimum leaf size, Maximum leaf size

These two parameters constrain the number of primitives per leaves in the acceleration structure tree. It impacts mostly rendering time. It is important to note that this parameter is no longer active when `quality=2` (triangle splitting). Figure 3.6 shows the distribution of average rendering time and average build time, with `minLeafSize` and `maxLeafSize` in the horizontal and vertical axis. The relationship `minLeafSize` $<$ `maxLeafSize` makes the graph triangle shaped. Acceleration structure build time and rendering time present a clear trade-off: build time is improved with larger leaves while rendering time is faster with smaller leaves. The optimal point will necessarily be input dependent. We include in the appendix the extension of this analysis on all scenes, left out here for conciseness. We explain in the following section the interplay between these two parameters and the `sahBlockSize` parameter.

Parameter: SAH Block size

The parameter `sahBlockSize` is a parameter in the SAH computation. It refers to the number of primitives that can be intersected at once. For example, given SSE4.2, one can intersect four primitives at once. Given eight primitives, and the optimal SAH cut would be six and two, the `sahBlockSize` intervenes to warn that this would cause two

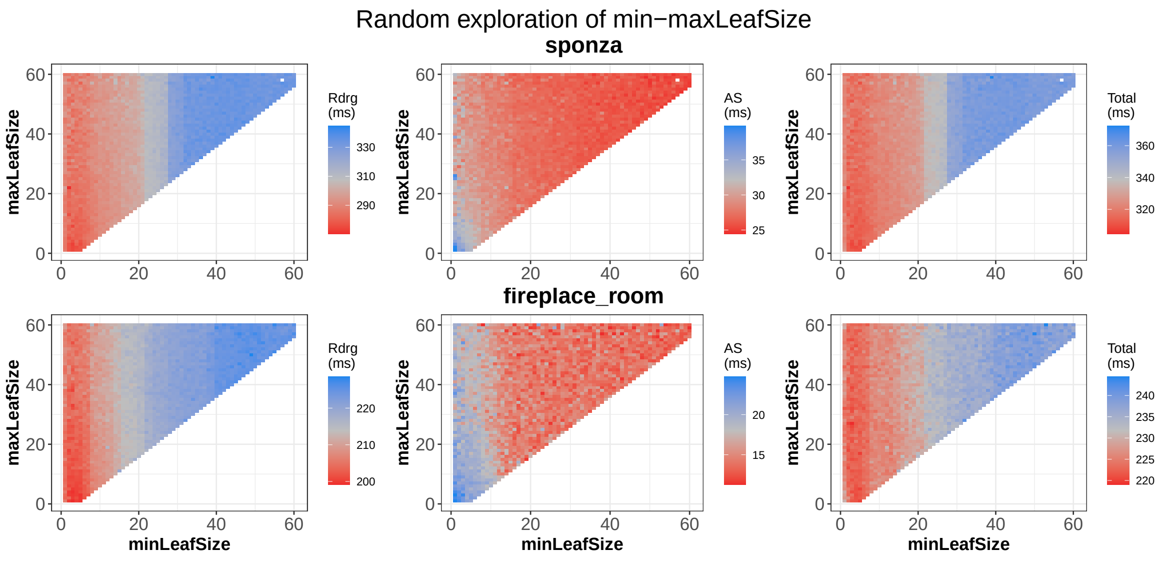
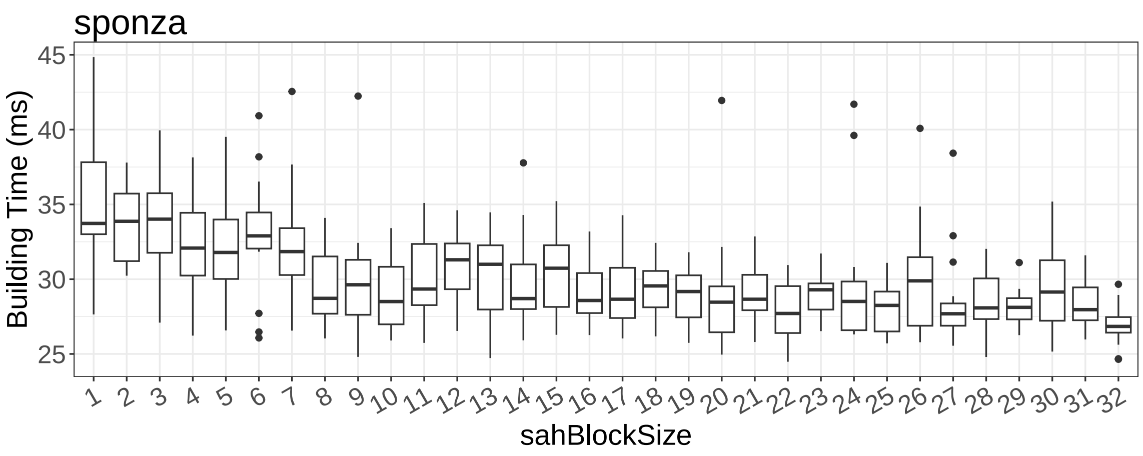


Figure 3.6: Min Max Leaf Size. Performance distribution given `quality` = 1. Similar trend is observed among all scenes: small leaf size is preferable for rendering, big leaf size for AS build time. Left, center and right columns correspond respectively to Rendering time, Acceleration structure buiding time and Total time.

intersection calls on one side and one on the other whereas a four/four split would only incur one intersection call everytime. This is decoupled from the `minLeafSize` parameter as it allows cuts to have an equal number of primives on each side that can be a multiple of the SIMD size. Instead of using hardware values, Embree allows it to be a parameter for building acceleration structures for other hardware having different SIMD size (dedicated intersection machines or GPUs).

We wanted to gain a more practical insight on this parameter: surely its best performance is given by the SIMD level of the machine hardware. We tested this assumption by selecting all available values of `sahBlockSize` and checked that the leaf size values were chosen accordingly. We obtained mixed results, as seen on the Figure 3.7, even after controlling for `quality`, `compact`, `min/maxLeafize`. There is an effect, and one can discern steps on the graph every power of two. We thought this effect was more significant. To verify this, we used our autotuner to only probe `sahBlocksize`, keeping all other parameter from interfering. The graph obtained is much clearer, and the steps clearly appear, which is the expected behavior. We explain in more details in the next section how we obtained this data.

The `sahBlockSize` parameter does improve rendering time at the hardware value, but worsens building time at lower values, as shown in Figure 3.8 for the *sponza* scene. We see clearly a step pattern in the Build time graph (top). This is due to the internal computation of a $ceil(\frac{N_{primitives}}{sahBlockSize})$. This computes the number of intersection tests needed for a given number of primitives in a branch. On the other hand, in the rendering time graph (bottom), the optimum is indeed found between 4 and 7 included. This is coherent with the SIMD of our testing machine. For *sponza*, the result is quite clear cut, but this is not so obvious for other scenes. For instance *vokselia*'s optimal value is not at the SIMD value as shown in



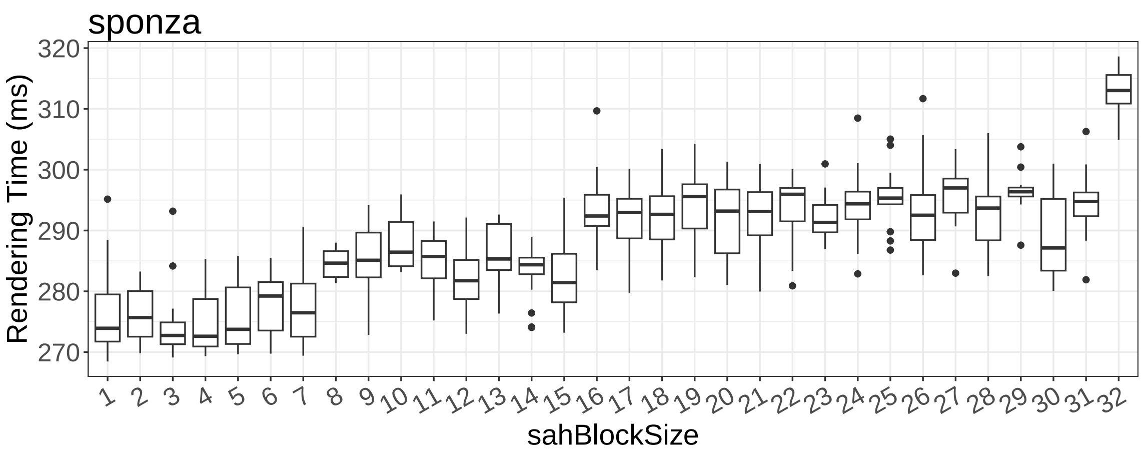


Figure 3.7: sahBlockSize impacts performance but the data is quite noisy. The spread is due to sampling a variety of configurations, more or less optimal. We perform another experiment to evaluate the performance of this parameter further below.

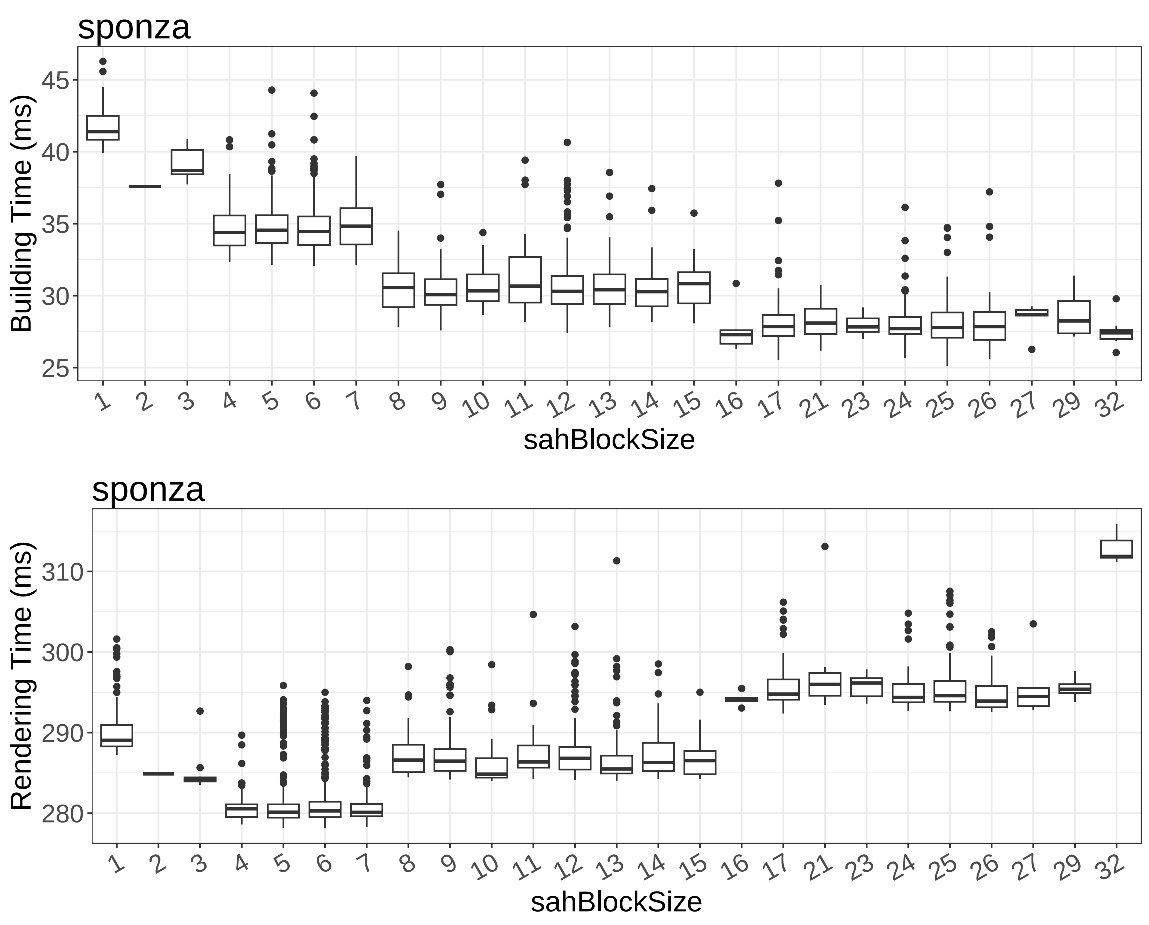


Figure 3.8: sahBlockSize's effect on performance when using an autotuner. This is the aggregation of multiple tuning targets, rendering time and total time, and many different tuning runs.

Figure 3.9. In this scene the rendering time is only barely affected by the `sahBlockSize`. According to our experiments, if the rendering workload was about 8 times bigger, the benefits of using the SIMD value would become visible.

Parameters: Intersection cost and Traversal cost

The parameters `intcost`/`travcost` appear in the SAH computation when building the tree. These parameters, respectively intersection cost and traversal cost, allow to tilt the tradeoff between adding a new node to traverse or keeping it as it is. This value can be theoretically derived from the number of instructions necessary in each case. We noted that some values still perform better than others when the autotuner optimizes them. Their correlation coefficient is quite small but it is well above the p-value. We will explore them in more details in the following section.

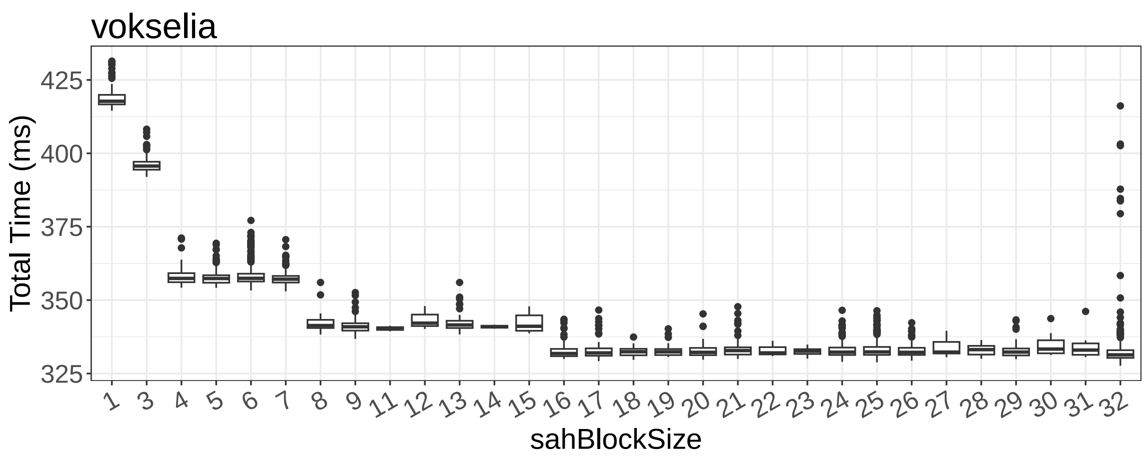


Figure 3.9: sahBlockSize's effect on the total time. The optimal performance is not attained at the corresponding hardware SIMD value for the *vokselia* scene. There is no data for points 2 and 10.

3.2.2 Partial Tuning

Parameters can influence each other and it can be difficult to grasp with random sampling, even using many samples. We will focus on the parameters previously identified:

- `minLeafSize` and `maxLeafSize`
- `intcost` and `travcost`
- `intcost`, `travcost` and `sahBlockSize`

We selected the parameters studied in the previous section that have the biggest impact on performance, and high degree of interdependency. We use our autotuner to optimize only the chosen parameters, leaving the others with default. We ran many separate optimization runs, with different starting points, and recorded every configuration along the way.

Tuning only the `minLeafSize` and `maxLeafSize` parameters improve performance up to 8% compared to the recommended configuration. Figure 3.10 shows the performance distribution as previously except that configurations are not random, but given by the tuner as it converges. For the *sponza* scene, the optimum leaf size is 4 to 7, while between 25 and 30 is best for *vokselia*. It is scene dependent as expected. Moreover, these results confirm that the tuner converges indeed toward the expected optimum values inferred from Figure 3.6.

The SAH metric uses both `intcost/travcost`. Figure 3.11 presents the exploration of the parameter space by the autotuner. High traversal cost and small intersection cost seem to be favored. A high traversal cost makes the builder create shallow trees, which improves AS build performance but decreases rendering performance. The visible presence of diagonals shows the path of the tuner converging. Here, `sahBlockSize` is still set to 1. We will now see how this parameter changes the distribution.

The modification of the vectorization parameter influence modifies the optimum of some other parameters, mostly the SAH costs. We study the triplet of parameters `intcost`, `travcost`, `sahBlockSize`. Figure 3.12 represents data the same way as the previous plots,

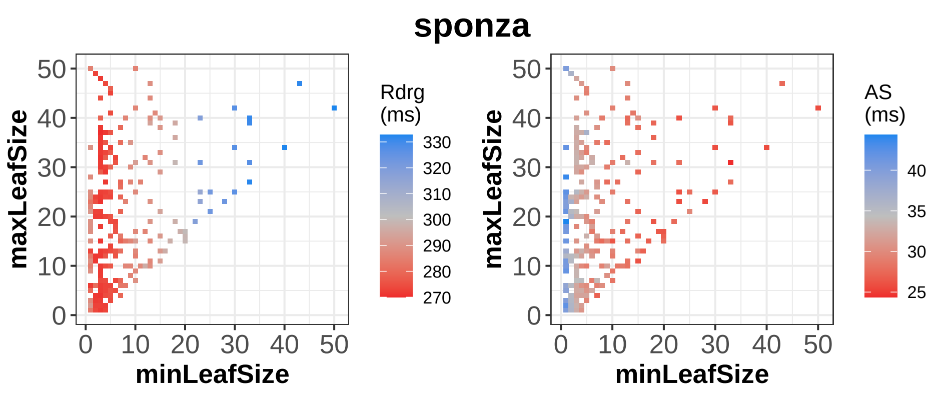


Figure 3.10: Performance distribution found by tuning `minLeafSize`, `maxLeafSize`. Small to medium scenes prefer small leafSize and *vokselia*'s optimum is 25 to 30 as expected from Figure 3.6.

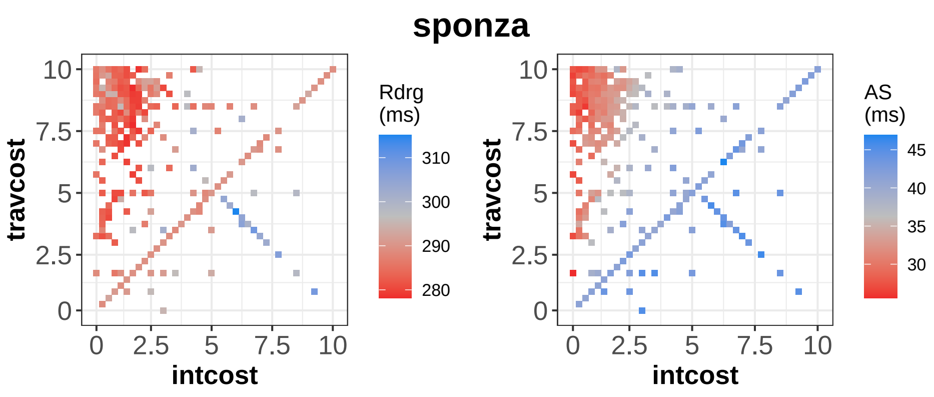


Figure 3.11: Performance distribution found by tuning `intcost`, `travcost`. Optima are bundled around the top left corner: Low `intcost` and high `travcost` make the tree structure shallower when no other parameter controls the tree building.

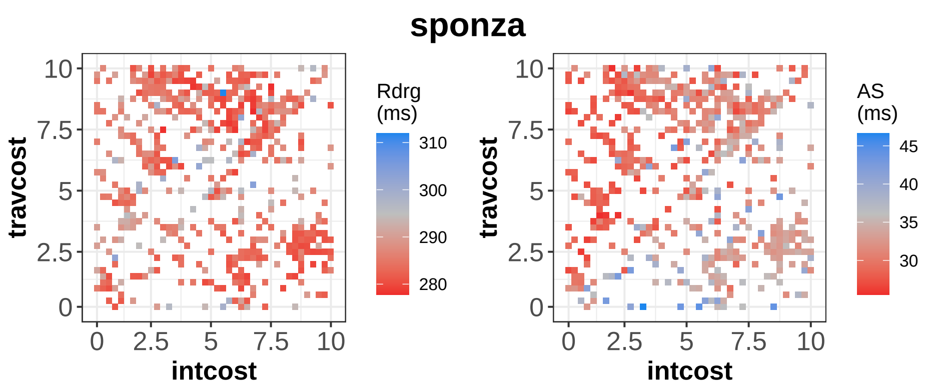


Figure 3.12: Performance distribution found by tuning `intcost`, `travcost` and `sahBlockSize`. `sahBlockSize` is tuned at the same time but is not shown here. Optima are scattered significantly more than in Figure 3.11, but still form clusters. Adding `sahBlockSize` to the equation considerably changes the distribution, a sign of interaction between these parameters.

but here `sahBlockSize` is tuned as well. `sahBlockSize` is not represented because its converged value is the same as when tuned alone, regardless of the other two parameters. Each scene presents different clusters of convergence. The density of samples in the cluster regions is very high, indicating a consistent optimum. The scene *vokselia* shows more similarities with the previous graph than the other scenes. It seems more efficient for this scene to still construct a shallow tree. The construction of shallow trees (with leaves containing many primitives) is allowed by the default values of Embree: `minLeafSize` $= 1$ and `maxLeafSize` $= 32$. If these two values can change as well, the resulting performance and convergence is expected to change.

Up until now, the scenes were tested as they were. Adding difficult geometry challenges the builder and provide insights on dynamic effects and limits of the acceleration structure and of its building algorithm.

3.2.3 Build time stability

The stability and reproducibility seemingly provided by triangle splitting will be put in perspective in this section. To expose the behavior of the builder with complicated cases, we use a rotating cylinder containing long and thin triangles, intersecting the geometry of the scene. The cylinder is half as long as the diagonal of the scene. These triangles will put the builder under heavy load, allowing us to see how it handles hard cases. Figure 3.13 shows the performance during the rotation of the cylinder. The confidence interval at 99.99% is displayed as a greyed area around the average sampled value.

All scenes show improved rendering performance with splitting, but a significantly worse build performance: more than doubled for all scenes. The variability of the rendering time — or the vertical spread — is also very high as shown particularly clearly with `sponza` where rendering time spans almost $175\,\mathrm{ms}$ which is 50% of variation around the average value.

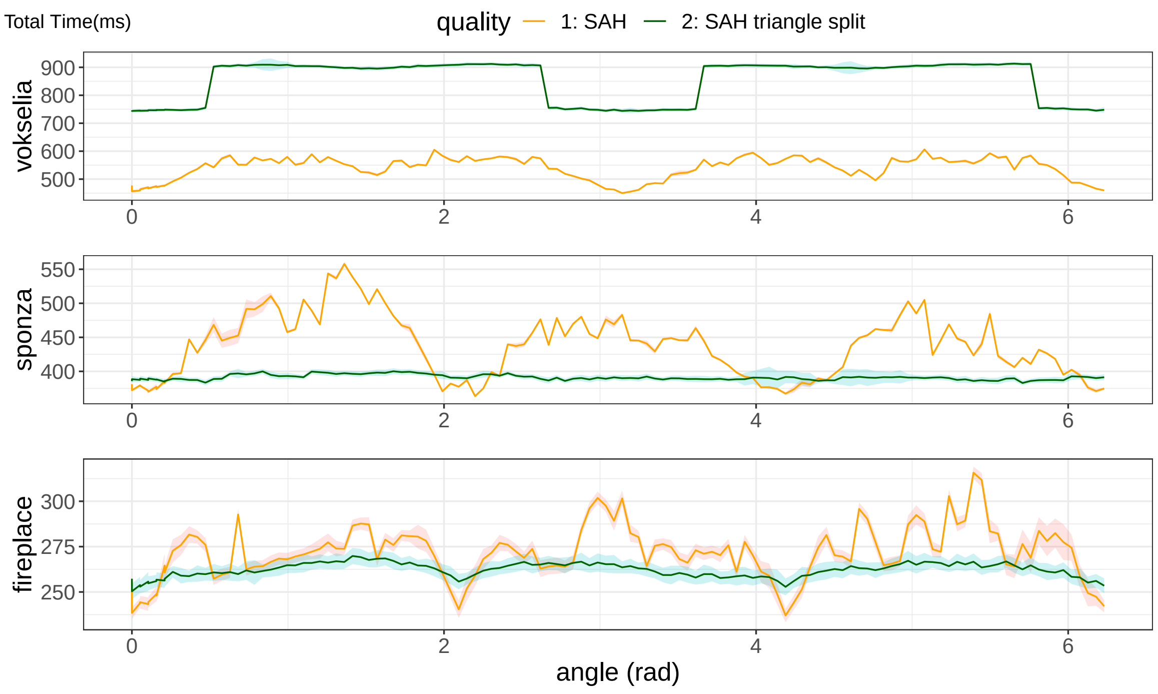


Figure 3.13: Evolution of performance depending on triangle splitting along a rotation of the cylinder. Each graph is one of three scenarios: when splitting is *not* worth it: `vokselia`, when it is: `sponza`, and when it is about the same: `fireplace_room`

`vokselia`'s build time in Figure 3.13 takes a 26% jump: from $560\,\text{ms}$ to $710\,\text{ms}$ with triangle splitting. At that moment, the cylinder starts intersecting a significant portion of the underground geometry, causing a large amount of triangle splits.

3.2.4 Observations

The analysis exposes certain behaviors of acceleration structures in general. This section summarizes the findings for different use cases.

When only rendering time is important, triangle splitting offers good performance and stability (choosing `quality` $= 2$).

When the build time is also important, disabling the triangle splitting is a significantly more versatile setting. It allows for fine adjustments on different trade-offs, more specifically, the minimum leaf size tends to reduce rendering time; $4 - 10$ seem optimal in our test cases. To reduce build time, `minLeafSize` can be increased. A value between 20 to 30 offers a good reduction in build time. Triangle splitting is tempting but is to be used only if the input is carefully controlled, without long, thin, or intersecting triangles.

Unsurprisingly, `sahBlockSize` will perform best for rendering time at the corresponding hardware value. It is 2 for SSE, 4 for AVX2, 8 for AVX512, 32 for GPUs (warp size). Taking larger values will increase rendering time and decrease build time.

Thorough analysis of scenes is of course not common, and dynamic scenes require even more care. In the next section, we expose how we create models able to handle dynamic context, while ensuring near optimal performance.

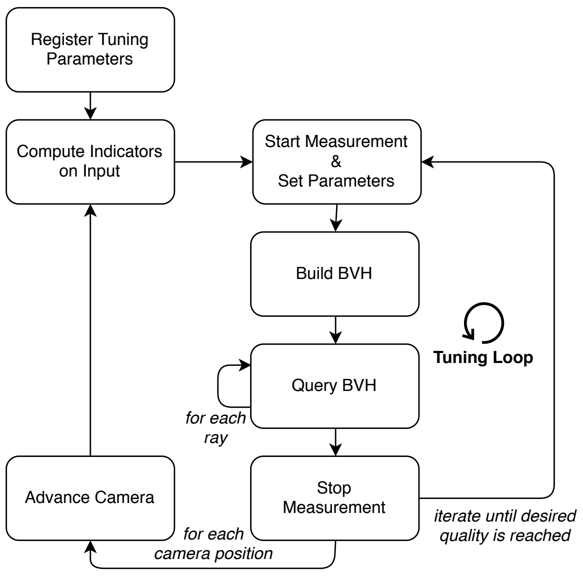

Register Tuning Parameters
Compute Indicators on Input
Start Measurement & Set Parameters
Build BVH
Tuning Loop
Query BVH
for each ray
Advance Camera
Stop Measurement
iterate until desired quality is reached
for each camera position

Figure 3.14: Autotuning loop. Integration of the tuner in the ray tracing workflow. The Query BVH box corresponds to the rendering process in our case, although it could be queried for something else than rendering in theory.

3.3 Hybrid Autotuning

One of the main obstacles to practical deployments of machine-learning or model-based methods is training. Pure offline techniques require massive amounts of samples a-priori to build the models. Most of the quality of the models depends on the representativeness of the samples. Online approaches, on the other hand, learn from new data that is generated in the deployment context. The main question becomes how to construct the initial model. Starting with an imprecise initial model means sub-optimal predictions and decreased performance. While offline training is certainly viable to seed an online approach, it still requires input samples, which can be hard to obtain in particular given the heterogeneity of 3D scenes. As a compromise, we propose *hybrid online autotuning*, combining classical search-based tuning with model-based prediction. The high-level tuning process is shown in Figure 3.14. The tuner observes the program and the system state through the indicators. They serve as an approximation of the program and system state. We therefore use the term "state" to refer to the current indicator values. For every change in state, the hybrid tuner chooses to either exploit the model or to explore the configuration space. In either mode, performance feedback from the application for every sampled configuration is used to update the model. Observed states, sampled configurations, and the performance feedback can be stored in the tuning database to avoid sampling the same state-configuration pair twice. Choosing between exploration to gather new data and exploitation of known information is a central operation in Reinforcement Learning (RL) [SB18]. The decision process is implemented through the ϵ-Greedy algorithm. The ϵ-Greedy algorithm is remarkably simple: With a probability of ϵ, choose to explore, otherwise greedily exploit.

A key difference between exploring and exploiting is that exploitation is a one-step process, without any iterations. For a new state the autotuner produces exactly one configuration which once applied is kept until the next state change. Exploring on the other hand can sample multiple configurations per state. That contrast is important because for ray tracing, updating the configuration incurs cost: Changing the BVH parameters requires rebuilding the data structure, which is an expensive operation. For exploration, we use a variant of the Nelder-Mead simplex algorithm [NM65]. Our version is similar to that of Chang [Cha12], using Latin Hypercube sampling [MBC79] as an initialization method.

We investigate two different types of models for the exploitation in this work. The first one is a naive tabular approach and the second is based on a generalized reinforcement learning method. Both approaches are described in detail after a discussion about the state itself.

3.3.1 Indicators

Describing a scene does not get more complex than its raw data, and simple statistics such as number of triangles do not yield sufficient insight in the scene's properties. The ideal case would be to have few metrics allowing the model to figure out a reliable mapping between state and high performance configurations.

We created aggregated statistic that inform more meaningfully on the actual content of a scene, while remaining as compact as a few numbers. We compute 17 indicators as listed in Table 3.2 to describe the input scene data.

We roughly estimate the complexity of the scene with the following indicators:

- number of triangles
- number of meshes (triangularly tessellated surfaces)
- number light sources

However, scenes with the same number of triangles can have different optimal configurations. Triangle sizes vary depending on the meshes in the scene, as walls for instance may have large triangles while detailed objects have smaller ones. In addition, triangle sizes can vary within a mesh and BVHs are susceptible to this variation. To take this into account, we compute the variance of the triangle sizes for each mesh and name this set `VarTriSize`. It would be difficult to use an indicator that works per mesh without aggregation as scenes can have many meshes that may not be relevant, so we compute the average and variance of `VarTriSize`. These indicators are named `mean of VarTriSize` and `variance of VarTriSize`. Additionally, we compute the average area per mesh and the total area of the scene. Because scenes can have wildly different coordinate ranges, such as some having coordinates between 0 and 1, and some others reaching 1000, we have a scaling issue. The areas and distances will impact the indicator in a way that is not representative of the uniqueness of the scene, but of the coordinate system used. To correct this scaling issue, we divide any surface-related value by the square of the diagonal of the scene's bounding box. This normalization is not technically accurate but sufficient for our needs as the scaling is made even across all scenes. The bounding box diagonal is also used as an indicator for the scene's extent. The diffuse ratio indicator is a measure of the average specularity of the surfaces using material properties. The diffuse ratio is between

Table 3.2: Indicators. The camera uses the target position to extract the orientation relatively to an up vector $(0, 1, 0)$. `VarTriSize` is a list containing the variance of triangle sizes for each mesh.

Name	Range
Number of meshes	0 to 10^5
Number of triangles	0 to 10^7
Number of lights	0 to 10^5
Camera Position	-10 to 10 (x,y and z)
Target Position	-10 to 10 (x,y and z)
Vertical FoV	0 to 180
Diffuse Ratio	0 to 1
Extent	0 to 10^5
Total area	0 to 10
Area of lights	0 to 10
Area per Mesh	0 to 10
Mean of VarTriSize	0 to 10^{-2}
Variance of VarTriSize	0 to 10^{-2}

zero and one and for transparent or mirror-like objects, it is zero. For all intermediate specularities, the value will be between zero and one. We also added camera parameters: position, orientation, and the vertical field of view to account for visibility.

3.3.2 Model-Based Prediction

Nearest-Neighbor Prediction

The naive way to predict configurations is to record a table of previous states, configurations, and the observed runtime of those configurations. When the application enters a known state, the best-known configuration can be selected from the table.

Although this solution is obvious, it is also obviously limited: The state space is practically unbounded and too large to store let alone to explore it in its entirety. Even if it could be stored, we must assume that most states would not be in the table. Therefore, a successful predictor must be able to service queries for unknown states. To realize such a predictor using the tabular model, we use a nearest neighbor approach: If the current state is unknown, respond with the best configuration for the state that is closest regarding the distance metric (Euclidean in our case) in the state space.

Prediction Through Function Approximation

Even when using nearest-neighbor predictions, achieving high accuracy may require enormous tables, which are expensive to store and query. Fortunately, that problem has been investigated in the past in the field of Reinforcement Learning. Generalized RL methods offer control (i.e., state-sensitive decision making) for large, both discrete and continuous state spaces through function approximation [SB18]. A recent such algorithm

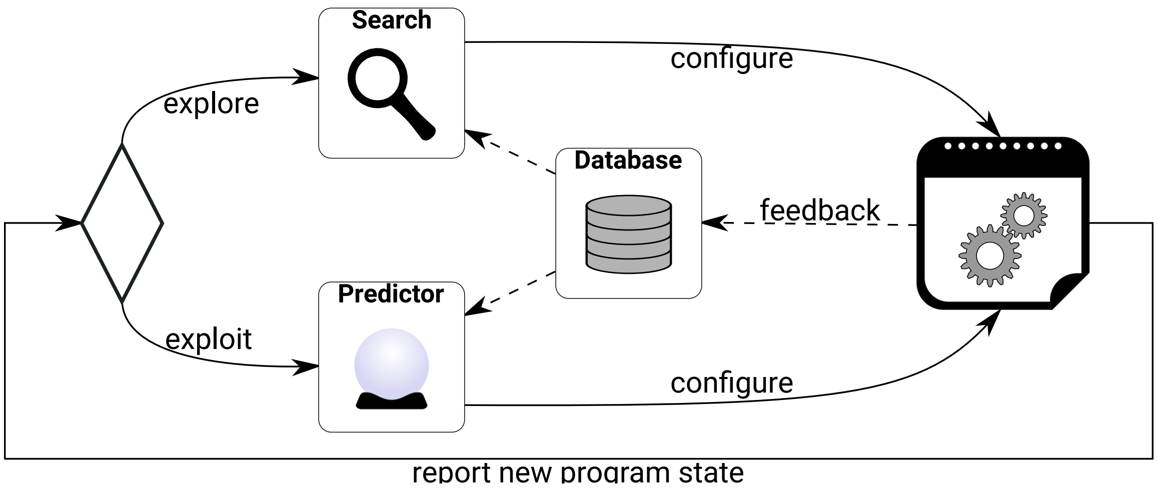


Figure 3.15: The hybrid online tuning workflow.

is Greedy-GQ [Mae+10]. The algorithm is an off-policy gradient-based temporal difference learning method. It is relevant to our use-case because of a set of interesting properties: Off-policy learning enables learning from samples obtained using a different method than the predictor. That allows us to learn from offline data (e.g., from past experiments or tuning searches) but also during online search. Additionally, Greedy-GQ supports incremental online training and imposes no limits on the features we can use to represent the model inputs. Features are high-dimensional functions, in our case Gaussians, representing how parameters and indicators relate with performance. More features mean we can more finely describe this relation, but we also risk overfitting. Both the memory and runtime complexity of the Greedy-GQ algorithm are linear in the number of features. The number of features is both constant and much smaller than the cardinality of the search space. Greedy-GQ is thus an improvement in terms of size over the tabular approach.

At its core, Greedy-GQ manages the linear value function:

$$Q_\theta(s,a) = \boldsymbol{\theta} \cdot \varphi(s,a),$$

where $\varphi(s,a)$ is a vector of real-valued features, and $\boldsymbol{\theta}$ are the coefficients to be learned. This function associates with every state s and "action" a a scalar value. Intuitively, this value estimates the benefit (higher is better) of choosing the action a in state s. An action corresponds to one of the configurations investigated by the tuning search. We thus use the terms interchangeably here. During exploitation, the predictor selects the action that maximizes Q_θ for the current state. When a state-action pair s_t, a_t is evaluated in search mode, the coefficients are updated based on the "reward" R_{t+1} observed by the application, which is the inverse of the runtime measure for the configuration a_t. We use the inverse of the runtime to have a minimisation problem instead of a maximisation one.

The update equations for $\boldsymbol{\theta}$ are controlled by three hyper-parameters: α and β, which are learning rates, and γ, which is a discount factor governing the influence of expected future values. The γ parameter has a particularly interesting semantic in our use case. In the basic reinforcement learning framework, actions taken in a given state influence the possible future states. However, in our application, state changes are unaffected by the parameter configuration we pick. This means that we can set γ to zero, creating an algorithm that is sometimes called a "myopic" [SB18]. The update rule in our implementation of Greedy-GQ thus becomes:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha(R_{t+1} - \boldsymbol{\theta}_t \cdot \varphi(s_t, a_t))\varphi(s_t, a_t).$$

As features we use radial basis functions [KA97]. A radial basis function (RBF) is an N-dimensional Gaussian:

$$\varphi_i(x) = \exp(-\omega_i ||x - c_i||),$$

where $x = (s, a)$ is the concatenation of the state and action vectors. The RBF is centered at c_i with a width determined by ω_i. The hyper-parameters of our RBF model are the number of features and each feature's center and width. Based on preliminary experiments we chose 3.5 times the number of parameters and indicators as feature count, which results in 100 features. The widths are set to a fixed value of 1.

The scales of the indicators vary drastically. Because the distance metric is scale-variant, indicators spanning larger scales will carry more weight in the distance computation than smaller ones. We thus need to normalize parameters and indicators to a comparable scale before passing them to the RBF model. For each of them we estimate their minimum and their maximum value. This is quite easy for the parameters since they are constrained by the design of the acceleration structure. For the indicators, we need to select a meaningful maximum based on data. This preprocess is necessary to compute our features' parameters, c_i in our case.

The largest evaluation scene has 1.8 million triangles, we chose 10 million as our virtual upper bound. Regarding the extent, we chose 100000 and for the number of lights we chose 100000 even though most scenes have only the environment light. The maximum number of meshes was set to 100000. The widest scene has an area of 5.83, so the maximum area was set to 10. For the field of view, it is straightforward to set a maximum of 180 and 1.0 for the diffuse ratio. For the camera positions a maximum of 10000 was used.

Finally, we need to define the centers of the features to form the RBF model. We produce the centers by quasi-randomly placing them in the down-scaled parameter and indicator space. Using Latin Hypercube Sampling again we obtain a space-spanning set of points.

3.3.3 Evaluation

In this section we present the evaluation of our method. We compare hybrid autotuning to classical continuous online-autotuning. We use the purely stochastic ϵ-Greedy policy to choose between exploration and exploitation. Therefore, we are able to evaluate the performance in production from the two behaviors in isolation.

We benchmark on seven scenes with differing complexities and characteristics. The scenes are *sponza*, *gallery*, *vokselia_spawn*, *conference*, *fireplace_room*, *buddha* [Mor17] and *bmw-m6* (which we call *car*) [PJH17]. These scenes have been chosen for their wide variety of geometry. The *vokselia_spawn* scene for instance is a rather large, open world but with a peculiar aspect: triangles are almost all the same size because the world is composed of boxes. The *buddha* scene contains a highly tessellated mesh, but it only occupies a portion of the viewport at a time. The distance to the object is thus particularly important as fewer rays will intersect the object when the distance between camera and object increases. The *car* scene is quite similar to *buddha*, being finely tessellated, but it also features two reflective planes that bounce rays back to the car instead of ending in the environment map. The *gallery*, *conference*, *fireplace_room*, and *sponza* scenes enclose the cameras, so most of the lighting is indirect. Light paths are therefore occluded and rays tend to bounce more often causing the average path length to be higher.

We distinguish two ray tracing use-cases: progressive and real-time rendering. In the former, the acceleration data structure is constructed once, and the rendering takes place as long as the scene/camera does not move. For real-time, the acceleration structure is used for a single rendering step only and rebuilt thereafter. This distinction affects the target function of our tuning scenario. The performance of progressive rendering is dominated by the rendering step, so the tuner should minimize rendering time. The target function of real-time rendering must additionally account for the time required to build the data structure. We minimize the sum of build and rendering time in this case.

3.3.4 Performance Results: Exploration Through Search-Based Online-Autotuning

In Section 3.1.5 we demonstrated that autotuning does improve performance. Figure 3.3 shows typical tuning runs. We observe again the initial performance degradation during the first few iterations. After about 60 iterations the runtime converges to a better result than the recommended configuration.

Additionally, we see how the choice of the target function (i.e., optimizing rendering time or total time) affects the configuration found. When tuning rendering time, the tuning result outperforms the default configuration by 18% as shown in the left-hand plot. Considering the total time on the other hand, the *same* tuning result shown in the right-hand plot degrades performance by 14%. However, long convergence and wildly varying render time are still a major problem for online-autotuning.

3.3.5 Performance Results: Exploitation Through Predictive Online-Autotuning

In the following we report the performance results achieved by the prediction component of our hybrid tuner.

We compare the performance of the predicted configurations to search based tuning. To understand the comparisons, consider the motivating example shown in Figure 3.16: we show the typical runtime behavior as observed during the tuning process for one of the camera positions of the *car* scene. The jagged black curve shows the measured runtime during every iteration of a Nelder-Mead search. The purple dashed line represents the tuning curve (roofline) of an ideal predictive autotuner: using the best configuration from the baseline. Note that this is not the *true* roofline, which we cannot know without exhaustively exploring the configuration space. We consider as roofline here the best configuration we found using the search. Lastly the green horizontal line exemplifies the result produced by our predictor. In general, it produces a configuration that is worse

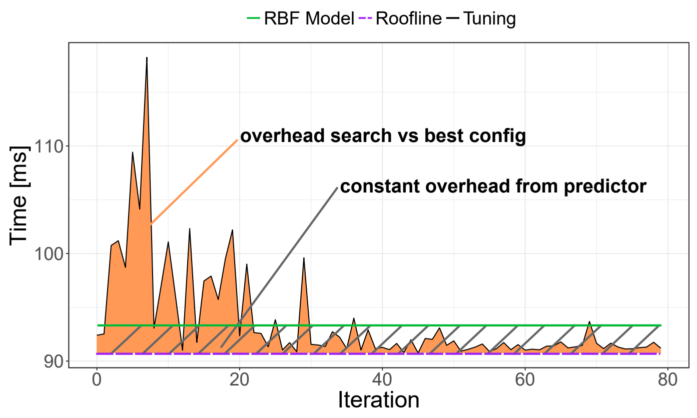


Figure 3.16: Example: Tuning on the *car* scene. The orange area shows the total overhead of 80 iterations over the best configuration (purple dashed). The hashed area shows the difference between our predictor (green) and an ideal predictor (purple dashed).

than both the roofline and the tuning result. In the following sections, we quantify how much worse the prediction result is. The primary goal of the predictor is not to produce a configuration as good as the roofline or the tuning search. Instead, its purpose is to reduce the total time spent searching. Progressive rendering accumulates each subsequent frame to improve quality. For our evaluation, we want to improve the time required to compute a fixed number of frames. The time required to complete 80 frames, or iterations, in the example in Figure 3.16 is the area under the respective curves. The area highlighted in orange in the figure shows the performance of the autotuner. The hashed area in turn shows the predictor performance. In the evaluation presented in this section we analyze our tuner with regard to both aspects: How much does our predictor improve the time required to complete a fixed number of iterations, and how close to the ideal performance does it get.

3.3.5.1 Generating Training and Validation Data

We want to generate the baseline in order to train and evaluate our model. We need a variety of indicator states for the evaluation scenes, for which we generate camera positions throughout the scenes. The camera path is designed on each scene to explore different amount of visible geometry and more or less complex lighting environments. We create 100 cameras positions for every scene, interpolating the movement using a simple spline.

To produce training and validation data, we first run the classic Nelder-Mead tuner for 80 iterations. The number is sufficient for the search to converge for all possible camera positions. Each search iteration builds the acceleration structure and renders one sample per pixel. Given seven scenes, 100 cameras and 80 iterations, we obtain 56000 data samples. The evaluation machine features an i7-6700 at 3.40 GHz with 8 hardware threads, 8 MB of cache and 32 GB of RAM. The baseline data is recorded as mappings of indicator states to parameter configurations and timings.

From the baseline data we randomly pick 80 camera positions for training, and the remaining 20 for validation for each scene. We repeat the training and verification process 15 times. To train the nearest-neighbor predictor, we generate a lookup table. It maps indicator states to the optimal configuration from the baseline found for this state. The table for each training round contains $80 \cdot 7$ configurations. For an unknown state, the nearest-neighbor predictor returns the configuration that the table maps to the closest known indicator state in terms of Euclidean distance. The RBF model is trained on the full training set of $80 \cdot 80 \cdot 7 = 44700$ data points. Using the update rule described in Section 3.3.2 we obtain the linear combination of RBF features.

For the verification phase we evaluate the predictor on the 20 remaining camera positions. We compare the performance of the configuration returned by the predictor with the baseline data. Note that this leaves room for error: the baseline data does not necessarily contain the globally optimal configuration. Only an exhaustive exploration of the parameter space could find that configuration. However, since we are only comparing search-based and predictive autotuning here, we believe that our comparison is sufficient.

3.3.5.2 Optimizing Render Time

We first analyze the behavior of our predictor for progressive rendering, which means rendering time only is to be minimized. In Figure 3.17 we show the speedup achieved by the predictors over the search baseline. The search baselines are the accumulated rendering times for 80 spp for the respective scenes and camera positions. In the figure, we compare the result of the table-based nearest neighbor predictor and the RBF model. We train the model as explained above. Although all observations are used to build the model, only the last iteration of each search are considered by the predictor, assuming the search converged.

The speedup results show that both the nearest neighbor and the RBF model predictor outperform the baseline in most cases. On average, using the geo-mean, they achieve a speedup of 1.05 and 1.04, respectively. However, we see two scenes where the RBF model does not find adequate configurations, namely *buddha* and *vokselia_spawn*.

Based on the speedup results we can also compare the overheads. We consider the overhead against a virtual baseline, which is the best-known configuration for a camera and scene. Unlike for the roofline comparison done below, we choose the best among both the search and the predictions to make sure the overhead is non-negative. We achieve a geo-mean overhead reduction of up to 87.5% (for the *sponza* scene) using the nearest-neighbor predictor. The RBF predictor reduces only up to 79.2% of the overhead (for the *gallery* scene). Although the speedups appear to be small, the presentation here hides an important fact: We are comparing only render time. The Nelder-Mead search samples different configurations at every iteration. Changing the configuration requires rebuilding the acceleration structure. In practice this incurs a substantial overhead for the baseline, but we exclude this in our comparison here for fairness.

In Figure 3.18 we compare our predictors against the roofline for every scene and camera position. In all cases both nearest neighbor and the RBF model are close to a speedup of one versus the rooflines. The geo-mean for both is above 95%. Although the speedup we achieve over the search appears small, we are close to what is actually achievable.

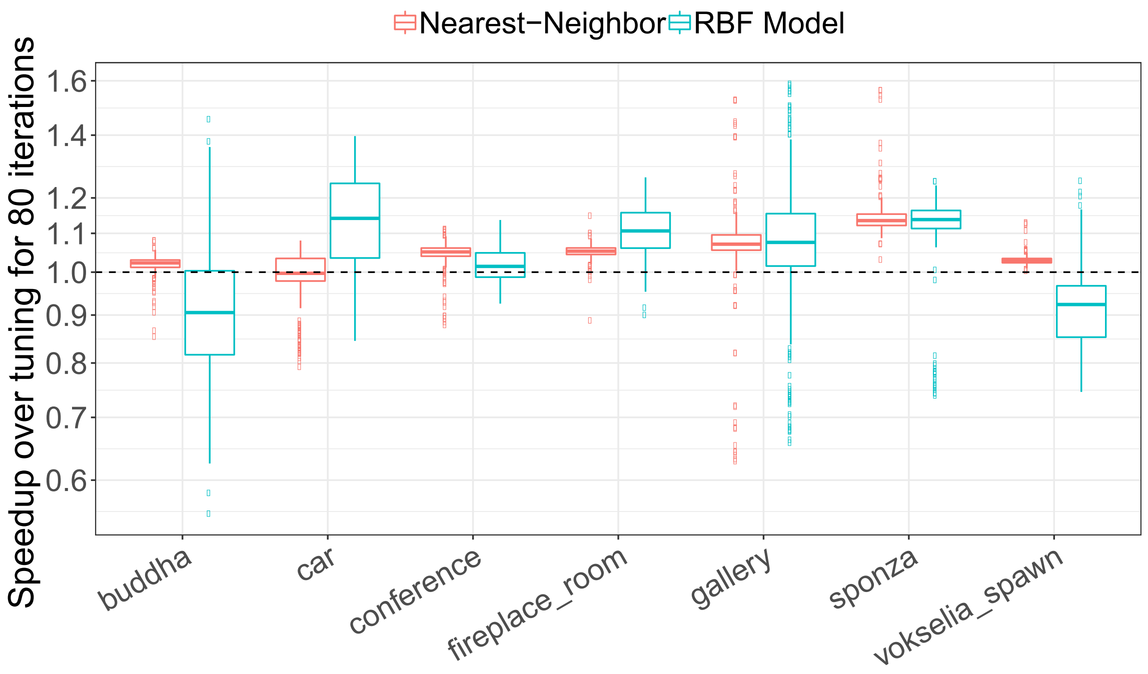


Figure 3.17: Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time only.

Interestingly, the RBF model outperforms the roofline in several cases. This indicates that the baseline search has not found the globally optimal configuration for these scenes. This is a caveat of the Nelder-Mead search algorithm, which only converges to local optima. In Figures 3.17 and 3.18 we note outliers for the *gallery* and *sponza* scenes. These data points are consistent with the camera transitioning from non-occluded to highly occluded configuration.

Given the performance increase of only five percent one may ask: Why choose prediction over search? The answer lies in the overhead of the baseline. We present another view on the search and prediction comparison in Figure 3.19. The graph shows the number of tuning iterations required for the search to outperform the prediction. The graph shows the number of iterations at which the cumulative search time and cumulative prediction time cross. For the RBF model the average break-even point is 362, and 1771 for nearest neighbor. On average at least 362 iterations are necessary to observe a benefit from the configuration found by the search. For visualization, we excluded 875 data points from the plot. That set includes 30 points for which the break-even point is greater than 10000. Those refer to predicted configurations which yield similar performance to what the search produced, so the curves are nearly parallel. For 845 points the break even point is negative, which are cases where the predictor found a better configuration than the search. For the RBF model these stem predominantly from the *car* and *fireplace_room* scenes, where roughly 12% of the data points outperform the roofline.

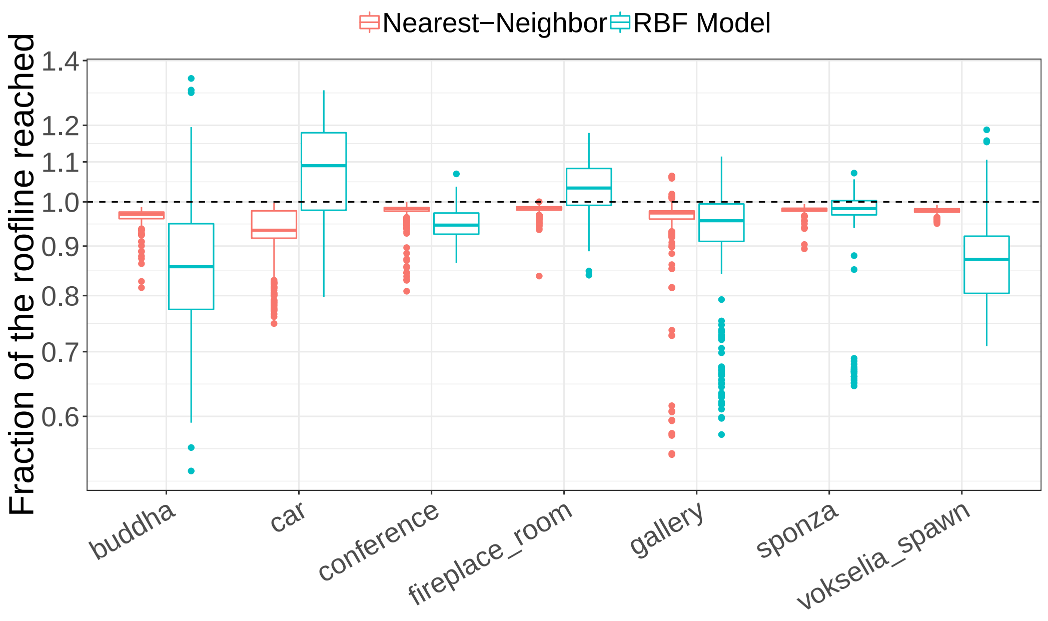


Figure 3.18: Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time only.

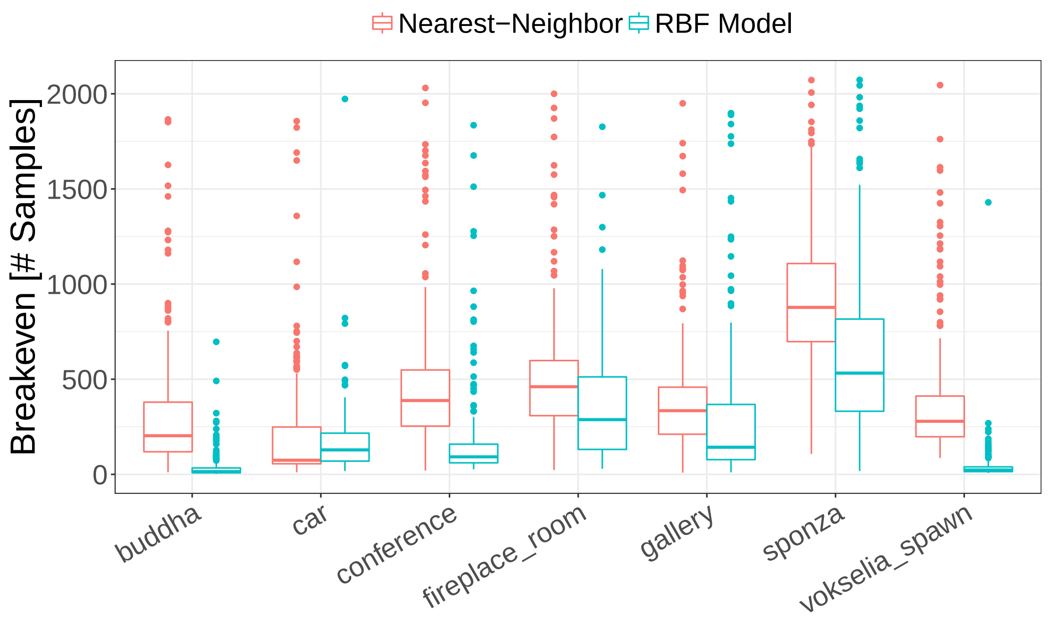


Figure 3.19: Break-even points of the Nelder-Mead search. A substantial number of Nelder-Mead iterations is required to break-even with predicted configurations.

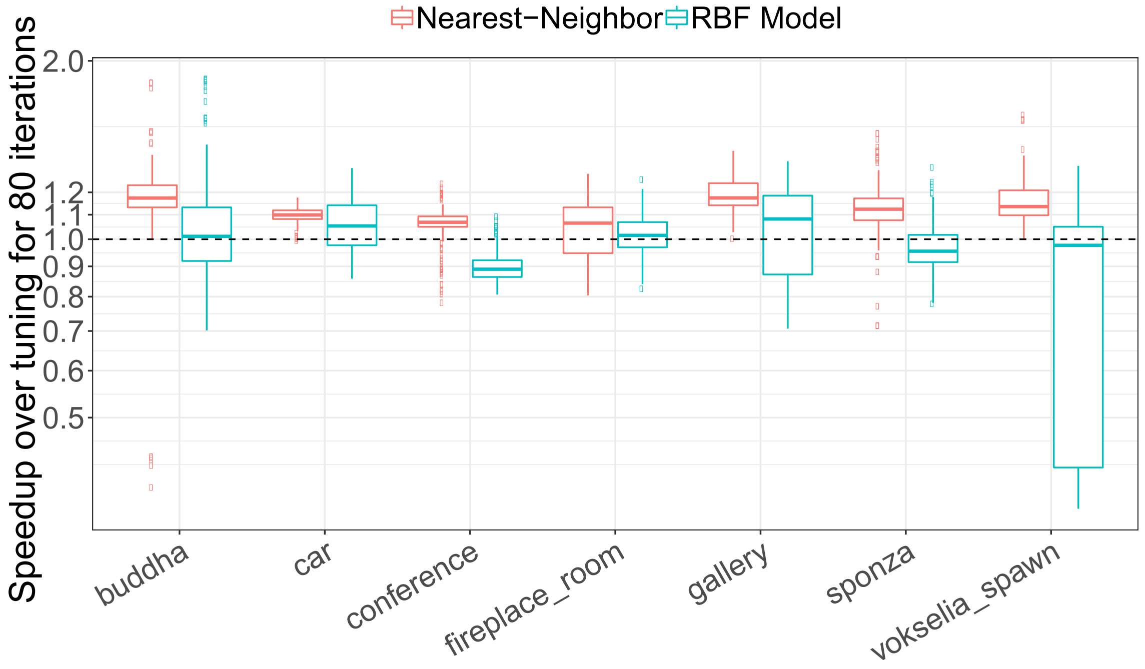


Figure 3.20: Speedups of our predictor over search-based tuning. Both search and predictor minimize rendering time and building time.

3.3.5.3 Optimizing Render and Build Time

We also evaluate the behavior of our predictors in a real-time raytracing context. In this use case the BVH must be rebuilt for every frame. The feedback function of the tuner measures the sum of rendering time and building time in this scenario. Figure 3.20 shows that the nearest neighbor approach still outperforms the baseline in most scenes. The only scene where more than 25% of the runs do not outperform the baseline is *fireplace_room*. On average, the nearest-neighbor approach achieves a speedup of 12%. The RBF model shows good behavior on *car* and *gallery*, but the performance is underwhelming compared to the previous approach and does not accelerate rendering. When minimizing total time, we achieve a geo-mean overhead reduction of up to 89.3% (for the *vokselia_spawn* scene) using the nearest-neighbor predictor. The RBF predictor reduces only up to 53.4% of the overhead (for the *gallery* scene). In Figure 3.21 we compare our predictors to the roofline for every scene and camera position. The roofline is the best configuration found during search. The nearest-neighbor predictor is close to the roofline in most cases with an average of 92%. Compared to the nearest-neighbor predictor our RBF model shows greater variance and slower performing averages with 83%. We note that the RBF model tends to cross the roofline more often than the nearest neighbor variant. This is due to the RBF model finding a better local optimum in configurations that do not satisfy the nearest-neighbor criterion.

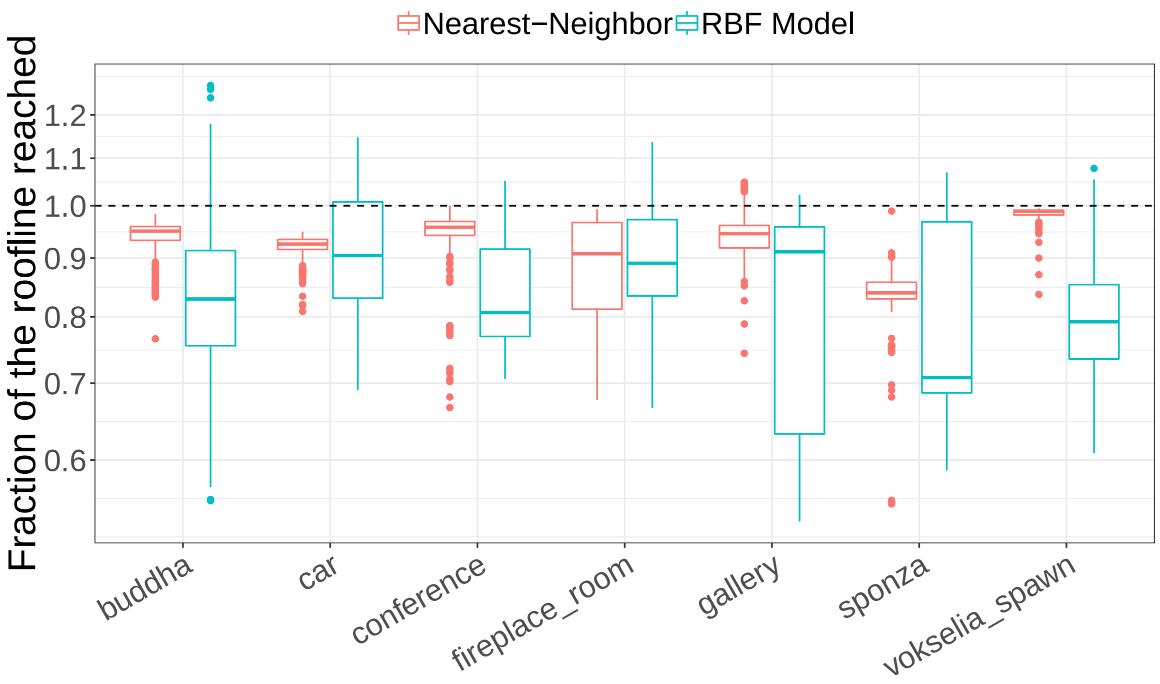


Figure 3.21: Comparison of the predictors against the rooflines for every scene and camera position. Both baseline and predictor minimize rendering time and building time.

3.4 Conclusion

In this chapter, we addressed two major caveats encountered with acceleration structure autotuning. We explored the context sensitivity of their parameters, and showed that while some parameters can be easily tweaked by an experienced engineer, some other parameters, such as the SAH costs, require automated optimization schemes.

We used a correlation matrix to explore the relationships between the parameters. We showed that triangle splitting offers low variance for both rendering and building time. Yet this stability is only apparent. When under load, the performance drop in the case of animations cann be considerable: 26% in our test scene. Selecting or constraining the tuning parameters will reduce the parameter space and improve the convergence of the autotuner.

The convergence stays relatively expensive within the constraints of ray tracing applications, specifically real-time ones. The Hybrid Online Autotuning method further alleviates this caveat. We analyzed scene data, from which we developed several indicators relevant to the performance of the acceleration structure. We introduced a machine learning model that maps these indicators to the parameter configuration of the acceleration structure. This model and the autotuner are integrated in a renderer. The ϵ-Greedy algorithm chooses whether to explore using the autotuner or exploit using our model.

Our model-based prediction does not need a sampling scheme, eliminating the need to rebuild acceleration structures, thereby saving precious computing time. Combining both search and model approaches is a good balance to mitigate the downsides of each strategy. Our evaluation on seven scenes of varying complexity shows that our predictors can compensate for the overhead of the search-based tuning.

A nearest-neighbor method achieves 95% of the performance offered by the search-based tuning while reducing the overhead by 90% of the rendering time. Because the nearest-neighbor predictor requires maintaining large input state tables, we also investigated a function approximation approach. We trained a radial basis function model during the search to provide the predictions. Although the space complexity of the model is constant with respect to the number of inputs, its performance is competitive. Our approach is also scalable to many-node parallelism. We can either use a predictor for each node or a predictor for the entire architecture. Hierarchical optimization or per node optimisation is promising and future work in this direction is a likely avenue for more improvement.

4 Autotuning Markov Chain Monte Carlo Metropolis Light Transport

4.1 Introduction

Markov Chain Monte Carlo rendering techniques take a conceptual side step from typical Monte Carlo techniques. Indeed, instead of re-creating a path from scratch, Markov Chain techniques modify the current path. How to best modify the current path is still a very active area of research, and it corresponds to finding good mutation techniques. The current research suggests that some mutation strategies are better at some specific rendering scenarii, such as diffuse-specular-diffuse reflections. MCMC frameworks are versatile and allow combining several techniques to effectively resolve different rendering scenarios For instance, combining an approach that is good for specular reflections with an approach that is good for difficult geometry. We propose to improve on the fundamental principle underlying most methods: the mutation strategy. Most mutation strategies use parameters to adjust to the use case. There is often room for improvement for scene-to-scene optimization. In Primary Sample Space MLT [Kel+02], the way the path is modified depends on a probability called large step mutation probability. If the large step check fails, the path is determined by a normal distribution, centered on the current value, of which the width parameter is fixed. If the check succeeds, the path is modified in a uniformly random manner, which is the large step. This fixed width value is satisfactory in combination with a high probability of a fully uniform random sample. Yet, the ideal parameter value depends on the local situation at the sample point. We propose a modification of this width parameter of the normal law according to the material the samples land on. We optimize a number of parameter values and use them to improve the convergence of the technique. Our optimization is performed with an autotuner during the proper rendering of the image. Our autotuner uses a Nelder-Mead algorithm for finding one of the best performing configurations. Our contribution then covers the introduction of out-of-the-loop autotuning, the demonstration that this method can improve performance of PSSMLT and the introduction of our σ-binning strategy. This work led to the submission of a short paper to EuroGraphics in December 2022 from which this chapter is based on.

4.2 Context

We now detail MCMC MLT methods with the notation we will use. This is the extended part of the fundamentals, pertaining specifically to this section of the dissertation.

MCMC rendering methods aim at computing the light contribution on the camera sensor from all possible paths $\bar{u} = (x_1, \ldots, x_k)$ for vertex counts $k >= 2$ in path space Ω. The value of each measurement I_j, with Veach's notation, in terms of an integral over all paths:

$$I_j = \int_\Omega f_j(\bar{u}) d\mu(\bar{u}) \tag{4.1}$$

with f_j the the measurement contribution function, including the pixel filter, and μ is the product area measure. One core property of the path integral formulation is that, contrary to the integral equation, it is not recursive and incorporates both measurement and light transport equations. The main idea of an MCMC MLT algorithm with this formulation is to mutate each path and either accept the new proposed path or reject it. Instead of creating entirely new paths, it only changes existing ones. Many mutation schemes exist for various applications.

One difficulty is the design of new mutation strategies. Moving the vertices for instance is a typical mutation scheme, for which one needs to sample a somehow relevant distance. Primary Sample Space MLT moves this difficulty to the space of random numbers, essentially applying the mutations to a unit hypercube that the mutation uses to generate their samples.

Multiplexed MLT goes one step further by also allowing the length of the camera and light paths to be changed and then reconnected.

For an overview of the literature, we refer to the survey by Sik et al. [SK20]. For more details, we refer to the Section 2.6.

MCMC algorithms, introduced to rendering through MLT [Vea98] often make use of symmetrical proposal distributions, i.e. Metropolis algorithms [Gey05]. PSSMLT[Kel+02] in particular makes this very obvious by replacing complex path space mutations by a simple mutation in the random number hypercube. The complex path space mutations can be specific to each purpose. We modify this simple mutation scheme to account for specific scene-dependent features.

Adaptive Mutations

Several approaches strive to control the perturbation (parameters) for MCMC rendering. Zsolnai et al. [ZS13] propose to control the selection probability for the large-step mutations based on gathered statistics. The approach by Li et al. [Li+15] adapts to the local structure of the state space by using the derivatives of the integrand. Hachisuka et al. [HJ11] applied adaptive MCMC in the context of photon tracing. Otsu et al. [Ots+18] adapted the mutation to the surrounding geometry around a path using BVH cuts. These approaches rely on domain knowledge obtained from either the target function or the generated samples, thus they are tightly coupled to the specific mutation strategy. In contrast, we aim at not being dependent on the mutation strategy. Dahm et al. [DK17] improved light source sampling using a model-free Reinforcement Learning technique. They reduced the number of zero contribution paths and the average path length.

Autotuning has been applied to Monte Carlo methods by Fichtner et al. [Fic+21]. They optimize weights in a matrix corresponding to their system to solve a non-linear inverse problem related to particle trajectories.

4.3 Method

Motivation

In this section we give an overview of our method which treats the rendering process as a black-box whose performance depends on a set of input parameters, i.e. our method is independent of the underlying technique. The autotuning process only modifies these inputs and assesses the output without using domain knowledge of the rendering process. As the optimization is outside of this black-box, we refer to it as out-of-the-loop autotuning. Since we rely on the high-level interface of the renderer, the underlying technique can be versatile, contributing to the flexibility of the design.

We apply our technique on a well-known method, PSSMLT, that has notably one important parameter. To not only demonstrate our framework for a single, global parameter, we introduce more fine-grained tuning capabilities with our novel σ-binning (Section 4.4).

This increased granularity (more tuning parameters) would be more difficult to control for a user and thus can be considered a good application for autotuning.

Top-Level Optimization Loop

Starting from an initial set of parameters, out-of-the-loop autotuning executes the rendering process with a fixed number of samples. Next, the metric to be optimized (e.g., a perceptual metric, variance, rendering time, acceptance ratio, etc.) is computed from the outputs of the rendering process. Based on the history of input parameters and the outputs, the autotuner determines the next candidate set for the parameters. We iterate this process for a fixed number of iterations, but it can be extended to find better parameters. Indeed, the parameter landscape is quite large, as it is for most optimizations, and finding the global optimum is not a practical objective. However, the convergence is often satisfactory after about 50 iterations. When we stop the tuning, we continue using the best configuration explored so far.

4.4 Parameterization and Autotuning with σ-binning

Variable Scaling for Local Direction Sampling

The original PSSMLT employs the single global parameter for the perturbation, which specifies the standard deviation of the per-dimension normal distributions. Mapping through the path sampler, the perturbation of an element of the primary sample corresponds to the perturbation of the sampled local directions. Thus the directional perturbation occurs irrespective to the surface properties. Our idea for the new mutation technique is based on the observation that the scaling parameter should be dependent on the surface properties.

Parameterization

We use the PDF of the material to define which value of width of the normal distribution, the parameter σ, is to be used next. The PDF describes how likely are particular events. Using this information, our autotuner can infer the proper values for the values of σ yielding the highest reduction of variance on average, or any other variable of our choosing.

Given different incoming directions and materials, the PDF for BSDF sampling obtained during the path creation will also be different. Based on this insight, we classify the scaling parameters according to the reciprocal PDF of the local direction sampling. In other words, we select the value of σ using the PDF of the BSDF at the intersection point parametrized by the incoming angle. Let $\boldsymbol{u}$ be the element of primary sample corresponding to the direction sampling in 3D, which is a 2D vector. We define a function

$$\Theta(\boldsymbol{u}) = \frac{1}{\alpha \cdot p_\sigma(\omega(\boldsymbol{u}))},$$

where $\omega(\boldsymbol{u})$ is the mapped direction using the path sampler and $p_\sigma(\omega(\boldsymbol{u}))$ is the PDF of the local direction sampling with respect to the solid angle measure. Here, the parameter α controls the spread of the PDF, which we will detail later in this section. We define the binning as follows. Let M be the number of bins. We split the interval $[0, 1)$ into the partition $A_1, A_2, \ldots, A_{M-1}$, where $0 < A_1 < \cdots < A_{M-1} < 1$. For convenience, $A_0 := 0$ and $A_M := 1$. Here, the interval $[A_{i-1}, A_i)$ corresponds to the i-th bin. We define the scaling parameters $\sigma_1, \sigma_2, \ldots, \sigma_M$ for each bin. Then the scaling parameter corresponding to the current state u is defined by

$$\sigma(u) = \sigma_i \qquad \text{if} \qquad \Theta(\boldsymbol{u}) \in [A_{i-1}, A_i).$$

We use the same mapping Θ for all local direction samplings, irrespective of the number of bounces. The partition $A_1, \ldots, A_{M-1}$ and the scaling parameters $\sigma_1, \sigma_2, \ldots, \sigma_M$ are exposed as a set of parameter for the mutation strategy. These parameters are updated by the autotuner. According to our testing, a tractable number of bins for the autotuner is around 5 to 8. To evaluate how impactful the bins will be, we used $M = 200$ equally spaced bins and counted how many samples fell in each to produce the histograms in Figure 4.1. This gives an intuition of the landscape of PDF in the scene. This can also be used to target specific materials. According to our experiments summarized in Figure 4.1, we chose to use $\alpha = 4\pi$ in the remaining of the paper. This is a sweetspot between not clamping to much energy into the 1 bin, and having a spread out PDF with relevant features well separated.

We chose to bin the PDFs according to $C = \frac{1}{N\pi p(\omega_o|\omega_i)}$. This allows to separate the different materials and situations in the scene, and give similar σ to the samples that have a similar value in this mapped space.

We created several variations of the *cornell-box* to highlight that different materials produce a different binning landscape (Figure 4.2), by changing the materials of the two boxes; *specular-plastic*: specular and plastic (0.9 roughness), *box-plastic*: diffuse and plastic, and *specular-box*: specular and diffuse. This binning strategy can reveal which material typically has which PDF value.

Bin Positions as Parameters

Incorporating the bin positions into the optimization is crucial to obtain satisfactory results. The number of bins could also be an optimization parameter, but convergence has become much more unstable. We observed that less than ten bins are sufficient to achieve a decent separation. We report a run with seven different σ in Figure 4.3. This figure shows the bins

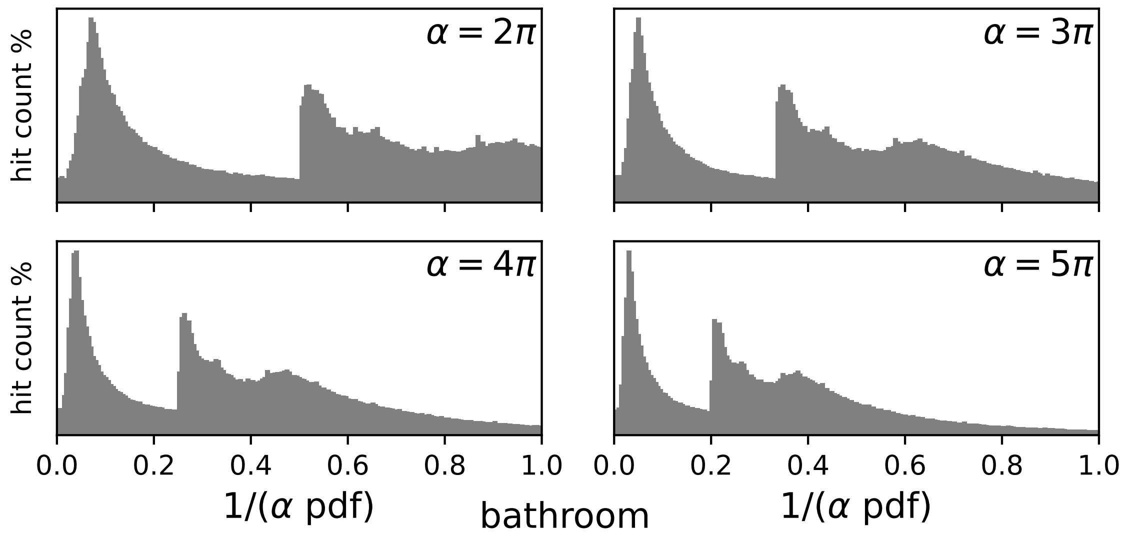


Figure 4.1: The distribution clumps together at high α. It does not sufficiently separate interesting features at low α. $\alpha = 3\pi$ or $\alpha = 4\pi$ seem the most appropriate compromises.

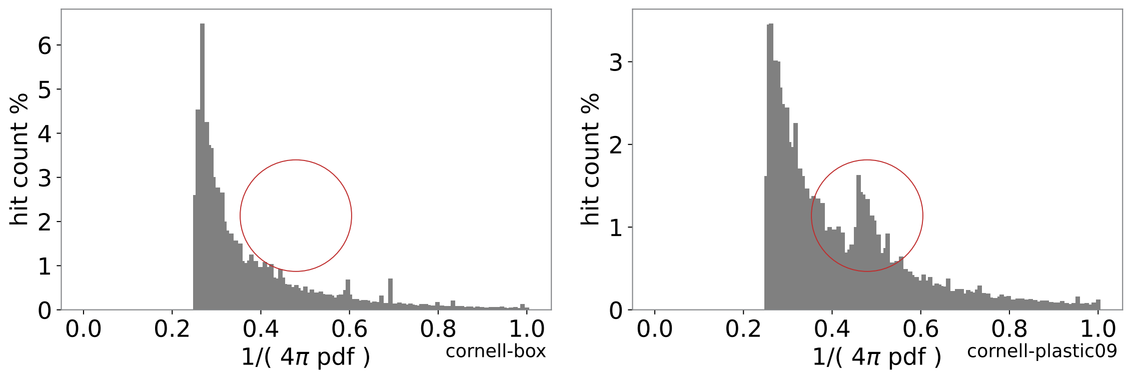


Figure 4.2: Left: Histogram of the PDF landscape of the *cornell-box* and **Right:** of the *cornell-box* with a plastic BSDF with roughness 0.9.

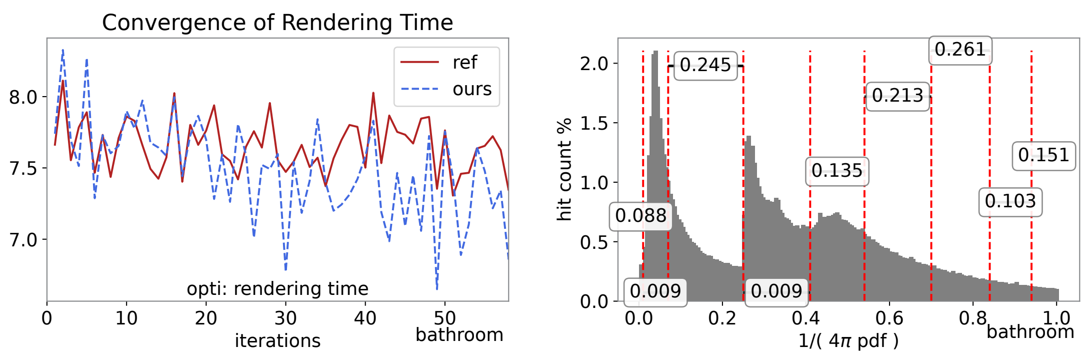


Figure 4.3: Left: Convergence during tuning. Red line is a non-tuning run (always same configuration) to show the variance between iterations. **Right:** Different bin positions and σ-values have been optimized. Bins locate features and the values are in a sensibe range.

correctly encircling interesting features. The first and second bin separate the leftmost sharp feature from the flat part preceding the next sharp peak. The large (resp. small) σ values show where the samples in this remapped space require larger (resp. smaller) mutation size. Note that each bin has an effect proportional to the area of the histogram they contain. It is clear from this distribution that less bins could have been employed as the tail of the distribution seem to prefer uniformly higher values of σ.

Handling Constraints in Autotuning

The autotuner can only tune parameters between fixed bounds and cannot handle the constraints between parameters. This is an issue as we require our bins to be in increasing order, i.e. bin 2 cannot be further to the right than bin 4. Intuitively, we could have parameters that add together to one, and use their inclusive sum as bin position. As we said, there is no way to ensure the autotuner will produce values that fulfill a criteria that depend on other parameter. One solution is to use an additional parameter and use it to artificially induce our constraints.

In order to impose the constraint corresponding to the bins A_i being in sequential order, i.e. $0 < A_1 < \cdots < A_{M-1} < 1$, we introduce a mapping of the partition $A_1, A_2, \ldots, A_M$ by a sequence of numbers $a_1, \ldots, a_{M+1}$ in $[0, 1)$, defined by:

$$A_i = \frac{1}{S} \sum_{j=1}^{i} a_j, \tag{4.2}$$

$$\text{with} \quad S = \sum_{j=1}^{M+1} a_j. \tag{4.3}$$

In the first equation, i satisfies $i \in \{1, \ldots, M-1\}$ as we essentially compute the prefix sum of the parameters, S is the sum of all parameters, a_i is the ith parameter value returned by the autotuner. These equations in the end ensure the desired property of the bins adding to one:

$$\sum_i A_i = 1$$

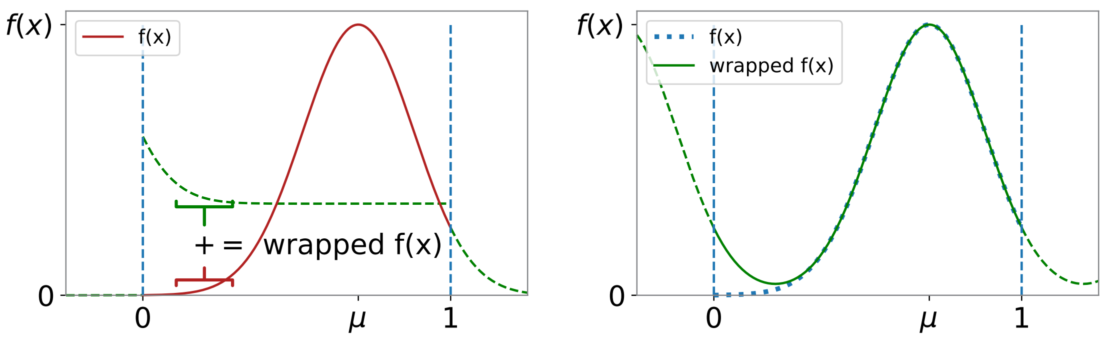


Figure 4.4: Left: Non-wrapped distribution. In green the rest of the function in $[1, \infty]$ and $[-\infty, 0]$. **Right:** Wrapped distribution. It is an infinite sum (only approximate here)

This mapping guarantees that the numbers (representing the bin position) $A_1, \ldots, A_{M-1}$. One way to see this process is that we use $M + 1$ parameters and constrain them with one equation, which removes one degree of freedom, leaving us with M proper parameters.

In our project, the bins are not the only parameters being optimized of course. The autotuner optimizes the parameters $a_1, \ldots, a_{M+1}$ and $\sigma_1, \ldots, \sigma_M$.

This corresponds to two tables. These values are optimized within predefined bounds: $[0, 1)$ for each of $a_1, \ldots, a_{M+1}$ and $[0.0001, 0.3]$ for the σ values.

Optimization Criteria

The optimization criteria can be a variety of measurable quantities, such as rendering time, (r)RMSE, acceptance ratio etc. We choose to optimize with respect to the acceptance ratio. Indeed, there is an optimal value of the acceptance ratio, 0.234, that can be derived mathematically under certain assumptions[AT08]. We found it to be satisfactory in our case except for very simple scenes where the acceptance ratio is supposed to be much bigger. We also tried other metrics such as the number of zero-radiance rays, rendering time, and other values for the acceptance ratio. It seemed that the assumptions for the optimal acceptance ratio are not fulfilled properly for all scenes, especially the simpler ones, and the optimal acceptance ratio may be different for different scenes. Optimizing for rendering time provide about the same improvement as the acceptance ratio. The optimization criteria is indicated on the different graphs presented.

4.5 Non-symmetrical MCMC mutation

Most MCMC methods use symmetrical mutations. PSSMLT makes it obvious by only using one value for the standard deviation σ, which can be seen in Figure 4.5. As we have seen in Section 4.3, we can only simplify the transition probabilities in this case. To obtain this probability, we need to compute the probability of landing on a particular sample of the wrapped normal distribution. The normal distribution having infinite support wraps around $[0, 1]$ infinitely many times, and we are left to estimate an infinite sum. Indeed, in the original case, one output value can be generated an infinite amount of ways, because

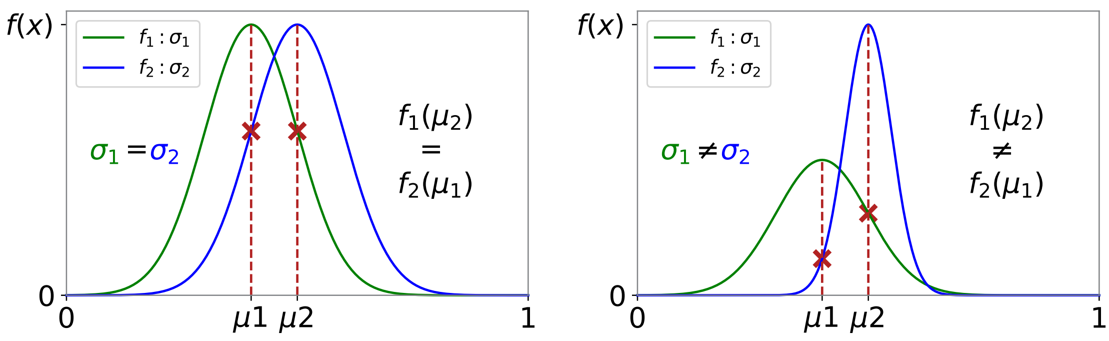


Figure 4.5: Left: Symmetrical mutation. The normal distributions have the same standard deviation. **Right:** The two distributions have a different standard deviation, which makes the mutation non-symmetrical.

the distribution is wrapped. To obtain the probability of such a sample is to take the sum of all the probabilities that this event happens.

A comparison between wrapped and non-wrapped distributions is depicted in Figure 4.4. This estimation is very costly and unrealistic in our time constrained context. We sidestep this issue by using a truncated distribution, which allows us to select the support as we please. We first experimented with an adaptive span of 3σ, but then noticed the number of wrappings would change, introducing inefficient branching code. Using a unit length support guarantees that each value in $[0, 1]$ has a potentially non-zero probability density. This also guarantees that the wrapping will always meet to form a continuum between $[0, 1]$, meaning we only need to evaluate the PDF once, compared to an infinite amount of times.

4.6 Implementation

To implement this method, we extended PBRT-v3. PBRT is a natural choice when it comes to repeatability and trust as it is a very well-known reference in the field. It also features an MMLT/PSSMLT implementation. We rely on this implementation for our method, and it is also used as a reference to compare against. The technique itself was easily implemented into the renderer. Unfortunely making our code into a simple plug-in to PBRT proved to be to much implementation hassle. Indeed, we modified several parts of PBRT to extract the necessary runtime informations.

We highlight here the most important mathematical considerations required for the implementation of our method.

Acceptance ratio

A typical PSSMLT implementation uses a normal distribution to mutate the current primary sample x_c into the proposal one x_p. In general the standard deviation σ_i never changes, and the acceptance ratio $a(x_c, x_p)$ of the Metropolis-Hastings update is the ratio of the contributions, i.e. the transition probabilities $T(a \to b) = T(b \to a)$ cancel out:

$$a(x_c, x_p) = \frac{f(x_p)T(x_p \to x_c)}{f(x_c)T(x_c \to x_p)}. \tag{4.4}$$

Our method changes this parameter, requiring the computation of transition probabilities for the acceptance ratio. Because PSSMLT uses a wrapped normal distribution, there is an infinite number of candidates for each sample position. Take $\mathcal{N}(\sigma = 0.2, \mu = 0.5)$ with PDF p, the probability of x is not necessary to compute $p(x)$ but also $p(x+1)$, $p(x+2)$... In fact, it is given by:

$$p_{wrapped}(x) = \sum_{i=-\infty}^{+\infty} p(x+i).$$

Instead of computing this infinite sum for the probabilities of the wrapped normal distribution, we use a truncated distribution, as seen in Chapter 2, Section 2.3. This truncated distribution allows for a closed form solution to this issue. We use the general acceptance ratio for PDFs with normal distribution:

$$D_c = \text{erf}(\frac{0.5}{\sqrt{2n_S}\sigma_c}) - \text{erf}(-\frac{0.5}{\sqrt{2n_S}\sigma_c}) \tag{4.5}$$

$$D_p = \text{erf}(\frac{0.5}{\sqrt{2n_S}\sigma_p}) - \text{erf}(-\frac{0.5}{\sqrt{2n_S}\sigma_p}) \tag{4.6}$$

$$a(x_c, x_p) = \frac{\sigma_c}{\sigma_p}\text{exp}\left(\frac{(x_p - x_c)^2}{2n_s}(\frac{1}{\sigma_c^2} - \frac{1}{\sigma_p^2})\right)\frac{D_c}{D_p}, \tag{4.7}$$

with n_S the number of jumps we perform, it increments every time the proposal is rejected, resets once accepted or a large jump is performed. The sampling is also modified, please see the appendix for the full derivation.

Tracking PDFs can be optimized by caching the sum of the all the terms inside the exponentials. Also tracking separately the terms outside the exponential allows to replace many multiplications and exponentiations by mostly additions and a single exponential.

Primary sample generation

To generate the primary samples, the MLT process uses a normal distribution. Typically, a small step mutation samples the normal distribution centered around the current sample with a fixed standard deviation. We use the truncated distribution, which changes the generating function. This distribution allows for lightweight probability computation compared to the wrapped normal distribution, on the left of Figure 4.4. The original wrapped distribution can sample on the infinite support, and does the wrapping after, allowing one sample to be generated in infinitely many ways. Our distribution has been chosen such that only one evaluation is necessary. We refer to the Section 2.3 for more details on truncated distribution sampling. The general form is:

$$T^{-1}(u) = F^{-1}(F(a) + \xi(F(b) - F(a))), \tag{4.8}$$

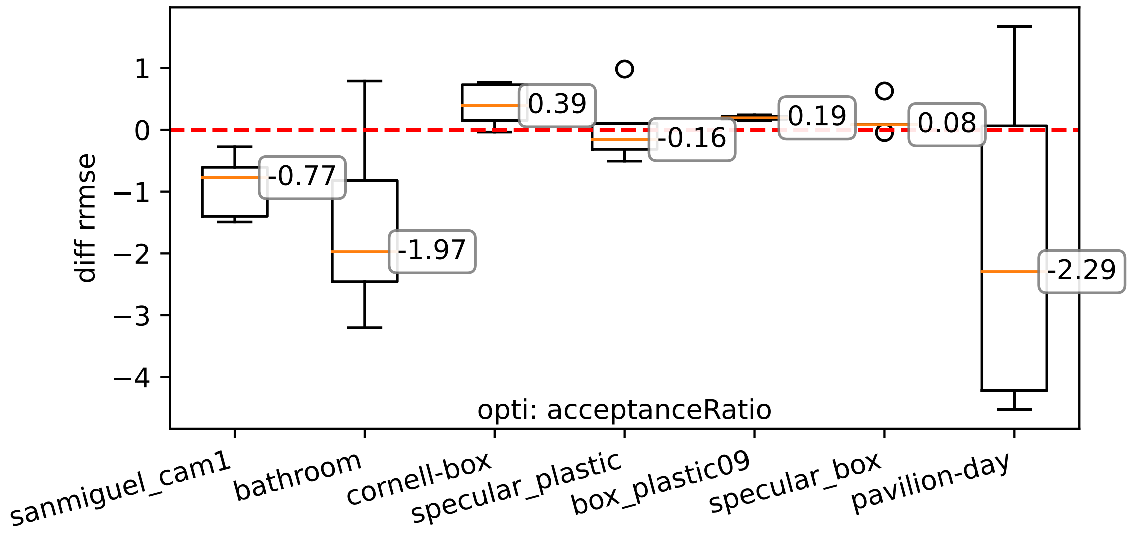


Figure 4.6: The distribution of the quality difference between *ours* and *original* on different scenes (lower is better). We observe that more complex scenes benefit more from our method.

which, for us corresponds to:

$$F_\sigma(k) = 0.5 + 0.5\,\mathrm{erf}(\frac{k}{\sqrt{2}\sigma}) \tag{4.9}$$

$$X = \sqrt{2}\sigma\ \mathrm{erf}^{-1}(2(F_\sigma(a) + \xi(F_\sigma(b) - F_\sigma(a))) - 1), \tag{4.10}$$

with $\xi \in [0, 1)$ a uniform random number and $a = -0.5$ and $b = 0.5$. When n successive jumps are needed, one can multiply the standard deviation σ by $\sqrt{n}$ directly instead of iterating this process several times, this goeg along with the consideration of the above paragraph.

The values of σ used in these computations are stored in the primary sample structure and are modified by the σ-binning method.

4.7 Evaluation

We have implemented our method in PBRT version 3 [PJH17]. We compared our method (*ours*) against the original small-step mutation (*original*), which uses the fixed scaling parameter $\sigma = 0.01$. This parameter value is the standard found in the literature, notably in the original paper and in the PBRT books. We treat the `large-step probability` as a hyperparameter, and chose a value that produces better results for each technique; *ours*: 0.1, *original*: 0.3 (default value). All experiments have been conducted on a machine with 8x Intel Xeon E7-8867 CPUs with 4TB RAM with 256 threads. For the experiments, we chose scenes with diverse characteristics: *san-miguel*, *contemporary-bathroom*, *pavilion-day* and *cornell-box*.

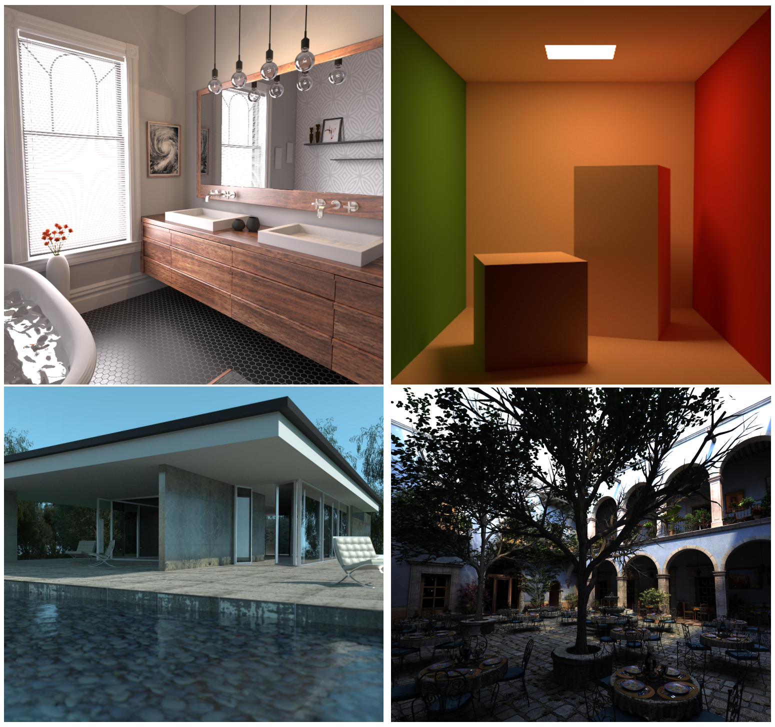


Figure 4.7: Top Left: Bathroom scene **Top Right:** Cornell-box scene
Bottom Left: Pavilion scene **Bottom Right:** San Miguel scene

san-miguel is a complex scene with difficult lighting. It features a building with a central outdoor opening, trees, many tables, chairs and reflective cutlery, and plates with complex textures. The light in this scene comes entirely from the environment map, and has to traverse through the center of the building. This makes for a difficult mixed direct and indirect illumation. It also contains fences, doors, large ceiling lights that are off (for some indoor part).

contemporary-bathroom is a small scene with many transparent and reflective surfaces. The light only comes from the wide window, partially covered by horizontal tassels from the blinds.

pavilion-day also features a pool and a villa with a complex interior. It has glass windows and is only illuminated by the environment map.

cornell-box is the well-known simplistic scene featuring a colored box with a few shapes inside. Note that we created variations of this scene, modifying the materials. This scene is a failure case for our method as it is so simple that our method only produces overhead.

For comparisons, we use the relative root mean square error (rRMSE). We optimize for acceptance ratio and error values are computed with respect to reference images, computed with 16384 samples per pixels with Bidirectional Path Tracing. We repeated the measurements between 5 and 10 times to display the error spread. For tuning, we perform 50 tuning iterations, each with 5 mutations per pixel (mpp). We select the best configuration explored. The tuning step is a Nelder-Mead update. Our implementation is reasonnably optimized, and this update is extremely fast for our number of parameters ($\ll 1\text{ms}$) and is negligible compared to rendering time. Note that every image produced

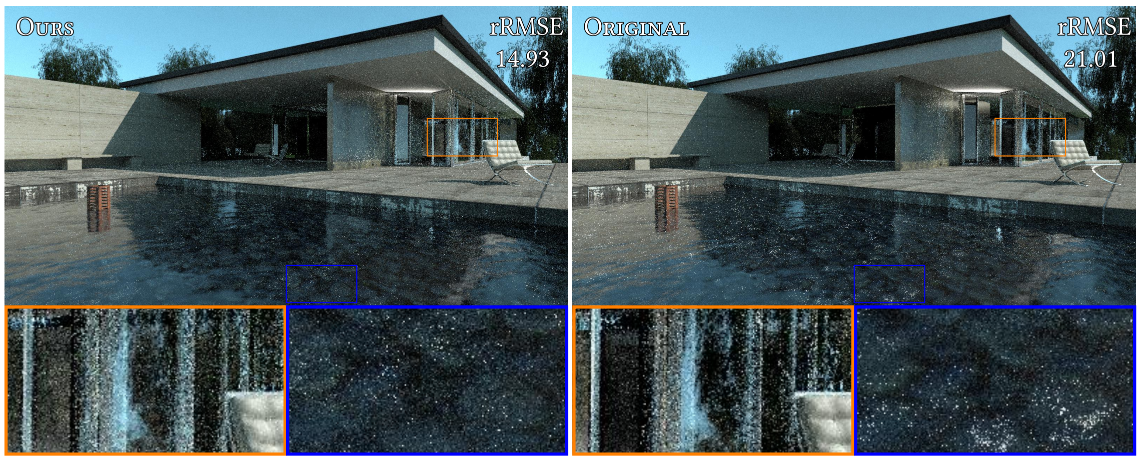


Figure 4.8: Equal-sample (100 mpp) rendering of *pavilion-day*. Our method achieves 1.3 seconds (4%) faster rendering time. The water surface has much fewer fireflies, the windows on the right are correctly resolved as well. This behavior is consistent with the Figure 4.6.

by the renderer, even if the quality for one specific iteration is subpar, improves the overall variance of the end image, as they are all combined. This is possible because our approach is unbiased.

We compare two approaches using the same number of samples: 100 mpp or 200 mpp depending on the scenes. Figure 4.6 summarizes the relative differences of rRMSE between the two approaches. Our method performs well on the scenes with more specular paths, as shown in *pavilion-day* (Figure 4.8) and *contemporary-bathroom* (Figure 4.9). We observe faster rendering for the former, and far fewer artifacts for the latter.

In the Figure 4.8 (corresponding to the most average point in the graph in Figure 4.6) we notice a visual improvement over the classic algorithm and was 1.3 (4%) seconds faster. This indicates that the quality difference is bigger with an equal time experiment. We noted a reduction in the average path length as well, which significantly reduces computation time and is likely responsible for the difference. The likely scenario is a better exploration at short path lengths, with more stability around the high contribution areas. In turn, this stability is normally disturbed by the `large-step probability` whose default value is 0.3. In our case, having such a high value prevents the autotuner from making a real impact on the scene as its parameters would be consistently disturbed. The autotuner effectively acts as a more granular way to implement this `large step probability`. Because our approach still introduces overhead via the probability computations, some scene can see their performance degrade. Indeed, our method does not seem to perform reliably well for very simple scenes such as *cornell-box*.

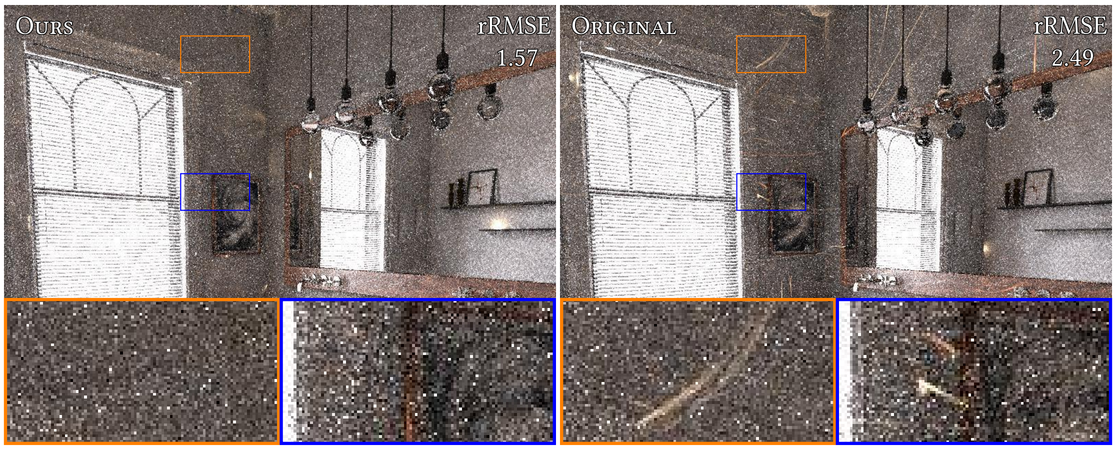


Figure 4.9: Equal-sample (200 mpp) rendering of *contemporary-bathroom*. The original implementation produces sharp features that are notoriously difficult to denoise whereas our approach presents far fewer of these artifacts, and a better rRMSE.

4.8 Conclusion

With the ever growing need for physically based rendering, MLT methods have proven to be reliable methods and fruitful research objects. However, these random walk sampling techniques require well parametrized mutation parameters to attain their full potential.

The method proposed in this thesis precisely addresses optimizing path space parameters, and is a first step towards self-configuring MCMC rendering pipelines. It uses an autotuner as an online optimizer. By replacing the fixed step size parameter by a collection of step sizes instead, we allow the control of the mutation size by other means. In our work, we presented a method using material properties and local sampling to determine the parameter value. The autotuner iteratively improves the different bin positions as well as the bin values for the standard deviation.

We tested our method on scenes of different complexities. While the computational overhead of the binning strategy yields diminishing returns for very simple scenes, the benefits emerge when tackling larger scenes. This novel σ-binning approach contributes to the field of context-aware path exploration. From drastically reducing noise and firefly artifacts to accelerating convergence in complex scenes, self-tuning approaches allows for better autonomous rendering capabilities.

Our method can be applied to all mutation strategies whose parameters can be scene dependent. The improvements will depend on the specific mutation strategy utilized.

5 Temporal anti aliasing

5.1 Introduction

Rendering and especially rasterization fundamentally rely on a sampling process to obtain colors for the pixels in the image frame. This inevitably results in aliasing artifacts because of the discrete nature of this process. In the past, many different techniques in the field of anti aliasing have been developed to reduce or remove these artifacts. Ideally, aliasing is reduced by increasing the sampling rate and combining multiple samples for a single pixel color, either in the form of supersampling or multi-sample anti aliasing [Ake93]. However, this approach results in higher computational load and memory consumption. Since the introduction of deferred shading rendering pipelines, such methods are no longer feasible and temporal anti aliasing (TAA) emerged as one of the most popular anti aliasing variants [YLS20]. TAA solves the aliasing problem by distributing subpixel samples over multiple frames which are accumulated after a reprojection step. Alternative approaches rely on spatial filtering by identifying aliased regions - such as jagged edges - and blurring them with an adaptive kernel [Res09]. Both methods are prone to overblurring and TAA can lead to visible ghosting artifacts.

In recent years, neural networks were utilized to solve the aliasing problem while reducing ghosting and artifacts. Neural solutions mainly rely on explicitly recreating additional subpixel samples through neural supersampling [Xia+20; Liu20]. This makes larger framebuffers and subsequently large networks necessary. In this paper, we propose a minimal, fully-convolutional neural network architecture that utilizes kernel prediction without supersampling to perform anti aliasing. Kernel prediction allows the network to output per pixel filter kernels instead of direct colors which, after its successful usage in networks for Monte Carlo denoising [Vog+18], was shown to work well for anti aliasing too [Tho+20]. Our architecture outputs a temporal and a spatial kernel, allowing for temporal supersampling as common in TAA as well as for spatial filtering. We show that our small network reduces ghosting and overblurring artifacts in comparison with TAA and has a high temporal stability. Our implementation is compact in its description, highly optimizable and yields promising results. In particular, our contributions are:

- a minimal neural network architecture for temporal anti aliasing,
- a method to let networks adapt to a set of temporal subpixel jitters from renderers without increasing their operation count,
- an analysis of adaptions on different architecture parts to gain insights into the requirements for TAA networks.

In Section 5.2 we give an overview of related works from anti aliasing for real-time rendering and deep learning solutions from denoising and supersampling. We present fundamentals required for performing temporal anti aliasing with neural networks in

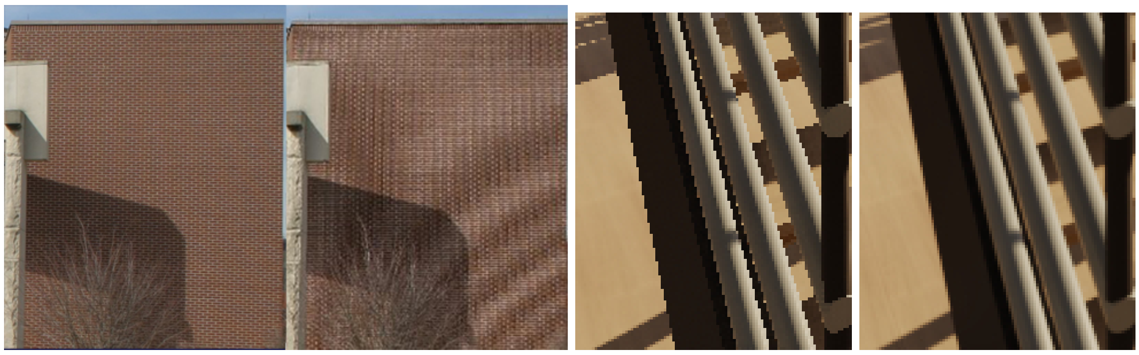


Figure 5.1: Left: Real-life example of aliasing on a brick wall. On the left, the camera has enough resolution to correctly resolve the pattern, on the right, the camera doesn't. **Right:** Example of aliasing (left) with rendered images. The side of the staircase visibly shows jagged edges, not present on the right.

Section 5.3 and explain our architecture in Section 5.4 and its training in Section 5.5. Finally, we evaluate the architecture with regards to its anti aliasing properties, reduction of artifacts, and operational cost including a comparison with a typical non-neural TAA solution.

5.1.1 Aliasing

Aliasing is the name given to the distortion of a signal having a frequency different from the actual signal, due to the sampling rate being below the Nyquist frequency of the signal. The Nyquist frequency correspond to the satisfaction of this inequality $f_s/2 > f$,with f_s the sampling rate for all frequencies f of the signal. It suffices that f_s be in practice at least double the highest frequency in the signal.

On the left side of Figure 5.1, not enough sensor pixels are present to accurately represent the wall, creating the Moiré effect. On the right, the rendering is done at a resolution too small. Because a straight edge contain infinitely high frequencies, it is not possible to simply try to resolve them. Instead we employ strategies of anti-aliasing. It can be as simple as blurring the image or using a low-pass filter, but of course over the years, many sophisticated algorithms have been developed.

5.2 Related Work

Anti-aliasing has a long history in computer graphics and rendering. One of the first approaches simply consisted in rendering the frame at larger resolution and using a filter to downsample it. This is of course unfeasible in most real-time scenarios as the frame budget is usually very tight. This approach is very similar to multi-sample anti aliasing [Ake93] (MSAA). In MSAA, the vertex shader uses 4 sample locations for calling the fragment shader. This incurs less overall cost than simply rendering at 4x resolution as some optimizations can be performed. But in combination with deferred rendering pipelines, MSAA lost its practicality and became less common as it leads to high loads on memory bandwidth [YLS20; Cra+15]. Single-frame methods that rely on morphological filters [Res09] (MLAA) and edge detection [Lot09] (FXAA) are usable in combination with deferred rendering. FXAA is now seen in many games as the cheapest possible technique and is still today a very good tradeoff between performance and quality. However, they are prone to overblurring and can misidentify patterns as aliasing since they do not solve the fundamental undersampling. For that reason, temporal anti alisiang [Yan+09] (TAA) is increasingly used in modern real-time rendering engines. Lei et al. [YLS20] present an excellent survey of TAA methods. As opposed to applying a locally higher sampling rate within a single frame as in MSAA, TAA accumulates samples over time. These samples are free in the sense that they would have been computed anyway. To accumulate information even on static viewpoints, a moving subpixel jitter is applied on the camera projection during rendering and the reprojected output of consecutive frames is averaged. This allows moving objects to be use reprojected information while static object also improve thanks to the subpixel jitter. The extensions of morphological anti aliasing proposed by Jimenez et al. also include temporal accumulation for that reason [Jim+12]. Commonly, a Halton 2,3 sequence is used for subpixel jitters [Kar14] which we also include in our approach. For moving cameras, reprojection using motion vectors as well as a rejection mechanism for previous samples that takes occlusion or changing illumination into account are required. This rejection is often heuristically solved, by clamping or clipping color history to a color space bounding box of pixels in a local neighborhood of the new sample [Kar14]. Salvi presents variance clipping TAA which takes the variance of color samples in such a neighborhood into account for clipping and produces comparably good results [Sal16]. TAA contains many challenges that need to be tuned manually [Ped16], including motion vector accuracy, reprojection, identifying stale history for rejection, or finding well performing blending weights [Zen+21]. This could be a very interesting setting in which to use an autotuner. Classic approaches often suffer from blurry output images or flickering from inaccurate history rejection [Ped16]. In recent years, deep learning solutions became popular for handling different subsampling problems with early techniques already mentioning anti aliasing [Nal+17] while commonly being applied to problems like denoising [Cha+17; Has+20; Tho+20] and upsampling [Wat20; Liu20; Xia+20].

Previous results from neural denoising showed that the prediction of local filter weights per pixel instead of a direct color yields better image quality and is trained faster, a method called kernel prediction [HY21]. Kalantari et al. [KBS15] propose this method for offline denoising of path traced images while Vogel et al. [Vog+18] extended the approach with motion reprojection. Kernel prediction was subsequently adapted for neural Monte Carlo denoising in real-time applications. Hasselgren et al. [Has+20] use a combined kernel prediction denoiser and direct sample guidance network. Fan et al. [Fan+21] compute an encoding of filter kernels to reduce the typically large number of weights of the kernel output layers in their networks. While computing TAA instead of denoising, our approach uses kernel prediction instead of directly outputting pixel colors.

Supersampling differs from anti aliasing in the sense that it creates a high resolution output from a low resolution input. This high resolution output could be filtered down to form a low resolution image without aliasing, but it is usually directly used. Anti-aliasing instead directly outputs the low resolution anti aliased image without explicitly creating the additional subpixel samples for an input image. Early experiments for neural temporal anti aliasing without upsampling were presented by Salvi [Sal17]. Even though non-deep learning solutions are still part of active research [AMD20], neural solutions are becoming fast enough for real time and are currently used in a wide range of video games and game engines. DLSS [Liu20] uses motion reprojection and temporal accumulation for neural supersampling. The DLSS framework also features a variant that directly performs anti aliasing called DLAA. Xiao et al. [Xia+20] present a large architecture that contains individual networks for processing the last 8 temporal subpixel samples. Our network does not have individual network paths for different frames to keep the number of required operations for computing pixel kernels low. Instead, we rotate through different sets of network weights linked to the current subpixel jitter index to handle samples independently in the temporal accumulation. Thomas et al. [Tho+20] significantly downscale the computational complexity of an image reconstruction network through integer quantization achieving good performance while maintaining good image quality.

Deep learning solutions for TAA solve problems that non-neural approaches face, mainly overblurring, ghosting, and flickering, but at the same time are expensive to compute because they rely on a larger number of instructions per pixel. In this paper, we investigate a minimal and simple convolutional kernel prediction network that is able to outperform variance clipping TAA without requiring manual finetuning, but at the same time does not rely on a high instruction count.

5.3 Background

Spatial single-frame anti aliasing uses a variety of methods and often relies on morphological information. Edge detection and smoothing are commonly used to remove aliasing artifacts [Lot09; Res09]. This produces decent results, but often leads to an oversmoothing of edges or details. The way TAA incorporates temporal information requires a variety of techniques to be viable such as motion vector reprojection and history rejection which are usually quite cheap to compute. Indeed, because the scene is in motion, objects move and visibility changes, the individual pixels must be processed before using them. Reprojection is the use of motion vectors describing how the individual pixels have moved since last frame. History rejection is the deletion of previous information that is considered stale, for instance because of visiblity changes.

It is typical to apply a cheaper anti aliasing algorithm on top of TAA, such as FXAA, which effectively produces one spatial pass and one pass using temporal information. This, though, can result in overblurring or ghosting artifacts. To build a well performing neural network architecture, we want to allow the network to combine these two data streams, spatial and temporal, and create an output based on both at once instead of two separated passes. The reference images for training the networks are produced from renderings at higher resolutions that are downsampled. To gather multiple samples per pixel, pixel jittering is always used, i.e. the subpixel position during rasterization or ray casting is offset from the center. We discuss this in more detail in Section 5.5.

5.3.1 Temporal Accumulation

Motion vectors are obtained directly out of the renderer and can be computed using the camera projection matrix from the previous frame. Each pixel has a motion vector $m = (x, y)$ representing the offset of that pixel from the previous to the current frame measured in decimal point pixels. The equation of the reprojected color values R as a function of the input image I is:

$$R_{i,j} = I(i - m_{i,j,x},\ j - m_{i,j,y}). \tag{5.1}$$

This reprojection queries arbitrary positions in the input image, requiring interpolation. By default, most frameworks offer bilinear interpolation. Due to the artifacts caused by bilinear interpolation, we decided to use Catmull-Rom interpolation as a higher order interpolation scheme. This is a good compromise as it removes most artifacts from bilinear interpolation but does not introduce ringing and is relatively cheap [YLS20]. In our case, the motion vectors are first computed from the difference between the current view's depth buffer position, and its backprojected version in the previous position's camera UV space. All other specific motions overwrite the existing motion vectors. Typically in TAA, the color framebuffer of a new frame C is blended with the reprojected history buffer R using an exponential moving average [Yan+09] as

$$I_{i,j} = \alpha \cdot C_{i,j} + (1 - \alpha) \cdot R_{i,j} \tag{5.2}$$

where α is a constant blend factor. It is possible to vary alpha per pixel, for example for rejecting a deprecated history buffer by setting $\alpha = 1$. More commonly, the history buffer R is rectified using clamping or clipping with a bounding box of neighborhood samples from C in color space to prohibit ghosting artifacts from occlusion or illumination changes [Ped16; YLS20]. Variance Clipping TAA, for example, adapts the size of the bounding box based on the first two moments of color samples in the local neighborhood of pixels [Sal16].

The reprojected information creates an image that can be merged with the current frame. In our work, the merge operation is the final step of the network. The network learns to process these different images so that averaging them produces the desired output. The core of this technique is finding the correct kernels to process the images, called Kernel Prediction.

5.3.2 Kernel Prediction

CNNs operating on image data usually directly predict pixel colors. This sometimes causes the networks to "dream up" features and may induce artifacts like color shifts and lead to longer training times.

With kernel prediction [KBS15; Tho+20], instead of generating a color buffer, the CNN outputs one or more K-sized per pixel kernels. These replace the constant α in Equation (5.2) and are used in a fixed function step to compute a weighted sum of neighboring pixels in several buffers as the final color output I. This limits the network to only transform inputs using existing color information. It is desirable to allow the network to mix multiple input buffers using $n = K \times K$ weights per kernel, requiring several per-pixel kernels. For example, two kernels can be used to interpolate colors from the new frame's color samples C and the history buffer R:

$$I_{i,j} = \sum_n \alpha_n^C \cdot C_{i+dx_n, j+dy_n} + \sum_n \alpha_n^R \cdot R_{i+dx_n, j+dy_n} \ , \tag{5.3}$$

where

$$(dx_n, dy_n) \in \left\{ x \mid x \in [-\lfloor \frac{K}{2} \rfloor, \lfloor \frac{K}{2} \rfloor] \right\} \times \left\{ y \mid y \in [-\lfloor \frac{K}{2} \rfloor, \lfloor \frac{K}{2} \rfloor] \right\}$$

and the

$$\sum_n \alpha_n^C + \sum_n \alpha_n^R = 1.$$

This leads to the combined processing of spatial and temporal filtering that was motivated earlier. The choice of the different inputs is crucial to the performance of the network. The final trainable layer of the network must output the set of all weights α per pixel. This layer is memory intensive as an entire frame of size $W \times H$ pixels accumulated with N K-sized per-pixel kernels requires the generation of $W \times H \times K \times K \times N$ interpolation weights.

5.4 CNN with kernel prediction

TAA typically makes use of motion vector and depth buffers. These are used for history rejection to drop an accumulated pixel color from the history buffer that no longer belongs to the same surface. Figure 5.2 shows the architecture of our network which uses the same data as input. The multi-layer convolutional neural network is embedded in a feedback loop that feeds the new frame's output color as an additional input to the next frame's processing. This feedback loop is implemented with an recurrent neural network cell (RNN) that we detail further down.

The main network consists of four convolutional layers that produce two 5×5 sized filter kernels per pixel (red in Figure 5.2). These kernels are used to compute a weighted sum of pixels in a local neighborhood from the current frame's color input and the previous frame's reprojected color output according to Equation (5.3). Before generating the kernels in the final layer, the network also outputs a five layer sized state carried over to the next frame's processing alongside the RGB output (blue). In the following, we present the stages of the network in detail.

Network Input

The kernel prediction network receives the currently rendered RGB frame in low-dynamic range, normalized to values between 0 and 1. The network works better with values in this range compared to the $[0, 255]$ range. These images are concatenated with per pixel motion vectors and depth values. Additionally, a feedback loop concatenates eight additional channels from the network output of the previous frame to this input. These feedback channels consist of the previously anti-aliased RGB output and five freely trainable channels that are conceptually equivalent to a hidden state of an RNN.

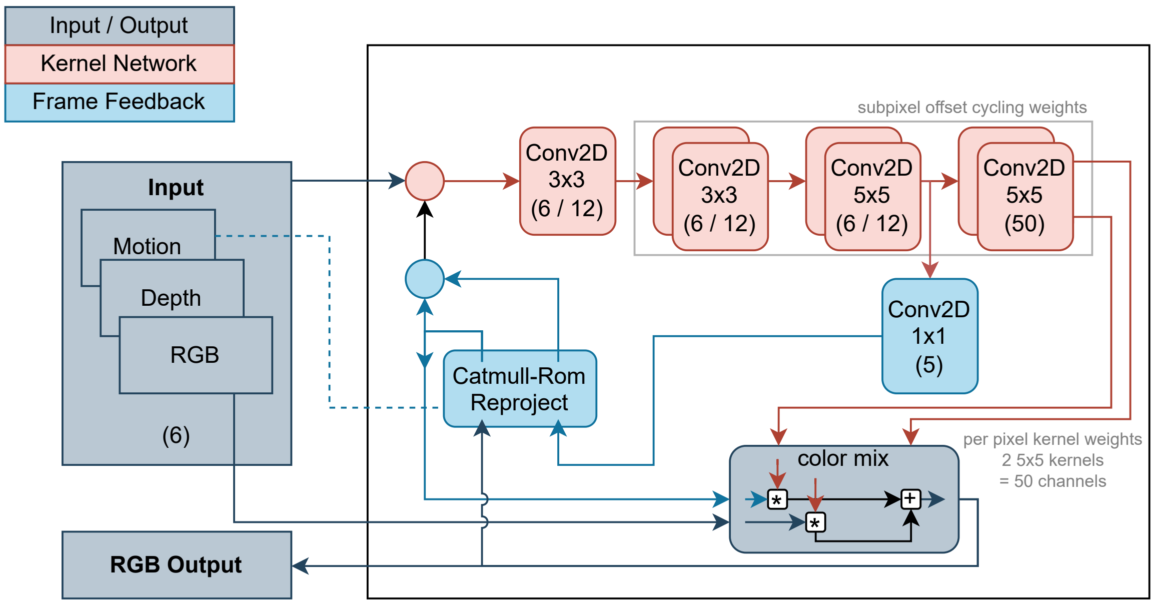


Figure 5.2: Architecture of the TAA network with layer counts in parentheses. The reprojected feedback uses Catmull-Rom interpolation and is implemented as a recurrent network layer. Three of the stages use a different set of layer weights depending on the Halton 2,3 index that is used for subpixel offsetting during rendering. The final layer outputs two 5×5 filter kernels applied to the new RGB buffer and previous network output.

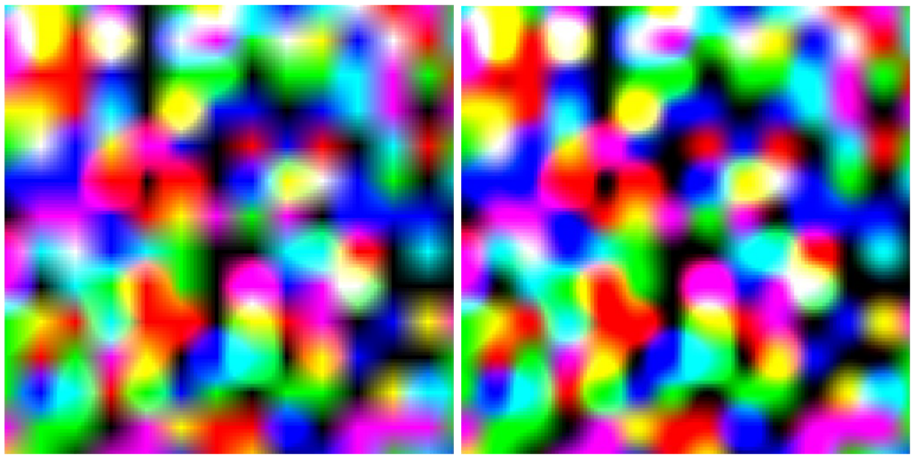


Figure 5.3: Left: Bilinear interpolation. **Right:** Catmull-Rom interpolation.
Images generated by taking a random set of pixels and interpolating them with one or the other implementation. This problem happens in our pipeline during the reprojection. The bilinear interpolation scheme on the left presents very noticeable artifacts, notably lines appear.

Temporal Feedback
Before the last convolutional layer of the kernel prediction network generates the per-pixel kernels, we use a small 1×1 convolution to output five additional channels (blue in Figure 5.2). In addition to the network's RGB output, these are carried over to the next frame to store additional temporal information. In the next frame's processing, we use the motion information from the renderer to reproject these feedback channels. The reprojection is performed with Catmull-Rom interpolation. This interpolation scheme uses cubic splines with precomputed coefficients using the 3×3 pixels in the neighborhood. This reduces typical bilinear interpolation artifacts (see Figure 5.3), and improves frequency content in the images, which is especially important for convolutions. The code for this part is released separately as a side contribution of this project, as it is not available in Tensorflow. The Catmull-Rom implementation typically used in production is orders of magnitude faster than ours. This has been investigated in the past (e.g. [Kar14] for Catmull-Rom, [SN15] for other kernels). Notably, they found ways to use only 5 texture reads to obtain a very good approximation, with significant performance improvement. To the day of writing, no optimized implementation is available in the Tensorflow framework. Because this has been extensively covered in the literature, the optimization of this algorithm is out of the scope of our paper. We discuss the performance implications of this choice in a later section. The RNN cell functions as a way to feed the network its own output. Note that using a loop or successive applications of the model is a valid approach. Choosing the RNN format allows us to use well documented functionalities such as different training window sizes, unrolling, managing hidden states and stateful mode. The total RNN implementation's hidden state consists of the current anti-aliased RGB output after the *color mix* operation (detailed just after) and the five freely trainable channels. Additionally, we keep track of the currently processed index of the Halton 2,3 sequence that the renderer uses for subpixel jittering.

We unroll the network for training, but for inference a stateful implementation that iteratively processes one frame after the other is used. We observed a noticeable difference in final output quality between stateful and non stateful, the stateful version performing better in both quality and processing time.

Kernel Prediction Network

The kernel prediction network (red in Figure 5.2) consists of four fully convolutional layers which use zero padding to retain the size of the feature maps. We evaluate the network in two variants using either six or twelve output channels in the first three layers. The last of the four layers outputs the two 5×5 filter kernels. Those kernels are later used in the kernel-weighted *color mix* operation to compute a weighted sum of the pixel neighborhoods in the RGB buffers from the currently rendered frame and the previous anti-aliased network output. The output size of this final layer is equal to the number of elements in both kernels. We use a ReLU activation after each layer except the last one, wkere we apply a sigmoid activation to normalize the sum of all kernel elements for a given pixel to 1. The hidden layer with a 1×1 convolution uses no activation function. We propose a method to let the network adapt to the cycle of jittering subpixel positions from the renderer: The last three layers of the kernel prediction network use eight different sets of weights $W_0 \ldots W_7$. For each index in the jitter sequence j, the same set of weights W_j is always used within the layers. Note that this duplication does not increase the number of operations in the network as only one set of weights is used at a time. Thus, instead of introducing new computation paths this only leads to a slightly higher memory consumption to store the seven additional sets of weights. We show the impact of these sets of weights in Section 5.6.

Note on U-Nets

U-nets are neural network architectures based on infering smaller and smaller layers, and then recreating another image of the original size. This architecture uses convolutional neural networks for infering smaller or larger buffers. In the first step, an image I_0 of size $N \times M$ is used as input and the output in an image I_1 of half the size, $\frac{N}{2} \times \frac{M}{2}$. This is performed several times, usually until the size is of the order of 32×32. On the reconstruction step, convolution layers use two images as inputs: I_i and I_{i-1}, but the smaller image I_i is first upsampled to the size of I_{i-1}. At the end we obtain an image of the original size with sometimes more or less channels. The advanage is that the information has the opportunity to propagate to the entire frame. This is more flexibly done this way than by setting a very large kernel size and a unique convolution pass.

One caveat is that many of these operations are sequential, hence introducing a definite bottleneck in the application. U-nets are rarely used to output image frames, but usually used for image processing techniques such as image segmentation and annotation. In our case, it failed to provide a reliable way to enhance the quality of our network in three different scenarios.

In the first scenario, the U-net was given as input the current image buffer and the reprojected previous image, and was outputting its result as a frame to combine with the output of the kernel prediction network.

In the second scenario, the U-net used the output of the first layer as input (in part or whole). Its output was given to the kernel prediction layer.

In the last scenario, the U-net was given is input the current image and the reprojected previous image. Its output was then given to the kernel prediction layer.

In all attempts, the network performed significantly worse in terms of quality. The performance impact was around 5 to 10%, which could likely be optimized better than our implementation, since we had to rely on our catmull-rom implementation for the upscaling. We did not find it interesting to optimize it in any way given the poor results. The network oversmoothed the details and created small blocky artifacts.

We decided against using these architectures in our model.

5.5 Training

5.5.1 Dataset

The training data is made of renderings of camera flythroughs in 3 scenes at a resolution of 2560×1440 pixels. These images are resized to 640×480 pixels using a Blackmann-Harris filter to create reference images without aliasing. Renderings from the same flythroughs at a native 640×480 $@24$ resolution are used as training input data. The image sequences contain successions of smooth and sudden movements at different speeds. The scenes have been chosen to contain difficult cases for anti aliasing such as high-frequency geometry, frequent visibility changes, and challenging low-frequency aliasing (almost vertical doorframes etc).

We used 3 different scenes. Staircaqse contains doors, stairs with empty space between each step, a thin railing, large columns and wide glass windows, projecting shadows. Cafeteria contains very thin geometry (wires) that are not properly resolved by the input, (but the reference properly does), straight edges with tables and chairs and difficult occlusions with the cafeteria's lampshades. Finally, the livingroom contains designer chairs that have a thin black stripe, a couch and a textured floor.

The renderer cycles through 8 different subpixel jitter offsets from a Halton 2,3 sequence. For that reason, training happens on a windows containing 8 frames, ensuring that all jitter indices were considered in the training step once. In particular, this is relevant so that backpropagation is always carried out on each of the 8 different sets of weights that we use in the last three convolutional layers of the kernel prediction network (see Section 5.4). During training, all sequences within a batch start with the same jitter index for performance purposes, but different batches start with different indices.

5.5.2 Data Processing

The network is trained on patches of 8 consecutive frames with a size of $p_w \times p_h = 128 \times 128$ pixels that are extracted from the dataset sequences. Our data is augmented using channel switching, and randomized patch selection. All possible starting positions are precomputed, then shuffled which ensures that the network processes the entire dataset prior to any repetition while the order of the input varies. For each of the $n_j = 8$ different jitter positions, we store a separate dictionary containing all possible training patches that start with the respective subpixel jitter index. For a sequence of $n_f = 100$ frames and resolution $w \times h = 640 \times 480$ pixels, $(w - p_w)(h - p_h)(n_f - (n_j - 1)) = (640 - 128)(480 - 128)(100 - 7) = 16,760,832$ patches are distributed over these eight dictionaries. We use several sequences, totaling more than 500 frames. The number of possible starting positions is so large that the processing of all patches far surpasses available training time. This motivates us to define an epoch as 1600 processed patches.

Neural Networks or machine learning models are susceptible to overfitting. Main causes of overfitting in our case could be models with too many parameters or degrees of freedom, low data variety, long training times. In our case, the network typically trains on less than one percent of the total training data but still performs very well on the entire dataset. Our parameter count is very low compared to the complexity of the task and the diversity of the dataset. This leads us to believe that our network is at very low risk of overfitting, and in practice, we did not notice any overfitting.

5.5.3 Training Loss

The loss for training a temporal anti aliasing network has to account for several effects. We penalize undesirable artifacts and reward closeness to the reference. Our general loss function is split into three parts as known from other image reconstruction networks [KBS15; Tho+20; Has+20]. Different weights control the influence of the individual losses:

- The *reference loss* L_r is the sum of all pixel-wise absolute L1 differences between the network output and the reference image.
- For the *gradient loss* L_g, we compute this difference on the gradient images of the reference and output images in x and y direction.
- Additionally, we compute a *temporal loss* L_t as the pixel-wise difference of temporal gradients in the reference and the network output. L_t is used to reduce flickering and other temporal artifacts.

We also experimented with using Nvidia's ꟻLIP [And+20] as a training loss. ꟻLIP is a perceptual error metric. We integrated the ꟻLIP algorithm into our pipeline and noticed that it was not possible to use as it was very slow. We made some core improvement in the code that was acknowledged by Nvidia. We also shared our mathematical derivations with them.

The ꟻLIP metric uses values from the monitor's resolution and the viewing distance to compute an error metric corresponding to the accuracy of a standard eye. This is computed using different mathematical models, involving convolutions with gaussian kernels. We showed that all the kernels could be separated into one dimensional convolutions, reducing the scaling of the algorithm from $O(n^2)$ to $O(n)$ in the number of pixels, with an advantageous constant factor. This resulted in acceptable performance for our use case. We then tested this error metric in combination to the above losses. This resulted in a consistent blurring of the output. After analysis, it turns out that the blurring introduced by the ꟻLIP metric spreads the gradient to multiple pixels, preventing the learning of sharp features as can be seen in Figure 5.4

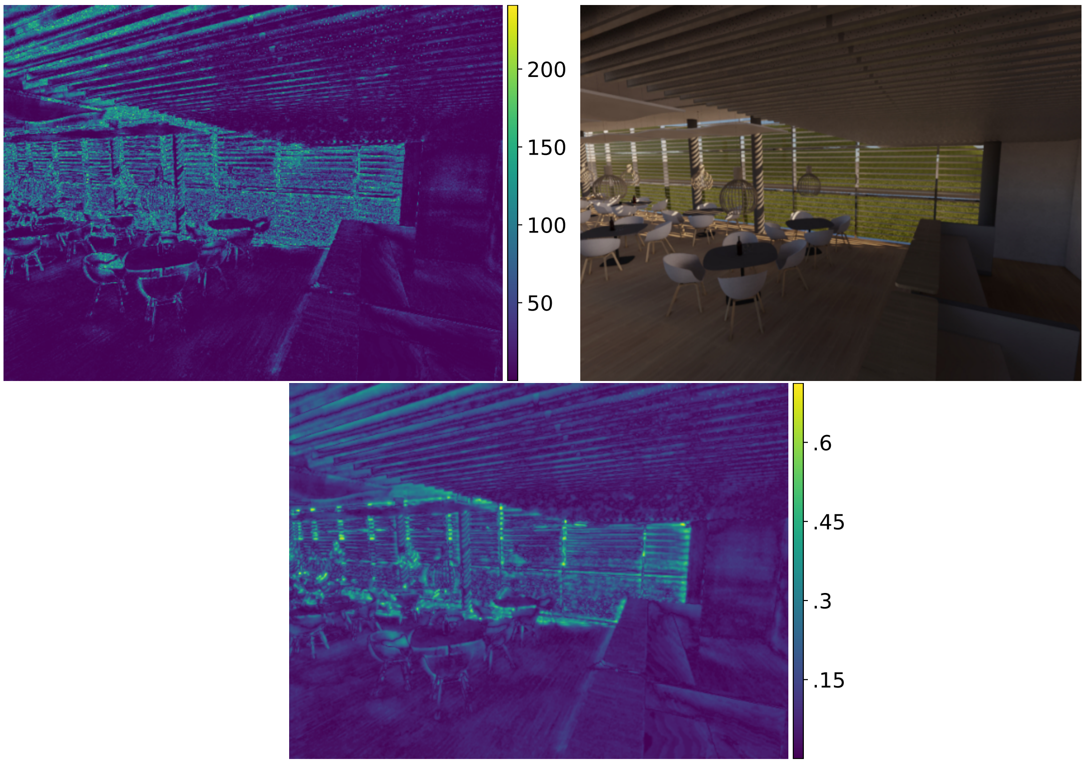


Figure 5.4: Left: ꟻLIP metric **bottom:** RMSE **Right:** Reference image
The FLIP metric visibly removes the sharpness in the data, information still present in RMSE. The RMSE has a wider range of values and better discriminates between slight discrepancy and bad value.

The parameter pixel_per_degree was tested on a sensible range of values and we observed no obvious benefit of using ꟻLIP for training rather than other metrics like RMSE. We therefore decided against using this metric in our work.

We introduce a per pixel aliasing mask M for the training images which contains a value between 0 and 1 per pixel. Values closer to one identify pixels containing aliasing artifacts. The mask is computed by summing up all RGB channel-wise differences between the training input and reference images. Additionally, we multiply the mask with a sensitivity parameter and clip values to $[0, 1]$. As a last step, a 3×3 dilation is applied to guarantee that all pixels of aliasing regions are considered. We use this mask to create two versions of our three loss functions each by weighting them either with M or $(1 - M)$. The first identifies loss in regions containing aliasing which correlates to the network lacking anti aliasing properties. The latter identifies loss in regions were the input and reference images were already close. Here, the network should at best avoid any alteration of the input data at all. A high loss in these regions usually stems from artifacts like ghosting or overblurring. We give all these parts different weights when computing the final loss.

We choose these weights to adjust the influence of each loss, allowing us to study a balanced situation or a situation where one loss dominates the others. This reweighting is necessary to consider all losses in the backpropagation as L_t in particular is roughly ten times smaller than L_r and L_g. Finally, we adapt the loss given the aliasing mask M so that image regions containing aliasing are 1.5 times as much important as the non-aliased regions. Though we did not conduct a large scale hyperparameter investigation, all six loss weights could be fine tuned and even modified through training with dedicated frameworks. Since we train on patches consisting of 8 frames, we compute the loss only on the last output frame to let the network accumulate temporal information. The training network uses an Adam optimizer [KB14] with $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-7}$ and clamping gradients to 1. Our initial learning rate of 0.001 follows an exponential decay with 1000 decay steps and a decay rate of 0.96. We train on a batch size of 24 where each batch element consists of 8 successive frames of the same portion of the image in subsequent frames.

5.6 Results

We evaluated the network on three different scenes. A large staircase with many long and thin straight lines. A cafeteria, with difficult thin geometry on the lamps. A studio with some textures and a transition through a glass window. We compare our results with variance clipping TAA and with our reference. See also our supplementary video.

In Figure 5.5, we present a direct comparison between our method, variance clipping TAA [Sal16] and the input including root-mean-square errors (RMSE). Our method retains crisp edges without overblurring.

We did not detect ghosting artifacts using our method. Figure 5.6 shows that variance clipping TAA (VCTAA) tends to blow up thin features during fast movement, while our method retains clearer edges.

The flickering produced by our final network is reduced and even better than the reference. The flickering in the reference comes from the large amount of occlusions and

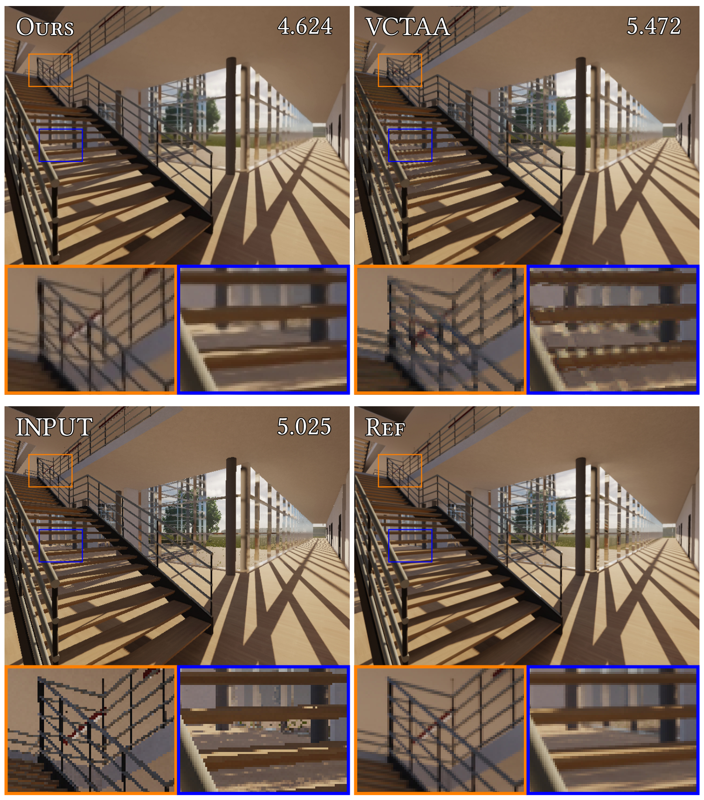


Figure 5.5: Top Right number: RMSE. The edges are very consistently antialiased without blurring (**Top Left**) while vctaa noticeably degrades the sharpness in the orange closeup . The ghosting artifact on the lighting of the stairs (**Top Right**, blue inset) is not present in our method.

Layer	K	I	O	**Total**
layer 1	3	11	12	2388
layer 2	3	12	12	2604
layer 3	5	12	12	7212
kernel	5	12	50	30050
hidden	1	12	5	125
total				42379

Table 5.1: Computation details for the number of FLOPs of the network itself (upper bound). 40k FLOPs/pixel

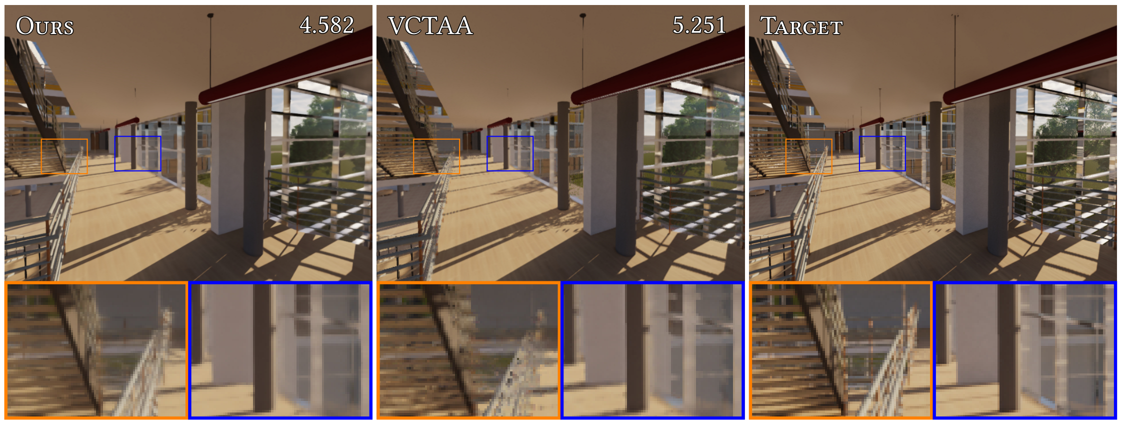


Figure 5.6: Numbers are RMSE. Comparison of behavior during fast camera movement. Our method resolves correctly the railing and stairs that are distorted by VCTAA.

resolution	Catmull-Rom	model	color mix
2560x1440	350.241	2.892	9.204
1920x1080	264.871	2.827	8.789
640x480	180.294	2.368	8.841

Table 5.2: In milliseconds (ms). Our Tensorflow implementation of Catmull-Rom is comparably unoptimized. The color mix combines the 50 output channels with the the input image and the reprojected previous frame. Measured on an RTX 3060 with Tensorflow python framework using 32bit computation.

disocclusions happening quickly on a relatively low resolution. Our network creates a much smoother result. The numerous occlusions happening in the overlapping stairs and railing area produce high temporal frequencies that our network has been trained to reduce via the temporal weights in our loss function.

Our unoptimized implementation takes almost 200ms per frame, Table 5.2. This processing time is dominated by our naive implementation of the Catmull-Rom interpolation, as this operation was not natively supported by Tensorflow. The inference time is remarkably fast with no more than 3ms. This is before any optimizations such as TensorRT [Nvi23], which performs remarkably well on convolutional networks, offering more than $6\times$ speedups according to documentation, and using reduced precisions, e.g. by using half floats where applicable (half floats should not be used with sigmoids). Our feedback loop implementation as a recurrent neural network (RNN) creates compatibility issues with TensorRT which can be resolved by replacing the RNN with a simpler cell that feeds the data of the previous frame to the next. We believe that our proof-of-concept network can be implemented in real-time applications if modest optimizations can be performed, notably on the Catmull-Rom interpolation.

We computed the total number of operations required for each layer, following the equations for counting multiply-add-accumulate and add operations:

$$\begin{aligned} MAC &= K^2 * I * O \\ ADD &= O \\ FLOPS &= (2 * MAC + ADD) * W * H \end{aligned}$$

with the kernel size K, the number of input and output channels I and O, and W, H the width and height of the image. We evaluate the same network we used for the measurements in Table 5.1.

With our smaller network, with six internal channels, the count drops to $18740 * W * H$. The activation functions (ReLU) of the first layers and the softmax of the final layer add in total 2000 operations per pixel, largely dominated by the cost of exponentiation. The color mix operation takes 100 operations per pixel. The Catmull-Rom interpolation requires less than a 100 operations per pixel. Better and faster alternatives to Catmull-Rom interpolation exist, we only used it as a proof of concept. The total number of floating point operations is below 50000 per pixel. Despite being a sizeable amount it is by no means out of reach for a well-optimized pipeline. The vast majority of these operations is in the kernel prediction layer, which suggests that adopting another strategy could result in great improvement in performance.

5.6.1 Jitter

The absence of subpixel jitter specific sets of weights for the last three convolution layers heavily reduces the quality of the anti aliasing as shown in Figure 5.7. The previous frame being reprojected has a different jitter pattern than the current frame, causing the network to improperly combine them. The network would need to adapt to each different combination of jitter patterns and associated motion vectors. Having one set of weights per jitter position removes this burden entirely from the network. These images have been generated by networks with an equal number of training epochs. The setup on the top right (2-3-K-H) requires more training to perform at least as well as the bottom right image (2-3-K) as shown by their lower RMSE. This also suggests that modifying the information the network chooses to keep for the next iteration is not a fruitful direction. Our method of using a different set of weights for each respective camera subpixel jitter position in the convolution layers performs significantly better than using only one set of weights (top left). These results hint at an even better way to perform these operations: making the network decide which jitter to use for the next iteration. This future work idea is further detailed in the discussion section.

5.6.2 Internal Layers

The number of internal layers and their size plays a big role in runtime performance. These layers gives the network more parameters for modifying and spreading the information accross the frame. This propagation is important for properly resolving low-frequency content (such as the slightly slanted doorframe). Even for resolved edges, more layers can

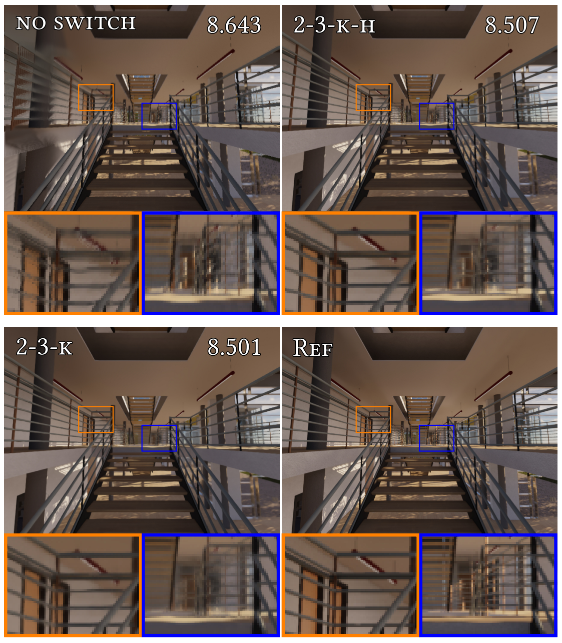


Figure 5.7: Top right number: RMSE. Comparison between switching different layer weights with the jitter cycle and no-switching. 2-3-K-H corresponds to changing the layers 2,3, the kernel prediction layer and the hidden layer.

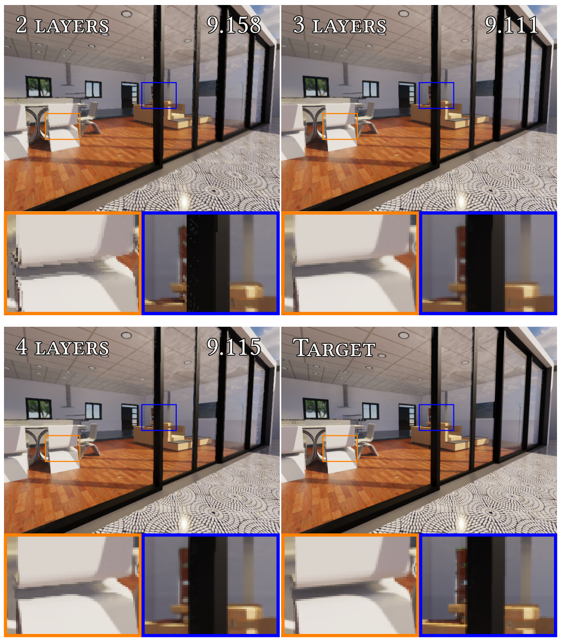


Figure 5.8: Top right number: RMSE. Different layer count with equal training epochs. More layers take longer to train and take longer to compute. Less layers lead to subpar quality. The quality is only very marginally improved by using 4 layers.

improve the sharpness or reduce artifacts. However, chaining layers implies introducing computation dependency, in the form of sequential execution which can make the process slower. Bigger layers on the other hand, benefit fully from parallelism. In our experiments, two layers were insufficient for a well-performing network, while improvement were quite marginal above three.

5.6.3 Hidden Layers

In addition to the anti aliased output, the network's feedback loop contains a trainable state that is updated every iteration. The update step has its own layer through which information is stored in the per pixel state given the input data (blue convolutional layer in Figure 5.2). This memory area has the size of a frame, but has as many channels as we desire. Giving the network enough state space is crucial, but a larger state increases inference time and memory consumption. Our analysis of the hidden layer channels reveals that they essentially consist of various edge detection filters, seemingly decomposing the image, as can be seen in Figure 5.9 which displays three of the state's channels. An exact interpretation is impossible, but these images offer us a better understanding of what information these channels are used for, and therefore what information the network may

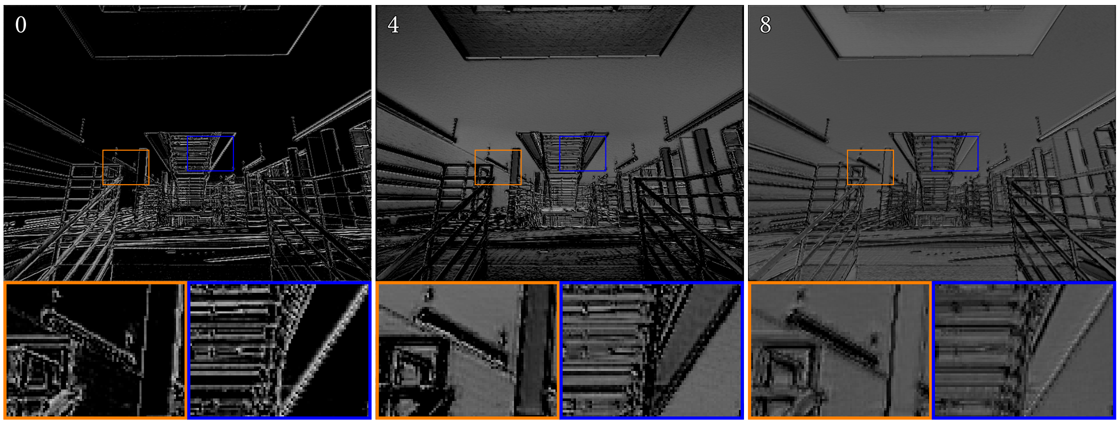


Figure 5.9: Left: Resembles a directional edge detection filter, the lines on the ceiling can help grasping what function is at play. **Center:** Another edge detection, but combined with a depth gradient although it is remapped in some way. Note that these two images are quite dark. **Right:** Seemingly another edge detection but also flattening the non edge parts of the scene.

lack in case one reduces the amount of available channels. In our experiments, using more than five channels did not offer significantly better quality.

5.6.4 Loss Weights

We tested different loss weight values (more details: Section 5.5.3, see Figure 5.10). The training was performed on an Nvidia Titan V, and took 8h. One noticeable finding is that having a dominant temporal loss increases image stability and reduces flickering, even to levels better than the reference. In our testing we found that having the temporal loss overall 20% higher than the aliasing loss produces better results. Temporal effects are difficult to convey using words, so if our descriptions are not enough, we suggest watching the accompying video, notably the ascent of the second half of the staircase. During the ascent, the pillars behind the railing bars on the top of the image can be seen quickly popping in and out, as well as the railing itself disappearing sometimes. The geometry behind the railing and stairs creates similar patterns. These fast occlusion-dissoclusions are typical of under-resolved thin geometry. Our approach improves both of these issues, producing a smoother experience. That being said, our approach does not eliminate this phenomenon entirely, and this is an area where a lot of improvement is possible.

5.6.5 Discussion

Our method only focuses on anti aliasing, but in most cases, several post processing techniques are performed on the output. It was our choice to not include denoising or upscaling, but it remains a limitation of this method for its adoption in industry context. It could very well be adapted to perform both denoising and anti aliasing. This could benefit from mutualizing computing resources as these networks being one and the same would surely perform better than two separate processes.

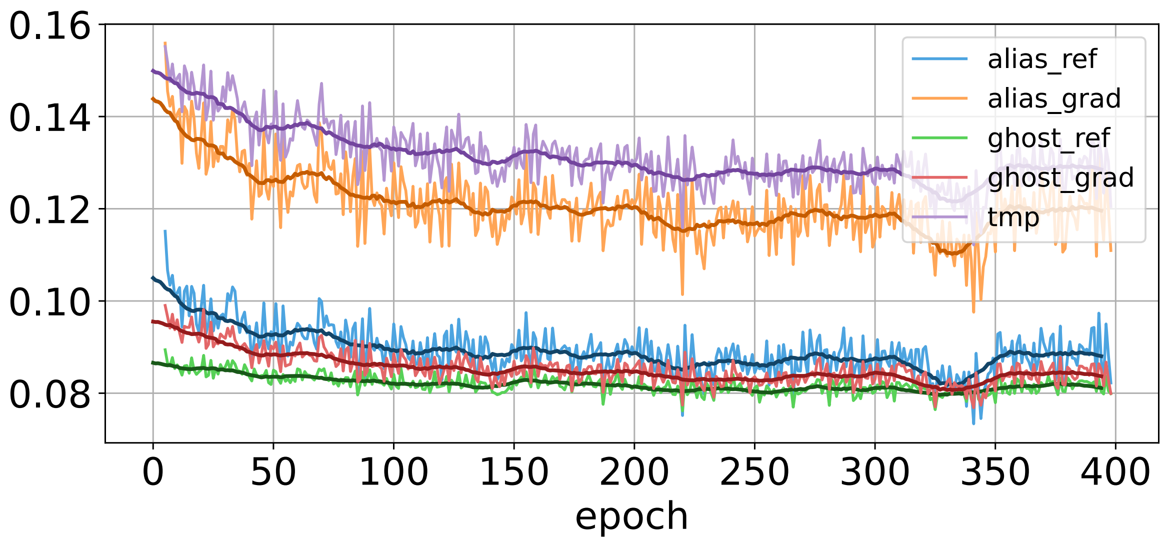


Figure 5.10: Weights are adjusted for the temporal loss (tmp) to be the most impactful, followed by the gradient loss. We noted better results by having the gradient loss being higher than the reference loss. The *alias* prefix refers to the loss in image regions with aliasing, while the *ghost* prefix denotes loss on regions without aliasing which typically occurs because of ghosting artifacts. Note the presence of a dip around epoch 340. We could not find a good explanation other than training data random variation.

One weaker point of this technique is texture reconstruction. We noted a slight loss of quality on complex and slanted textures. It is quite common for upscaling or TAA to use a sharpening pass at the end. The result would surely be slightly improved, but a deeper solution should be found for making this kind of method truly reliable for more complex textures. It is to be noted that the training dataset did not contain many different examples of complex textures. That is one limitation of our work, and taking extra precautions to include interesting textures in the dataset is advised for future work.

Our method improves the quality of low frequency image features, such as doorframes that are almost vertical, or slightly slanted objects in general. Low frequency context is very difficult to deal with for almost all anti aliasing techniques because of the use of regular grids. For this reason, the helpfulness of camera subpixel offset jittering is limited in traditional anti aliasing. Since the jittering happens identically on all pixels within the same framebuffer, the frequency content only changes after each frame. Exponential moving averages and neighborhood rectifications, as commonly used, cannot capture these changes efficiently enough. Applying different subpixel offsets to pixels even within the frame would allow for a better distribution of samples and for a better subsequent reconstruction for most of these cases. One such possibility is full raytracing, which would allow the use of low discrepancy sequences or even optimized sequences such as step blue noise. Indeed, certain variants of blue noise are provably the most efficient at reducing spatial aliasing for a given sample budget. One major advantage of step blue noise is its behavior at the resolving limit. Typical sampling strategies produce aliasing artifacts as seen in the introduction: new frequencies emerges that are not in the data. Step blue noise instead perfectly resolves when it can resolve and produces noise when it cannot. This is one edge or the other of the step.

One other avenue for future work would be to use a neural network directly on the raytracing output (ray position in frame and rbg color/depth), bypassing the filters that put samples into pixels. This neural network could then perform both anti-aliasing and denoising on the raw data, and output a pixel grid. The network could also predict optimal ray positions for the next frame. This would be an almost all-in-one network, rendering being the only part still done by classical algorithms.

5.7 Conclusion

We presented a small and concise convolutional neural network architecture for temporal anti aliasing. Our kernel prediction strategy prevents bigger artifacts from forming our architecture leverages higher quality reprojection by using the catmull-rom interpolation method, rather than the usual bilinear interpolation. Our analysis compares different variations of the network on different dimensions. We show that using different sets of weights for certain layers provide significant quality improvements, without additional computation, only a small amount of memory. We tested our method on different scenes and provided video samples and comparisons. Notably, we tested difficult scenarios with many difficult occlusions and thin geometry. Our method provides next to no artifacts while providing visibly improved image quality. We also suggested to take this research deeper into the real-time raytracing pipeline by having the network suggest sample positions and perform the reconstruction step. This would be an almost all-in-one Neural Network, capable of denoising, anti-aliasing and adaptive sampling.

6 Conclusion

Rendering as a field is in the spotlight in recent years since the introduction of raytracing hardware in consumer products. This induced a renewed interest for physically based rendering notably in video games. This sparked a wide interest in the research community for research in real-time domains. We showed in our work that carefully choosing parameters for these techniques is not enough. Indeed, automatic self configuring strategies are beneficial for optimal performance. Our work highlights the importance of these strategies in different places of the rendering pipeline: in intersection tests, core rendering sampling techniques and post processing.

Chapter 3 introduces a Hybrid Online Autotuning framework that combines traditional autotuning techniques with machine learning models to configure the parameters of acceleration structures. One of the main goals in this context is to remove the need for rebuilding acceleration structures, cost incurred in typical autotuning. We developed scene indicators that describe the scene. We train machine learning models to map these indicators to optimal acceleration structure parameter settings. Two model types were explored: a nearest-neighbor lookup and radial basis functions model.

The nearest-neighbor method achieves 95% of the performance offered by the search-based tuning while reducing the overhead by 90% of the rendering time. For the simpler case of optimizing only rendering performance, both models provided speedups around 1.04-$1.05\times$ over baseline. When also considering acceleration structure build time for real time applications, the nearest-neighbor approach gave a promising $1.12\times$ geometric mean speedup.

Our extensive analysis of BVH parameter behavior allowed us to form recommendations for strategic use of autotuning. Indeed, different building algorithms may have less interesting parameters that are better left untouched, because they are required in other parts of the code, or because they provide little improvement compared to the cost of autotuning convergence.

This work highlights the potential of hybrid techniques that combine exploration-based search with learning-based exploitation of prior knowledge. As the time constraints grows tighter, the benefits of deploying adaptive techniques will only continue to grow as well.

Chapter 4 describes a novel autotuning approach for optimizing the parameters controlling Markov Chain Monte Carlo (MCMC) path mutation strategies in algorithms like Primary Sample Space Metropolis Light Transport (PSSMLT). Instead of going through the process of finding good parameters for a given strategy, we propose a way to automate this process while rendering is ongoing. Our method uses an out-of-the-loop autotuner iterating over different configuration and converging on better ones. We used the step size in PSSMLT for demonstrating that our method indeed provides relevant benefits.

Instead of using a single scaling parameter to determine the step size of MCMC path perturbations, we introduced σ-binning where multiple step size parameters are used. We select the step size adaptively based on the local probability density.

The autotuner then optimizes these parameters by iteratively rendering with different configurations. The autotuner uses a guiding metric such as the acceptance ratio or the rendering time. Additionally, the wrapped normal distribution typically used for MCMC mutations was replaced with a truncated normal for more efficient probability computations.

Results on complex scenes with difficult lighting demonstrated that the adaptive approach provided faster convergence in terms of RMSE compared to using a single fixed parameter. Visually, reduced noise and artifacts were observed for equal sample counts, likely due to the method's ability to better explore important path contributions by adapting step sizes to the underlying BSDF.

Finally, Chapter 5 tackles the problem of aliasing artifacts that are notorious in most renderers. Classic techniques like supersampling and multisampling provide brute-force solutions that are not always desirable. Temporal Anti-Aliasing (TAA) is a relatively simple and sometimes cheaper alternative. However, traditional TAA methods still exhibit ghosting, flickering, and overblurring issues. Our method makes advances on several of these issues.

Given that recent advances in the field use closed source neural networks, we attempt at demistifying their approaches. Specifically, we build a fully convolutional neural network architecture trained to perform kernel prediction. Our network predicts per-pixel kernel weights that blend the current frame's colors with reprojected samples from previous frames. This strategy showed improvement over classic TAA by better preserving temporal stability while fully negating ghosting.

One particularity of our method is the use of multiple convolutional kernel weight sets that switch based on the camera subpixel jitter. There is no need for additional computation to handle the varying jitter, and the neural network can take fully advantage of the additional information it provides. The network also outputs a hidden state that is passed down to the next iteration, acting as a sort of memory.

The natural next step for such a neural method is now to unify steps like anti-aliasing, denoising, and maybe also upsampling into a single framework. This direction aligns with broader trends in rendering, where neural networks are increasingly integrated into the graphics pipeline to replace or augment traditional techniques as seen with Nvidia's DLSS. The future of rendering and autotuning will likely involve deeper synergy between learned models and traditional methods, enabling better quality for a similar time budget. Currently, some video games already use sophisticated shader optimization techniques. They diffuse different version of the shaders on many platforms and retain the ones that performs best, giving hardware-optimal shaders. A key issue common to all of these techniques is the lack of interpolation capabilities. Indeed, these approaches are not building a model of the relation between hardware and shader performance. This could be a fruitful area of future research.

Additionally, the problem of interpretability in learned approaches remains open. Unlike predefined heuristics where human input is key, neural networks are essentially black boxes, making it difficult to diagnose specific failure cases. In the context of autotuning, the future may see more sophisticated reinforcement learning approaches that learn optimal tuning policies over time, reducing the need for hand-crafted heuristics or search strategies. Ultimately, the future of rendering and autotuning is moving toward more holistic, self-optimizing pipelines that can adapt in real time, leveraging both traditional graphics principles and the power of machine learning.

Bibliography

[Ake93] Kurt Akeley. "Reality Engine Graphics". In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. 1993, pp. 109–116. DOI: 10.1145/166117.166131.

[AMD20] AMD. "FidelityFX". In: *Www.Amd.Com/En/Technologies/Radeon-Software-Fidelityfx*. 2020.

[And+20] Pontus Andersson et al. ": A Difference Evaluator for Alternating Images". In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 3.2 (2020), 15:1–15:23.

[Ans+09] Jason Ansel et al. "PetaBricks: A Language and Compiler for Algorithmic Choice". In: *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38–49. ISBN: 978-1-60558-392-1. DOI: 10.1145/1542476.1542481. (Visited on 10/13/2016).

[Ans+14] Jason Ansel et al. "OpenTuner: An Extensible Framework for Program Autotuning". In: ACM Press, 2014, pp. 303–316. ISBN: 978-1-4503-2809-8. DOI: 10.1145/2628071.2628092. (Visited on 04/21/2015).

[AT08] Christophe Andrieu and Johannes Thoms. "A Tutorial on Adaptive MCMC". In: *Statistics and Computing* 18 (Dec. 2008). DOI: 10.1007/s11222-008-9110-y.

[Bao+16] Wenlei Bao et al. "Static and Dynamic Frequency Scaling on Multicore CPUs". In: *ACM Trans. Archit. Code Optim.* 13.4 (Dec. 2016), 51:1–51:26. ISSN: 1544-3566. DOI: 10.1145/3011017. (Visited on 04/11/2018).

[BGW13] P. Balaprakash, R. B. Gramacy, and S. M. Wild. "Active-Learning-Based Surrogate Models for Empirical Performance Tuning". In: *2013 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2013, pp. 1–8. DOI: 10.1109/CLUSTER.2013.6702683.

[BHH15] Jiří Bittner, Michal Hapala, and Vlastimil Havran. "Incremental BVH Construction for Ray Tracing". In: *Computers & Graphics* 47 (Apr. 2015), pp. 135–144. ISSN: 0097-8493. DOI: 10.1016/j.cag.2014.12.001. (Visited on 03/07/2019).

[BPC12] J. Bergstra, N. Pinto, and D. Cox. "Machine Learning for Predictive Auto-Tuning with Boosted Regression Trees". In: *2012 Innovative Parallel Computing (InPar)*. May 2012, pp. 1–9. DOI: 10.1109/InPar.2012.6339587.

[BS87] Petr Beckmann and Andre Spizzichino. *The Scattering of Electromagnetic Waves from Rough Surfaces*. Jan. 1987. (Visited on 02/26/2024).

[Bur+18] Brent Burley et al. “The Design and Evolution of Disney’s Hyperion Renderer”. In: *ACM Trans. Graph.* 37.3 (July 2018), 33:1–33:22. ISSN: 0730-0301. DOI: 10.1145/3182159. (Visited on 03/07/2019).

[Cha+17] Chakravarty R. Alla Chaitanya et al. “Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder”. In: *ACM Transactions on Graphics* 36.4 (July 2017). ISSN: 0730-0301. DOI: 10.1145/3072959.3073601.

[Cha12] Kuo-Hao Chang. “Stochastic Nelder-Mead Simplex Method - A New Globally Convergent Direct Search Method for Simulation Optimization”. In: *European Journal of Operational Research* 220.3 (Aug. 2012), pp. 684–694.

[Chr+18] Per Christensen et al. “RenderMan: An Advanced Path-Tracing Architecture for Movie Rendering”. In: *ACM Trans. Graph.* 37.3 (Aug. 2018), 30:1–30:21. ISSN: 0730-0301. DOI: 10.1145/3182162. (Visited on 03/07/2019).

[Cra+15] Cyril Crassin et al. “Aggregate G-buffer Anti-Aliasing”. In: *Proc. ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games.* i3D ’15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 109–119. ISBN: 978-1-4503-3392-4. DOI: 10.1145/2699276.2699285.

[CT82] R. L. Cook and K. E. Torrance. “A Reflectance Model for Computer Graphics”. In: *ACM Transactions on Graphics* 1.1 (Jan. 1982), pp. 7–24. ISSN: 0730-0301. DOI: 10.1145/357290.357293. (Visited on 02/26/2024).

[DHK08] Holger Dammertz, Johannes Hanika, and Alexander Keller. “Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays”. In: *Proceedings of the Nineteenth Eurographics Conference on Rendering.* EGSR ’08. Aire-la-Ville, Switzerland: Eurographics Association, 2008, pp. 1225–1233. DOI: 10.1111/j.1467-8659.2008.01261.x. (Visited on 03/05/2019).

[DK17] Ken Dahm and Alexander Keller. “Learning Light Transport the Reinforced Way”. In: *arXiv:1701.07403 [cs]* (Jan. 2017). arXiv: 1701.07403 [cs]. (Visited on 03/06/2019).

[Fan+21] Hangming Fan et al. “Real-Time Monte Carlo Denoising with Weight Sharing Kernel Prediction Network”. In: *Computer Graphics Forum* 40.4 (2021), pp. 15–27. DOI: 10.1111/cgf.14338.

[Fas+18] Luca Fascione et al. “Manuka: A Batch-Shading Architecture for Spectral Path Tracing in Movie Production”. In: *ACM Transactions on Graphics* 37.3 (Aug. 2018), 31:1–31:18. ISSN: 0730-0301. DOI: 10.1145/3182161. (Visited on 03/30/2024).

[Fic+21] Andreas Fichtner et al. “Autotuning Hamiltonian Monte Carlo for Efficient Generalized Nullspace Exploration”. In: *Geophysical Journal International* 227.2 (Nov. 2021), pp. 941–968. ISSN: 0956-540X. DOI: 10.1093/gji/ggab270. (Visited on 05/03/2024).

[Geo+13] Iliyan Georgiev et al. “Joint Importance Sampling of Low-Order Volumetric Scattering”. In: *ACM Transactions on Graphics* 32.6 (Nov. 2013), 164:1–164:14. ISSN: 0730-0301. DOI: 10.1145/2508363.2508411. (Visited on 02/27/2024).

[Gey05] Charles J. Geyer. “The Metropolis-Hastings-Green Algorithm”. In: 2005.

[Gor+84] Cindy M. Goral et al. “Modeling the Interaction of Light between Diffuse Surfaces”. In: *ACM SIGGRAPH Computer Graphics* 18.3 (July 1984), pp. 213–222. ISSN: 0097-8930. DOI: 10.1145/964965.808601. (Visited on 02/05/2024).

[Gru+16] Adrien Gruson et al. “A Spatial Target Function for Metropolis Photon Tracing”. In: *ACM Transactions on Graphics* 36.1 (Nov. 2016), 4:1–4:13. ISSN: 0730-0301. DOI: 10.1145/2963097. (Visited on 02/27/2024).

[Has+20] J. Hasselgren et al. “Neural Temporal Adaptive Sampling and Denoising”. In: *Computer Graphics Forum* 39.2 (2020), pp. 147–155. DOI: 10.1111/cgf.13919.

[Has70] W. K. Hastings. “Monte Carlo Sampling Methods Using Markov Chains and Their Applications”. In: *Biometrika* 57.1 (1970), pp. 97–109. ISSN: 0006-3444. DOI: 10.2307/2334940. JSTOR: 2334940. (Visited on 12/15/2022).

[Her+19] Killian Herveau et al. “Hybrid Online Autotuning for Parallel Ray Tracing”. In: *Eurographics Symposium on Parallel Graphics and Visualization* (2019), 10 pages. ISSN: 1727-348X. DOI: 10.2312/PGV.20191110. (Visited on 11/06/2023).

[Her+21] Killian Herveau et al. “Analysis of Acceleration Structure Parameters and Hybrid Autotuning for Ray Tracing”. In: *IEEE Transactions on Visualization and Computer Graphics* 29.2 (Feb. 2021), pp. 1345–1356. ISSN: 1941-0506. DOI: 10.1109/TVCG.2021.3113499. (Visited on 11/06/2023).

[HJ11] Toshiya Hachisuka and Henrik Wann Jensen. “Robust Adaptive Photon Tracing Using Photon Path Visibility”. In: *ACM Transactions on Graphics* 30.5 (Oct. 2011), 114:1–114:11. ISSN: 0730-0301. DOI: 10.1145/2019627.2019633. (Visited on 12/16/2022).

[HM08] W. Hunt and W. R. Mark. “Ray-Specialized Acceleration Structures for Ray Tracing”. In: *2008 IEEE Symposium on Interactive Ray Tracing*. Aug. 2008, pp. 3–10. DOI: 10.1109/RT.2008.4634613.

[HOD23] Killian Herveau, Hisanari Otsu, and Carsten Dachsbacher. *Out-of-the-Loop Autotuning of Metropolis Light Transport with Reciprocal Probability Binning*. The Eurographics Association, 2023. ISBN: 978-3-03868-209-7. DOI: 10.2312/egs.20231005. (Visited on 12/03/2023).

[HPD23] Killian Herveau, Max Piochowiak, and Carsten Dachsbacher. *Minimal Convolutional Neural Networks for Temporal Anti Aliasing*. The Eurographics Association, 2023. ISBN: 978-3-03868-229-5. DOI: 10.2312/hpg.20231134. (Visited on 12/03/2023).

[HY21] Yuchi Huo and Sung-eui Yoon. “A Survey on Deep Learning-Based Monte Carlo Denoising”. In: *Computational Visual Media* 7 (2021), pp. 169–185. DOI: 10.1007/s41095-021-0209-9.

[Jim+12] Jorge Jimenez et al. "SMAA: Enhanced Morphological Anti-aliasing". In: *Computer Graphics Forum (Proc. of Eurographics)* 31.2 (2012). DOI: `10.1111/j.1467-8659.2012.03014.x`.

[JM12] Wenzel Jakob and Steve Marschner. "Manifold Exploration: A Markov Chain Monte Carlo Technique for Rendering Scenes with Difficult Specular Transport". In: *ACM Transactions on Graphics* 31.4 (July 2012), 58:1–58:13. ISSN: 0730-0301. DOI: `10.1145/2185520.2185554`. (Visited on 02/27/2024).

[KA97] R.M. Kretchmar and C.W. Anderson. "Comparison of CMACs and Radial Basis Functions for Local Function Approximators in Reinforcement Learning". In: *Proceedings of International Conference on Neural Networks (ICNN'97)*. Vol. 2. Houston, TX, USA: IEEE, 1997, pp. 834–837. ISBN: 978-0-7803-4122-7. DOI: `10.1109/ICNN.1997.616132`.

[Kaj86] James T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. ISBN: 978-0-89791-196-2. DOI: `10.1145/15922.15902`. (Visited on 03/07/2019).

[Kar14] Brian Karis. "High-Quality Temporal Supersampling". In: *Advances in Real-Time Rendering in Games, SIGGRAPH Courses* 1.10.1145 (2014), pp. 2614028–2615455.

[KB14] Diederik P Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv preprint* (2014). DOI: `10.48550/arXiv.1412.6980`.

[KBK11] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. "Cost-Aware Function Migration in Heterogeneous Systems". In: ACM Press, 2011, p. 137. ISBN: 978-1-4503-0241-8. DOI: `10.1145/1944862.1944883`. (Visited on 06/28/2018).

[KBS15] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. "A Machine Learning Approach for Filtering Monte Carlo Noise". In: *ACM Transactions on Graphics* 34.4 (July 2015). ISSN: 0730-0301. DOI: `10.1145/2766977`.

[Kel+02] Csaba Kelemen et al. "A Simple and Robust Mutation Strategy for the Metropolis Light Transport Algorithm". In: *Computer Graphics Forum* 21.3 (2002), pp. 531–540. ISSN: 1467-8659. DOI: `10.1111/1467-8659.t01-1-00703`. (Visited on 12/15/2022).

[Klo+98] J.T. Klosowski et al. "Efficient Collision Detection Using Bounding Volume Hierarchies of K-DOPs". In: *IEEE Transactions on Visualization and Computer Graphics* 4.1 (Jan. 1998), pp. 21–36. ISSN: 1941-0506. DOI: `10.1109/2945.675649`. (Visited on 12/10/2023).

[Lau+09] C. Lauterbach et al. "Fast BVH Construction on GPUs". In: *Computer Graphics Forum* 28.2 (Apr. 2009), pp. 375–384. ISSN: 0167-7055, 1467-8659. DOI: `10.1111/j.1467-8659.2009.01377.x`. (Visited on 02/27/2024).

[Li+15] Tzu-Mao Li et al. "Anisotropic Gaussian Mutations for Metropolis Light Transport through Hessian-Hamiltonian Dynamics". In: *ACM Transactions on Graphics* 34.6 (Nov. 2015), 209:1–209:13. ISSN: 0730-0301. DOI: 10.1145/2816795.2818084. (Visited on 12/16/2022).

[Liu20] Edward Liu. "DLSS 2.0 – Image Reconstruction for Real-Time Rendering with Deep Learning". In: *GPU Technology Conference (GTC)*. 2020.

[Lot09] T. Lottes. "FXAA". In: *Nvidia White Paper*. 2009.

[Mae+10] Hamid Reza Maei et al. "Toward Off-Policy Learning Control with Function Approximation". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. USA: Omnipress, 2010, pp. 719–726. ISBN: 978-1-60558-907-7.

[MB18] D. Meister and J. Bittner. "Parallel Reinsertion for Bounding Volume Hierarchy Optimization". In: *Computer Graphics Forum* 37.2 (May 2018), pp. 463–473. ISSN: 1467-8659. DOI: 10.1111/cgf.13376. (Visited on 03/07/2019).

[MB90] J. David MacDonald and Kellogg S. Booth. "Heuristics for Ray Tracing Using Space Subdivision". In: *The Visual Computer* 6.3 (May 1990), pp. 153–166. ISSN: 1432-2315. DOI: 10.1007/BF01911006. (Visited on 03/04/2019).

[MBC79] M. D. McKay, R. J. Beckman, and W. J. Conover. "A Comparison of Three Methods for Selecting Values of Input Variables in the Analysis of Output from a Computer Code". In: *Technometrics* 21.2 (1979), pp. 239–245. ISSN: 0040-1706. DOI: 10.2307/1268522. JSTOR: 1268522. (Visited on 03/05/2019).

[Mei+21] Daniel Meister et al. "A Survey on Bounding Volume Hierarchies for Ray Tracing". In: *Computer Graphics Forum* 40.2 (2021), pp. 683–712. ISSN: 1467-8659. DOI: 10.1111/cgf.142662. (Visited on 12/10/2023).

[Mor17] Morgan McGuire. *Computer Graphics Archive*. http://casual-effects.com/data/. Scenes Archive. July 2017. (Visited on 03/01/2019).

[MT05] Tomas Möller and Ben Trumbore. "Fast, Minimum Storage Ray/Triangle Intersection". In: *ACM SIGGRAPH 2005 Courses*. SIGGRAPH '05. New York, NY, USA: Association for Computing Machinery, July 2005, 7–es. ISBN: 978-1-4503-7833-8. DOI: 10.1145/1198555.1198746. (Visited on 12/07/2023).

[Mur+14] S. Muralidharan et al. "Nitro: A Framework for Adaptive Code Variant Tuning". In: *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. May 2014, pp. 501–512. DOI: 10.1109/IPDPS.2014.59.

[Nal+17] O. Nalbach et al. "Deep Shading: Convolutional Neural Networks for Screen Space Shading". In: *Computer Graphics Forum* 36.4 (2017), pp. 65–78. DOI: 10.1111/cgf.13225.

[NHD10] Jan Novák, Vlastimil Havran, and Carsten Dachsbacher. "Path Regeneration for Interactive Path Tracing". In: *EUROGRAPHICS*. Eurographics Association, 2010, p. 4.

[NM65] J. A. Nelder and R. Mead. "A Simplex Method for Function Minimization". In: *The Computer Journal* 7.4 (Jan. 1965).

[Nvi23] Nvidia. "TensorRT". In: *https://developer.nvidia.com/tensorrt.* 2023.

[ON94] Michael Oren and Shree K. Nayar. "Generalization of Lambert's Reflectance Model". In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques.* SIGGRAPH '94. New York, NY, USA: Association for Computing Machinery, July 1994, pp. 239–246. ISBN: 978-0-89791-667-7. DOI: `10.1145/192161.192213`. (Visited on 02/26/2024).

[Ots+18] Hisanari Otsu et al. "Geometry-Aware Metropolis Light Transport". In: *ACM Transactions on Graphics* 37.6 (Dec. 2018), 278:1–278:11. ISSN: 0730-0301. DOI: `10.1145/3272127.3275106`. (Visited on 03/23/2022).

[Ped16] Lasse Jon Fuglsang Pedersen. "Temporal Reprojection Anti-Aliasing in INSIDE". In: *Game Developers Conference* 3.4 (2016), p. 10.

[PJH17] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation.* Third edition. Cambridge, MA: Morgan Kaufmann Publishers/Elsevier, 2017. ISBN: 978-0-12-800645-0.

[PL10] Jacopo Pantaleoni and David Luebke. *HLBVH: Hierarchical LBVH Construction for Real-Time Ray Tracing of Dynamic Geometry.* The Eurographics Association, 2010. ISBN: 978-3-905674-26-2. DOI: `10.2312/EGGH/HPG10/087-095`. (Visited on 02/27/2024).

[Rei+18] Florian Reibold et al. "Selective Guided Sampling with Complete Light Transport Paths". In: *ACM Transactions on Graphics* 37.6 (Dec. 2018), pp. 1–14. ISSN: 0730-0301, 1557-7368. DOI: `10.1145/3272127.3275030`. (Visited on 11/01/2023).

[Res09] Alexander Reshetov. "Morphological Anti-aliasing". In: *Proc. of ACM SIGGRAPH / Eurographics Conference on High Performance Graphics.* HPG '09. New York, NY, USA: Association for Computing Machinery, 2009, pp. 109–116. DOI: `10.1145/1572769.1572787`.

[Sal16] Marco Salvi. "An Excursion in Temporal Super Sampling". In: *Game Developers Conference* 3.7 (2016), p. 12.

[Sal17] Marco Salvi. "Deep Learning: The Future of Real-Time Rendering". In: *ACM SIGGRAPH Courses: Open Problems in Real-Time Rendering* 12 (2017).

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction.* MIT Press, Nov. 2018. ISBN: 978-0-262-03924-6.

[Sch+17] Christoph Schied et al. "Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination". In: *Proceedings of High Performance Graphics.* Los Angeles California: ACM, July 2017, pp. 1–12. ISBN: 978-1-4503-5101-0. DOI: `10.1145/3105762.3105770`. (Visited on 11/07/2023).

[Set09] Burr Settles. *Active Learning Literature Survey.* Computer Sciences Technical Report 1648. University of Wisconsin–Madison, 2009.

[SFD09] Martin Stich, Heiko Friedrich, and Andreas Dietrich. “Spatial Splits in Bounding Volume Hierarchies”. In: *Proceedings of the 1st ACM Conference on High Performance Graphics - HPG ’09*. New Orleans, Louisiana: ACM Press, 2009, p. 7. ISBN: 978-1-60558-603-8. DOI: 10.1145/1572769.1572771. (Visited on 03/12/2019).

[SK20] Martin Sik and Jaroslav Krivanek. “Survey of Markov Chain Monte Carlo Methods in Light Transport Simulation”. In: *IEEE Transactions on Visualization and Computer Graphics* 26.4 (Apr. 2020), pp. 1821–1840. ISSN: 1077-2626, 1941-0506, 2160-9306. DOI: 10.1109/TVCG.2018.2880455. (Visited on 03/23/2022).

[SN15] Leonardo Sacht and Diego Nehab. “Optimized Quasi-Interpolators for Image Reconstruction”. In: *IEEE Transactions on Image Processing* 24.12 (2015), pp. 5249–5259. DOI: 10.1109/TIP.2015.2478385.

[SPD18] Christoph Schied, Christoph Peters, and Carsten Dachsbacher. “Gradient Estimation for Real-time Adaptive Temporal Filtering”. In: *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 1.2 (Aug. 2018), pp. 1–16. ISSN: 2577-6193. DOI: 10.1145/3233301. (Visited on 11/07/2023).

[ŢCH02] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. “Active Harmony: Towards Automated Performance Tuning”. In: *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. SC ’02. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11. (Visited on 03/17/2015).

[Tho+20] Manu Mathew Thomas et al. “A Reduced-Precision Network for Image Reconstruction”. In: *ACM Transactions on Graphics* 39.6 (Nov. 2020). ISSN: 0730-0301. DOI: 10.1145/3414685.3417786.

[Til+16] M. Tillmann et al. “Online-Autotuning of Parallel SAH kD-Trees”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. May 2016, pp. 628–637. DOI: 10.1109/IPDPS.2016.31.

[TO12] Yusuke Tokuyoshi and Shinji Ogaki. “Real-Time Bidirectional Path Tracing via Rasterization”. In: *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D ’12. New York, NY, USA: ACM, 2012, pp. 183–190. ISBN: 978-1-4503-1194-6. DOI: 10.1145/2159616.2159647. (Visited on 03/07/2019).

[Vea98] Eric Veach. “Robust Monte Carlo Methods for Light Transport Simulation”. PhD thesis. Stanford, CA, USA: Stanford University, 1998.

[VHS12] Marek Vinkler, Vlastimil Havran, and Jiří Sochor. “Visibility Driven BVH Build up Algorithm for Ray Tracing”. In: *Computers & Graphics*. Applications of Geometry Processing 36.4 (June 2012), pp. 283–296. ISSN: 0097-8493. DOI: 10.1016/j.cag.2012.02.013. (Visited on 03/05/2019).

[Vog+18] Thijs Vogels et al. “Denoising with Kernel Prediction and Asymmetric Loss Functions”. In: *ACM Transactions on Graphics (Proc. SIGGRAPH)* 37.4 (2018), 124:1–124:15. DOI: 10.1145/3197517.3201388.

[Vor+14] Jiří Vorba et al. "On-Line Learning of Parametric Mixture Models for Light Transport Simulation". In: *ACM Transactions on Graphics* 33.4 (July 2014), pp. 1–11. ISSN: 07300301. DOI: 10.1145/2601097.2601203. (Visited on 03/11/2019).

[Wal+07] Bruce Walter et al. "Microfacet Models for Refraction Through Rough Surfaces". In: *Proceedings of the 18th Eurographics Conference on Rendering Techniques*. 2007.

[Wal+14] Ingo Wald et al. "Embree: A Kernel Framework for Efficient CPU Ray Tracing". In: *ACM Trans. Graph.* 33.4 (July 2014), 143:1–143:8. ISSN: 0730-0301. DOI: 10.1145/2601097.2601199. (Visited on 03/05/2019).

[Wal+17] I Wald et al. "OSPRay - A CPU Ray Tracing Framework for Scientific Visualization". In: *IEEE Transactions on Visualization and Computer Graphics* 23.1 (Jan. 2017), pp. 931–940. ISSN: 1077-2626. DOI: 10.1109/TVCG.2016.2599041. (Visited on 03/13/2019).

[Wat20] Alexander Watson. "Deep Learning Techniques for Super-Resolution in Video Games". In: *arXiv preprint* (2020). DOI: 10.48550/arXiv.2012.09810.

[WD98] R. C. Whaley and J. J. Dongarra. "Automatically Tuned Linear Algebra Software". In: *IEEE/ACM Conference on Supercomputing, 1998.SC98*. Nov. 1998, pp. 38–38. DOI: 10.1109/SC.1998.10004.

[WG17] Dominik Wodniok and Michael Goesele. "Construction of Bounding Volume Hierarchies with SAH Cost Approximation on Temporary Subtrees". In: *Computers & Graphics* 62 (Feb. 2017), pp. 41–52. ISSN: 0097-8493. DOI: 10.1016/j.cag.2016.12.003. (Visited on 03/07/2019).

[Xia+20] Lei Xiao et al. "Neural Supersampling for Real-Time Rendering". In: *ACM Transactions on Graphics* 39.4 (Aug. 2020). ISSN: 0730-0301. DOI: 10.1145/3386569.3392376.

[Yan+09] Lei Yang et al. "Amortized Supersampling". In: *ACM Transactions on Graphics* 28.5 (Dec. 2009), pp. 1–12. ISSN: 0730-0301. DOI: 10.1145/1618452.1618481.

[YLS20] Lei Yang, Shiqiu Liu, and Marco Salvi. "A Survey of Temporal Anti-aliasing Techniques". In: *Computer Graphics Forum* 39.2 (2020), pp. 607–621. ISSN: 1467-8659. DOI: 10.1111/cgf.14018. (Visited on 03/06/2023).

[Zen+21] Zheng Zeng et al. "Temporally Reliable Motion Vectors for Real-Time Ray Tracing". In: *Computer Graphics Forum* 40.2 (2021), pp. 79–90. DOI: 10.1111/cgf.142616.

[ZS13] Károly Zsolnai and László Szirmay-Kalos. "Automatic Parameter Control for Metropolis Light Transport". In: (2013). ISSN: 1017-4656. DOI: 10.2312/conf/EG2013/short/053-056. (Visited on 12/16/2022).

A Appendix

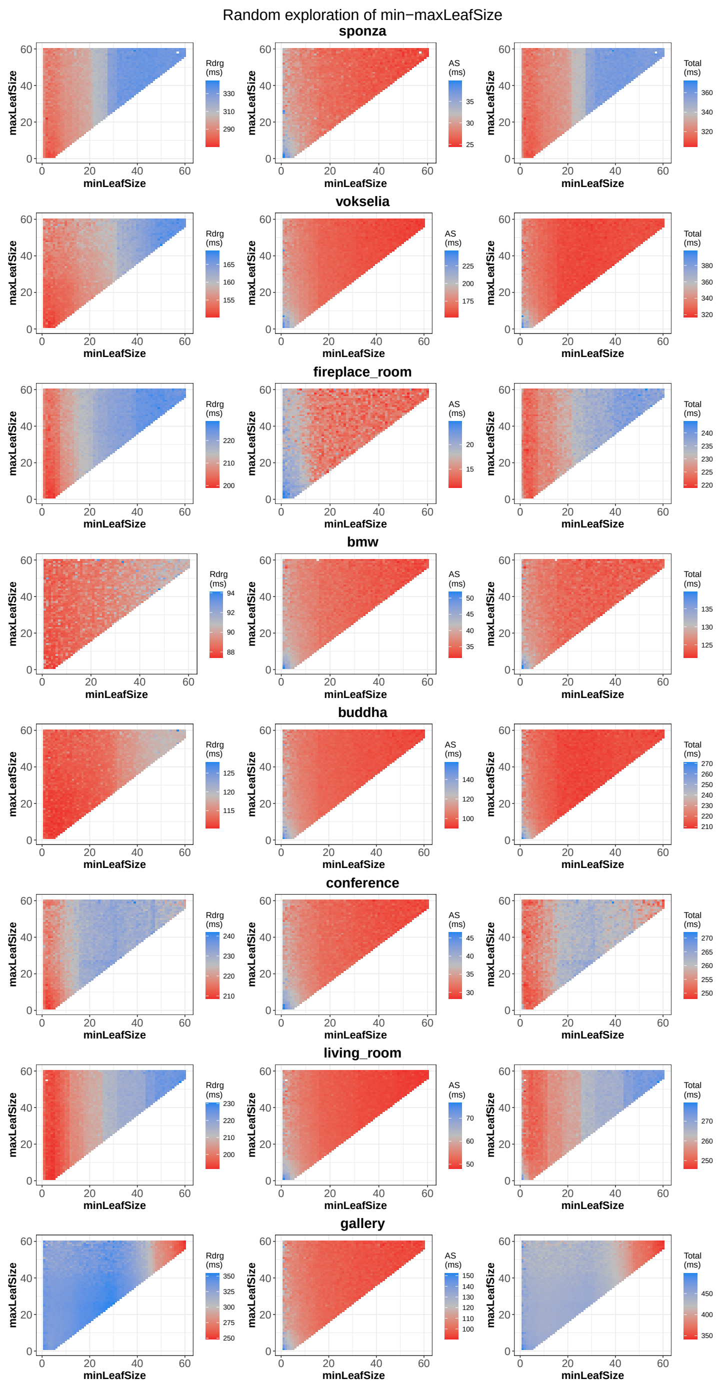


Figure A.1: Random exploration of `minLeafSize` and `maxLeafSize` Scenes behave the same way: big leaves are good for AS build time, small for rendering time. Yet optimal tradeoff is different for every scene. Only gallery presents an unexpected behavior. This scene is made with scanned 3D data, which may have an odd structure

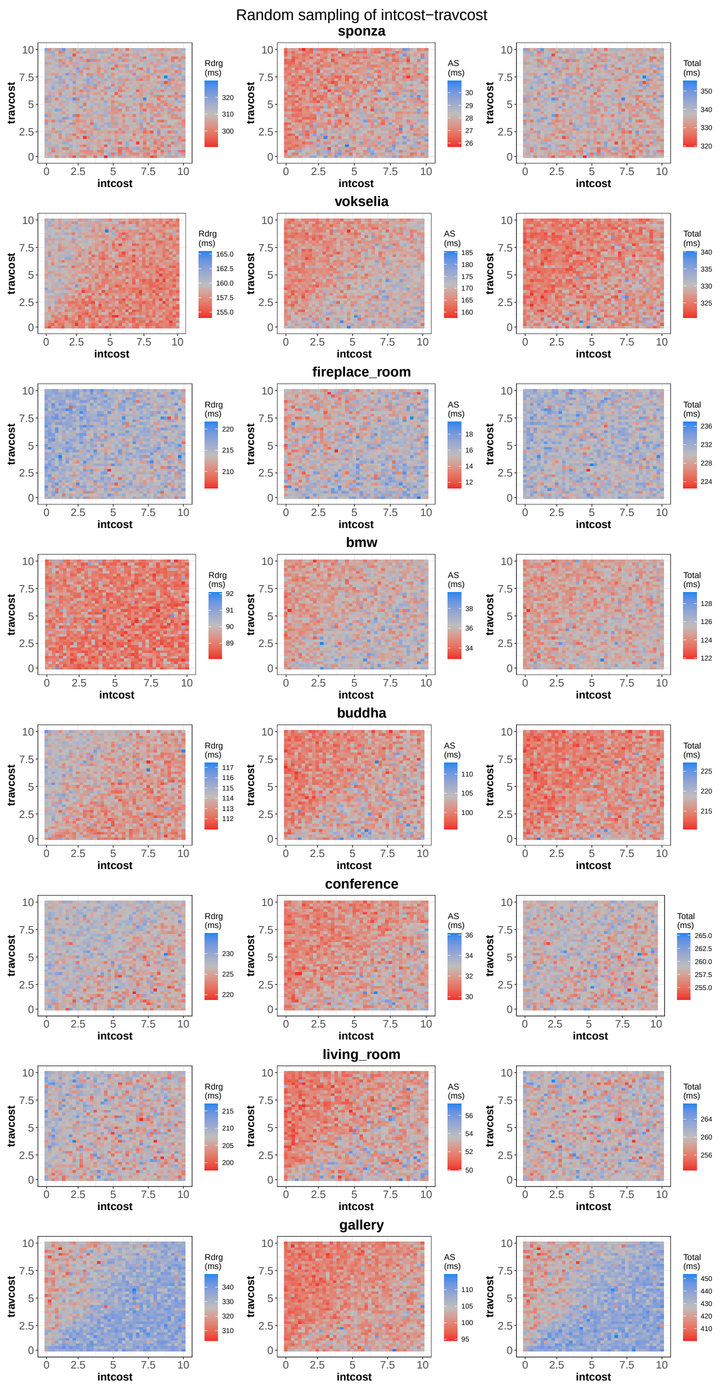


Figure A.2: Random exploration of `intcost` and `travcost` Note: `quality` = 1. Most scenes show gradient of performance where the top left corner is better for build time (second column), and worse for rendering time (first column). This graph shows that finding a couple of parameters is not necessary beneficial without a specific configuration of the other parameters.

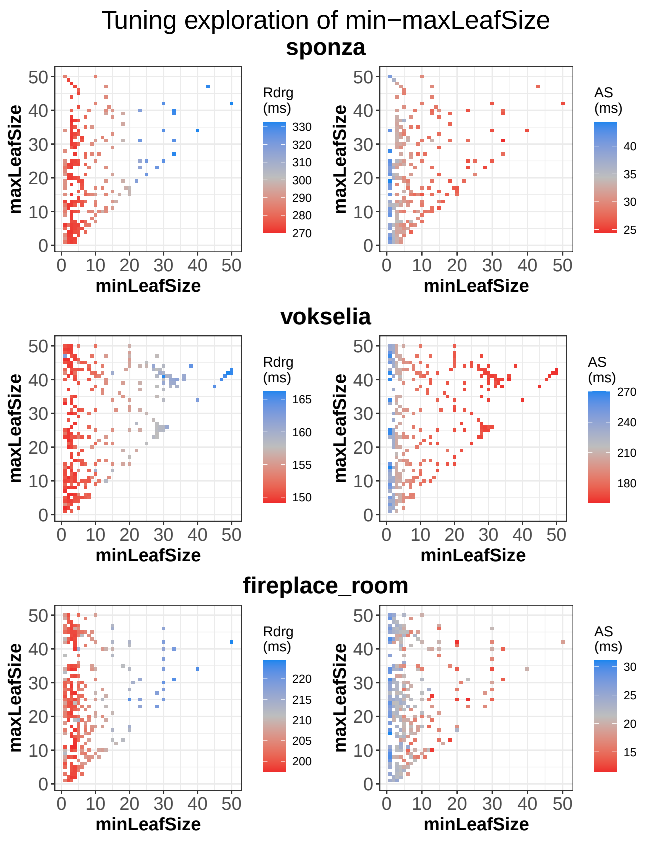


Figure A.3: Optima distribution found by tuning `minLeafSize`, `maxLeafSize`. small to medium scenes prefer small leafSize and `vokselia`'s optimum is 25 to 30. It is similar to the distribution obtained in Figure A.1

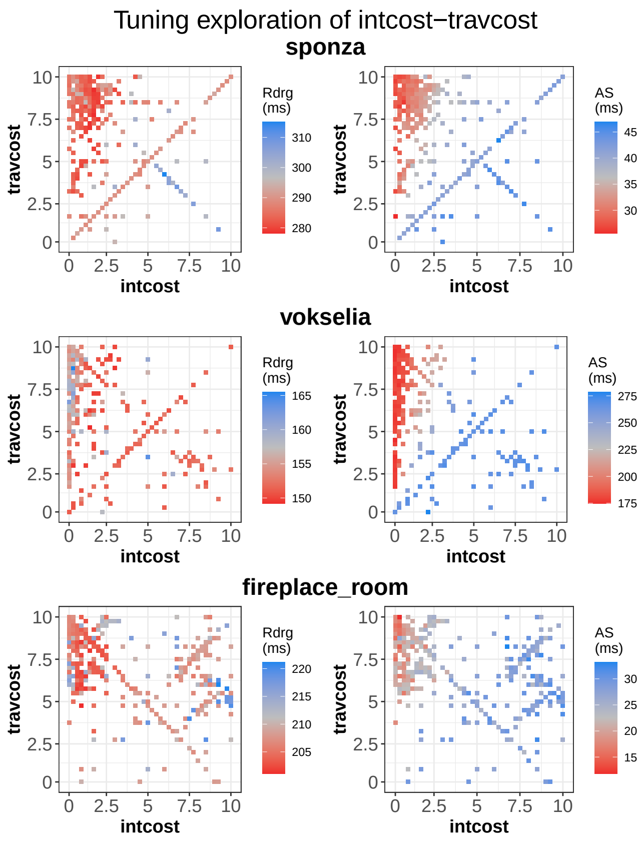


Figure A.5: Optima distribution found by tuning `intcost`, `travcost`. Optima are bundled around the top left corner: Low `intcost` and high `travcost` make the tree structure shallower. Straight lines simply show converging attempts of the tuner, exploring the space. `vokselia` presents a local optimum found by the tuner.

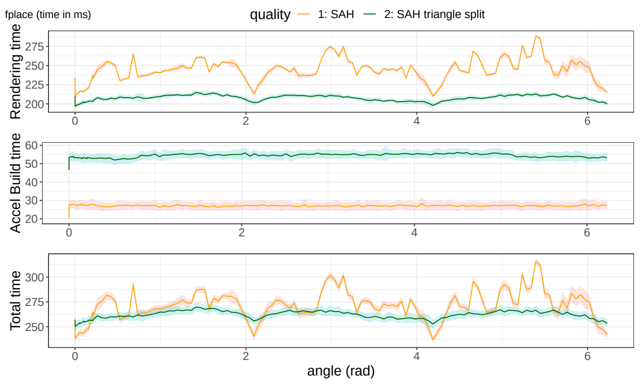


Figure A.4: Evolution of performance during cylinder rotation experiment. `fireplace_room`. The evolution of AS building time is very stable for this scene. The scene is too small for the cylinder to make a significant difference, even when intersecting most of the geometry.

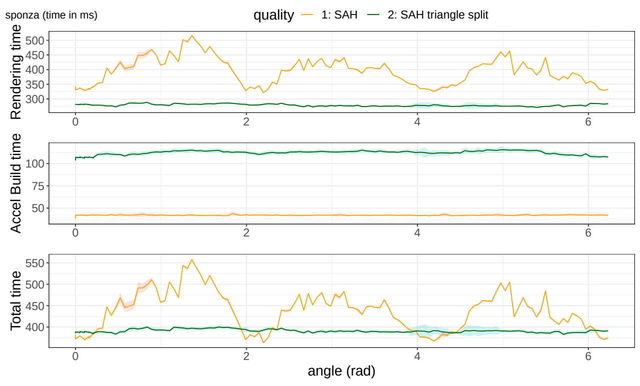


Figure A.6: Evolution of performance during cylinder rotation experiment. `sponza`. Triangle splitting is the better choice most of the time here. Rendering time is heavily influenced by the intersections, the worst performance is 1.5 times the one of the start.

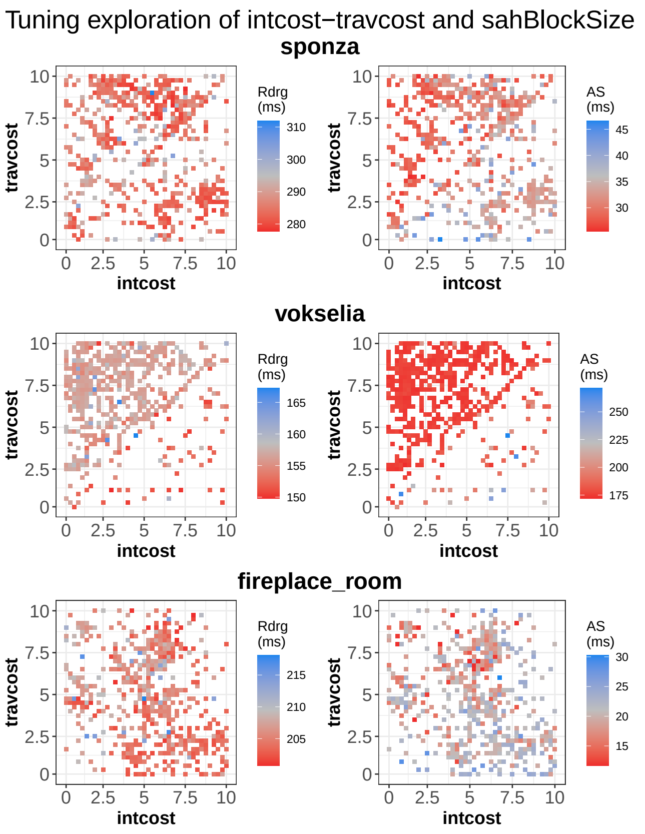


Figure A.7: Optima distribution found by tuning `intcost`, `travcost` and `sahBlockSize`. `sahBlockSize` is tuned at the same time but is not shown here. Optima are scattered significantly more than in Figure A.5 but still organized in clusters. The sample density is very high in the clusters, indicating consistent convergence.

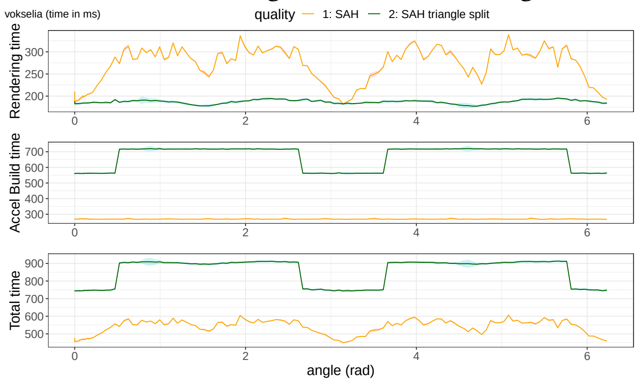


Figure A.8: Evolution of performance during cylinder rotation experiment. `vokselia`. Sudden drop of performance at 0.5 rad, triangle split is not worth the cost for this configuration. This drop in performance is due to the cylinder intersecting a large amount of nodes, causing a lot of subsequent triangle splits. Performance in rendering time is far superior with splitting