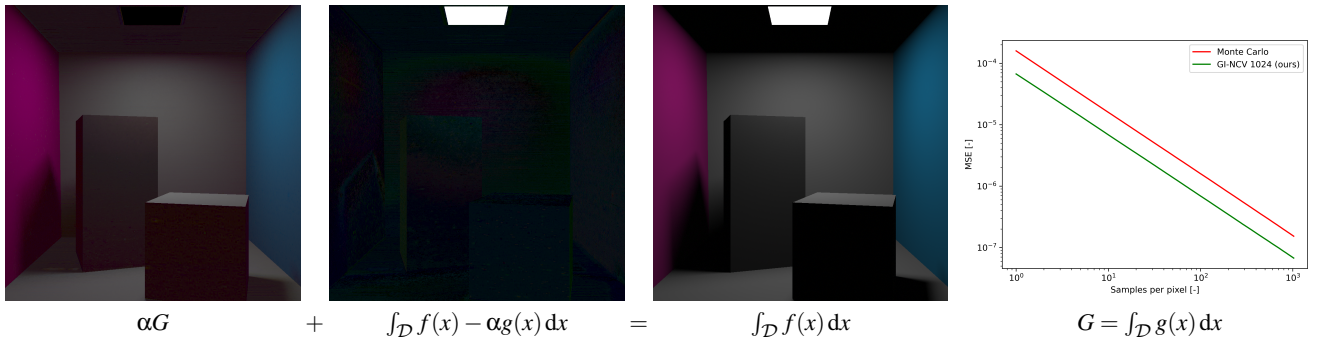


# Geometric Integration for Neural Control Variates

D. Meister<sup>id</sup> and T. Harada<sup>id</sup>

Advanced Micro Devices, Inc.



**Figure 1:** We approximate the integrand by a small neural network (i.e., multilayered perceptron). Thanks to our geometric subdivision, we can integrate the neural network analytically, and use it as a control variate for Monte Carlo integration. The integral of the approximation provides a biased estimate (left), which is corrected by Monte Carlo integration of the residual integrand (center left), obtaining the final unbiased estimate (center right), which can achieve a lower error than vanilla Monte Carlo (right).

## Abstract

Control variates are a variance-reduction technique for Monte Carlo integration. The principle involves approximating the integrand by a function that can be analytically integrated, and integrating using the Monte Carlo method only the residual difference between the integrand and the approximation, to obtain an unbiased estimate. Neural networks are universal approximators that could potentially be used as a control variate. However, the challenge lies in the analytic integration, which is not possible in general. In this manuscript, we study one of the simplest neural network models, the multilayered perceptron (MLP) with continuous piecewise linear activation functions, and its possible analytic integration. We propose an integration method based on integration domain subdivision, employing techniques from computational geometry to solve this problem in 2D. We demonstrate that an MLP can be used as a control variate in combination with our integration method, showing applications in the light transport simulation.

## 1. Introduction

To synthesize photorealistic images, we need to solve notoriously complex integrals that model the underlying light transport. In general, these integrals do not have an analytic solution, and thus we employ tools of numerical integration to solve them. Among those, Monte Carlo integration is prominent, providing a general and robust solution, efficiently dealing with, for example, high dimensions or discontinuities that other numerical methods may struggle with. Monte Carlo converges to a correct solution with an increasing number of samples; however, it may require a large number of samples to suppress variance, that otherwise exhibits as high-frequency noise in the rendered images, under an acceptable threshold.

Thus, various variance-reduction techniques have been proposed. Widely studied importance sampling minimizes the variance by sampling from a probability distribution that matches the underlying integrand. In physically-based rendering, the goal is to sample directions from which incoming radiance is high. Such techniques include sampling according to the bidirectional reflectance distribution function (BRDF), sampling of directly visible light sources, or more general path guiding methods accounting for complex global illumination phenomena.

Control variates are another variance-reduction technique that received significantly less attention in computer graphics than importance sampling. The principle is to approximate the integrand by a simpler function that can be analytically integrated. To get an un-

biased estimate the analytic integration is corrected by integrating the difference between the integrand and the approximation, thanks to the following identity:

$$F = \int_{\mathcal{D}} f(x) dx = \alpha G + \int_{\mathcal{D}} f(x) - \alpha g(x) dx, \quad (1)$$

where  $f(x)$  is the integrand,  $\mathcal{D}$  is an integration domain,  $g(x)$  is a function approximating  $f(x)$  such that  $G = \int_{\mathcal{D}} g(x) dx$ , and  $\alpha$  is a scale parameter, controlling influence of the control variates. The variance reduction depends on how  $g(x)$  matches  $f(x)$ .

Neural networks are universal approximators, making them excellent candidates for control variates. However, integrating a general neural network could be as difficult as integrating the original integrand. Recently, it has been showed that the multilayer perceptron (MLP), the simplest neural network model, with sophisticated input encoding [FYH23] is capable to represent credibly a high variety of functions [MRNK21, TLY\*21, FKH23]. Compared to complex architectures, the MLP is small and can be online-trained efficiently on contemporary GPUs [MRNK21].

In this manuscript, we focus on the MLP with continuous piecewise linear activation functions. The key observation is that the integration domain of such function can be split into convex subdomains, where each subdomain corresponds to a single affine function [Liu24]. Thanks to the linearity of the integrals, we can formulate the integration as a sum of the integrals of the individual subdomains. We utilize some of the algorithms and data structures from computational geometry [BCKO08] to efficiently compute the convex regions and the corresponding affine functions in 2D. In particular, our contributions are as follows:

- We formulate an analytic integration of the MLP as a geometric problem, offering a novel perspective on the subject.
- We propose an algorithm employing algorithms and data structures from computational geometry to solve this problem in 2D.
- We employ the proposed integration in combination with control variates, demonstrating the application in the light transport simulation.

## 2. Related Work

Control variates have been successfully adapted in various area such as operations research [HN98] or financial mathematics [KV90]. However, their application in computer graphics remains relatively limited. Rousselle et al. [RJN16] applied image-space control variates for re-rendering and the Poisson reconstruction in the gradient-domain rendering. Xu et al. [XLG\*24] proposed importance sampling for sampling the residual integral in the problem of re-rendering. Crespo et al. [CJMn21] proposed using piecewise polynomials as a control variate in the primary sample space. Similarly, Salaün et al. [SGH\*22] use the polynomial regression, additionally showing the connection between the least-square regression and Monte Carlo integration. Kondapaneni et al. [KVG\*19] showed that the optimal weights for multiple importance sampling can be interpreted as an optimal control variate. Nicolet et al. [NRN\*23] propose recursive control variates for differentiable rendering.

Since neural networks can approximate complex distributions

better than traditional methods, it is tempting to use them as control variates. The challenging part is the integration of a neural network ( $G$  in Equation 1). Müller et al. [MRKN20] proposed to use the normalizing flows scaled by an additional neural network to fit a given distribution, where the scale is trivially the integration of the approximation. The architecture of normalizing flows is quite complex due to the requirement for the invertibility of the mapping, consisting of a chain of coupling layers. Normalizing flows excel in high dimensions, but it is not straightforward to use them in lower dimensions (e.g., 1D or 2D). Note that the invertibility is a stronger requirement than integrability, limiting the expressiveness of the model. The parameters of each coupling layer are provided by a neural network, resulting in a model with a large number of parameters.

Lindell et al. [LMW21] proposed an automatic integration of an arbitrary neural network based on the fundamental theorem of calculus. The idea is to represent an antiderivative (that can be used for the integration) as a neural network (*integral network*) such that the output is differentiated (using an autodiff) before feeding to a loss function. The authors proposed to construct another neural network (*grad network*) that represents the derivative of the integral network such that both networks share the same parameters, and thus the grad network can be trained directly avoiding excessive use of the autodiff. Li et al. [LYZ\*24] proposed to use the automatic integration in combination with control variates, demonstrating the variance reduction in estimation of the Poisson and Laplace equations using the walk-on-spheres algorithm. Subr [Sub21] proposed a Q-NET integration algorithm, exploiting the fact that an antiderivative of a sum of sigmoids can be computed as a weighted sum of antiderivatives of the individual sigmoids.

There are several caveats about the automatic integration. For a  $d$ -dimensional integration domain, we need to compute  $d$ -th order derivatives, which might be very expensive for higher dimensions. Even if we can afford to compute higher order derivatives, the derivatives must be non-zero to be able to fit the function. For example, the second-order derivatives of an MLP with linear layers and rectified linear units (ReLU) results in zero (i.e., applying the product rule on the result of the chain rule). Another caveat is that the architecture of the grad network cannot be controlled directly (the size of the grad network is actually quadratic with the respect of the size of the integral network [LMW21]), making the training more difficult.

Müller et al. [MRNK21] and Hadadan et al. [HCZ21] showed that even a simple MLP can represent high-frequency radiance field. Furthermore, such a small neural network can be fit into an on-chip-memory, and thus can be trained/infer efficiently on modern GPUs [MRNK21]. An input encoding is an essential component to learn high-frequency signals from lower dimensional inputs. Müller et al. [MESK22] proposed multi-resolution hashgrid with learnable parameters (i.e., latent feature vectors) for encoding world-space positions. In this case, the MLP can be relatively shallow, serving as a decoder of the latent feature vectors. Our goal is to use such a small MLP with continuous piecewise linear activation functions [HBBB23] as a control variate. Liu [Liu24] showed that an MLP with the ReLU activation functions represents a piecewise affine function, and thus the integration can be formulated as a sum



of integrals, where each integral is an affine function defined on a convex subdomain. However, the author does not show how to find the subdomains. Dereviannykh et al. [DKHD25] proposed to use the fully-fused MLP to cache incidence radiance in combination with multi-level Monte Carlo integration, replacing the analytical integration in control variates by another Monte Carlo integration.

### 3. Geometric Integration

In order to use an MLP as a control variate, we need to be able to analytically integrate it (see Equation 1). In this section, we describe our integration method based on the integration domain subdivision. The key observation is that an MLP with the continuous piecewise linear activation functions, such as the step function or the (leaky) rectified linear unit (ReLU), itself represents a piecewise linear (affine) function [HBBB23]. The integration domain can be split into disjoint subdomains, where we can solve the integration separately on each subdomain. Assuming that function  $g(x)$  is represented by such MLP, we can obtain  $G$  by summing up  $n$  integrals on the  $n$  corresponding subdomains:

$$G = \int_{\mathcal{D}} g(x) dx = \sum_{i=1}^n \int_{\mathcal{D}_i} g(x) dx = \sum_{i=1}^n \int_{\mathcal{D}_i} g_i(x) dx, \quad (2)$$

such that  $\bigcup_{i=1}^n \mathcal{D}_i = \mathcal{D}$ ,  $\bigcap_{i=1}^n \mathcal{D}_i = \emptyset$ , and  $g_i(x)$  is an affine function defined on  $\mathcal{D}_i$ . Determining  $\mathcal{D}_i$  and  $g_i(x)$  presents a challenge.

#### 3.1. Integration Domain Subdivision

From now on, we restrict ourselves to integration in 2D (using variables  $x$  and  $y$ ). Without loss of generality, we can also assume that the integration domain is a unit square and the activation functions are ReLUs. The goal is to split the integration domain (i.e., the unit square) into subdomains such that each subdomain has an associated affine function that is identical to the function induced by the MLP.

We construct the subdomains and the corresponding affine functions incrementally, proceeding from the input layer to the output layer, based on whether previous neurons are activated or not. It is important to keep in mind that the elements we are working with are affine functions (i.e., an affine combination of  $x$  and  $y$  in the form of  $ax + by + c$ ) instead of numbers, relying on the fact that the affine functions are closed under composition.

The ReLU functions in neurons pass positive values and clamp negative ones to zero. Since, in our case, the input is an affine function, we need to decide for which  $(x, y)$ , the function is positive and negative. This geometrically means that a line (corresponding to where the function is zero) splits the 2D plane into negative and positive half-planes. In the case of single hidden layer, each neuron defines such a line, and a subdomain is an intersection of the corresponding half-planes; either positive or negative half-plane of the neuron participating in the intersection. An example of such subdivision is illustrated in Figure 2.

For multiple hidden layers, we process each subdomain from the previous layer recursively. The inputs for the current layer are outputs of the previous layer transformed by the corresponding weights and biases, resulting in new affine functions, which define

the lines of the current layer. Notice that the lines (and the corresponding affine functions) inside each subdomain are different, as different neurons of the previous layer have been activated (providing different inputs for the current layer) on different subdomains.

We repeat this process until we reach the output layer, where the outputs of the last hidden layer are transformed by the output layer weights and biases, obtaining the affine function for a bottom-most subdomain. If we use the ReLU function on the output, we need to perform one more level of subdivision. We can also use other continuous piecewise linear activation functions other than ReLU such as the leaky ReLU or the step function, as long as the output is an affine function. Otherwise, a non-linear output would result in a non-linear input for the following layers, resulting in non-linear boundaries (i.e., general curves instead of straight lines) between subdomains.

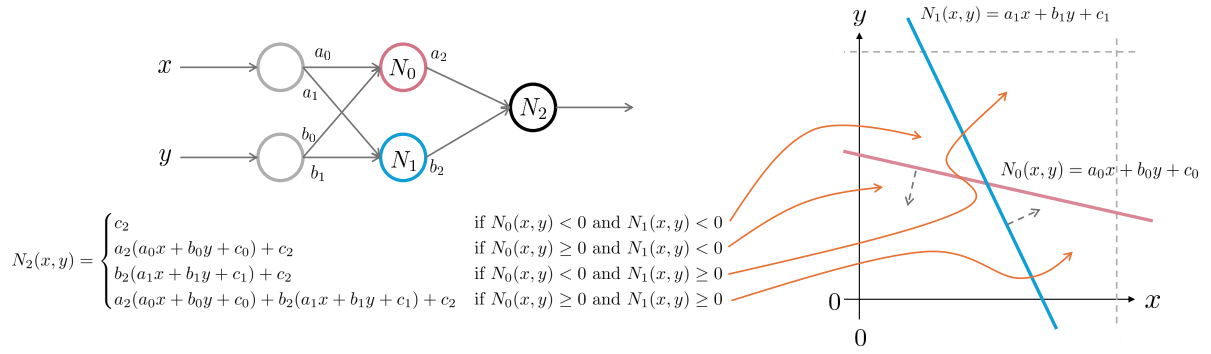
#### 3.2. Line Arrangement

In the previous section, we described how to find subdomains  $\mathcal{D}_i$  and the associated functions  $g_i$  that can be used for the integration (see Equation 2). However, so far, it is not clear how to algorithmically compute the intersections of half-planes.

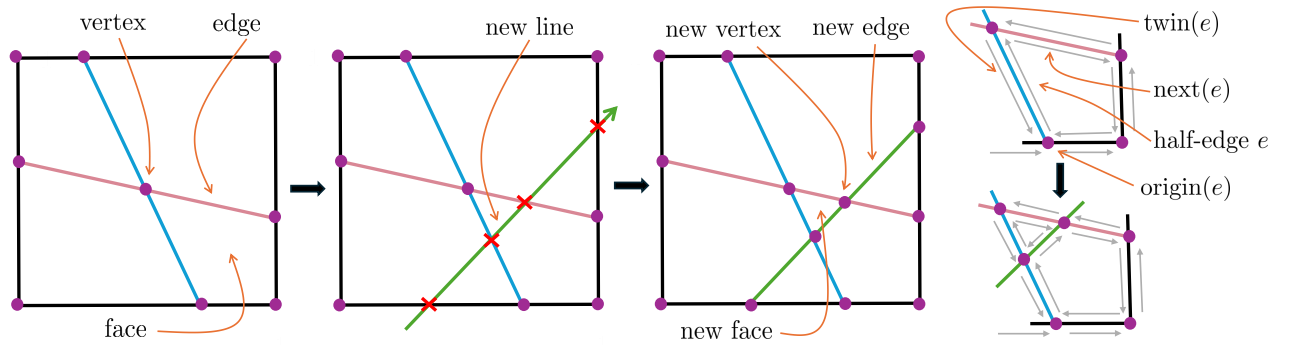
There is a problem known as *line arrangement*, and it is well-studied in computational geometry [BCKO08]. Given a set of lines, the goal is to find a planar subdivision defined by vertices (intersections of the lines), edges (line segments between intersections of one line with other lines), and faces (convex regions containing no lines), where the faces correspond to the subdomains we want to find. The lines are typically cropped by a bounding box to avoid unbounded edges. In our case, we can naturally use the unit square as the bounding box. An example of such planar subdivision is illustrated in Figure 3.

Doubly-connected edge list (DCEL) [BCKO08] is a data structure commonly used to store a planar subdivision. The advantage of the DCEL is that it contains topological information of individual elements. Each edge is represented by two (oriented) half-edges, where each half-edge is defined by one of the vertices (*origin*) of the edge, a pointer to a half-edge within the same face such that its origin is the other vertex of the first edge (*next*), and a pointer to a half-edge with an opposite orientation on the other side of the edge (*twin*). Thanks to this representation, we can walk around the boundary of a given face, or visit the neighboring faces. Each half-edge has a pointer to a face that belongs to, and each face has a pointer to one of its half-edges. A fragment of a DCEL is depicted in Figure 3.

Armed with the DCEL, we can now solve the line arrangement problem by incrementally inserting lines one by one. The DCEL is initialized with the unit square: four vertices, eight half-edges corresponding to four edges, one inner face, and one (unbounded) outer face. We start with intersecting the line with half-edges of the outer face. We split the half-edge and its twin by inserting a new vertex. Then, we continue to the face on the opposite side. Inside this face, we find the second intersection of the face, split the half-edge and its twin. We also split the face into two parts by inserting two additional half-edges, and then we recursively continue to the next face. This resembles 2D ray tracing finding all intersections.



**Figure 2:** An example of the integration domain subdivision for a simple MLP with one hidden layer containing two neurons (the weights are denoted by  $a_i$ ,  $b_i$ , and biases  $c_i$ ), using the ReLU activation function. The input to each hidden neuron is an affine combination of  $x$  and  $y$ , which is subsequently fed to the ReLU. The ReLU function is activated based on the sign of the input affine function, such that we can split the 2D plane by a line into positive (indicated by the dashed arrows) and negative half-planes, where the line itself corresponds to  $x$  and  $y$ , at which the function is zero. Two hidden neurons correspond to two lines, resulting in four subdomains defined by the intersection of either positive or negative half-planes.



**Figure 3:** An example illustrating a planar subdivision (left), the process of inserting a new line (middle), and a doubly-connected edge list (DCEL) (right). Inserting a new line requires finding the intersection with the existing edges, which correspond to new vertices. New edges are added between the consecutive intersections along the line. The planar subdivision is stored in the DCEL, that allows us to traverse through the planar subdivision, splitting edges and faces, thanks to the topological information provided by the DCEL.

The line arrangement problem can be generalized into arbitrary dimensions, where we replace lines by general hyperplanes. However, it suffers from the curse of dimensionality, which corresponds to the fact that any subset of neurons can be activated, leading to the exponential space complexity of the subdivision elements. Luckily, there are upper bounds for the planar subdivision [BCKO08]. For  $N$  lines (corresponding to the  $N$  neurons in a hidden layer), the number of faces is not greater than  $\binom{N}{2} + N + 1$ , the number of edges is not greater than  $N^2$ , and the number of vertices is not greater than  $\binom{N}{2}$ .

We need one additional piece of information in order to transit from one hidden layer to another. For each face, we store a mask of lines indicating whether the face lies on the positive side of a line. Note that some lines may lie completely outside the integration domain, which results in no new faces, making the above upper

bounds even more conservative. However, we still need to update the line mask if the integration domain lies on the positive side.

### 3.3. Integration Algorithm

In the previous section, we showed that the integration domain subdivision induced by a single hidden layer can be formulated as the line arrangement problem. In this section, we discuss how to generalize it for multiple hidden layers, and how to compute  $G$ .

As we discussed in Section 3.1, each face needs to be processed recursively. Naïvely, we could allocate a DCEL for each face in the current hidden layer, initialize it with the face and process it recursively. However, this would result in high memory consumption, getting worse with more hidden layers. The key observation is that we want to accumulate the  $G$ , and thus we do not need to store all subdomains, that have already processed, in the memory. In other

words, once the face is processed, we can reuse the DCEL for the next face, requiring only one DCEL per hidden layer.

This is summarized in Algorithm 1, resembling a traversal of a wide tree such that the branching factor is equal to the number of faces. For  $M$  hidden layers, we allocate  $M$  DCEL structures (line 4). Since we need to visit all subdomains, the traversal can be performed in a stackless manner. We keep the current depth and the index of the current face on each level (stored in variable *trail*). We initialize the root DCEL with the unit square, add the lines corresponding to the neurons of the first hidden layer (lines 5-10), and then we enter the main loop.

If we have not reached the last layer (lines 13-34), we first check whether all faces of the current hidden layer have been processed (lines 14-18). If so, we reset the trail and decrease the depth, going to the previous hidden layer. Otherwise, we initialize a DCEL of the next hidden layer with the current face and add the corresponding lines (lines 22-33). Each such line is an affine combination of lines (plus the bias) in the current layer, excluding those lines when the face lies on the negative side of these lines (lines 25-31). This corresponds to when the neuron is not activated, and it is exactly the information the line mask stores. We increase the trail and depth (lines 20-21), going to the next layer.

If we reach the last layer (lines 34-48), we compute the output function for each face as an affine combination of lines (lines 38-44). This is very similar to when we combine the lines between hidden layers. Assuming a single neuron in the output layer, the output weights are represented as a single vector of length  $M$  and the bias is a scalar value. This could be easily generalized into multiple output neurons. Once we have a face with the corresponding function, we can integrate it (line 45), which we discuss in the following section.

### 3.4. Subdomain Integration

In the previous section, we described an algorithm that finds a subdomain and the corresponding function. The last missing piece is to show how to integrate such function on the subdomain. Remember that the function is affine (thanks to the closure under composition) and the subdomain is convex (thanks to the intersection of half-planes). It is straightforward to find the antiderivative of the affine function. However, it is not possible to integrate such a function as an iterative integral (i.e., as two nested one-dimensional integrals) on a general convex subdomain, since  $x$  and  $y$  on the boundary are functions of each other. Thus, we need to split the subdomain into regions with a boundary where at least one of  $x$  and  $y$  is independent of the other one (e.g., axis-aligned rectangles or triangles). We opt for triangulation, where each triangle is transformed into a unit axis-aligned triangle. The triangulation is trivial since the subdomain is convex. For instance, we can split the subdomain into the triangle fan. After the triangulation, we can rewrite Equation 2 as follows:

$$G = \sum_{i=1}^n \int_{\mathcal{D}_i} g_i(x, y) dy dx = \sum_{i=1}^n \sum_{j=1}^{m_i} \int_{\mathcal{T}_{i,j}} a_i x + b_i y + c_i dy dx, \quad (3)$$

where  $\mathcal{T}_{i,j}$  is  $j$ -th triangle in  $\mathcal{D}_i$ ,  $m_i$  is the number of triangles in  $\mathcal{D}_i$ , and  $(a_i, b_i, c_i)$  are coefficients of the affine function associated with

**Algorithm 1** Our integration algorithm traverses the recursion tree in a stackless manner. In each step, we combine lines from the previous layer for the next layer, using weights, biases, and the line mask. At the last layer, we compute the output function as a combination of lines in the last hidden layer. Finally, we integrate the function on a particular face, accumulating the result in  $G$ .

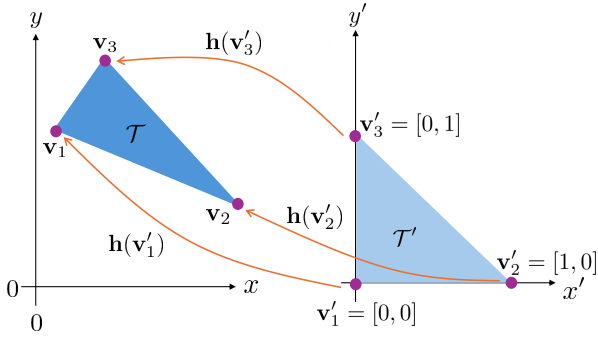
---

```

1:  $G \leftarrow 0$ 
2:  $depth \leftarrow 0$ 
3:  $trail[0, \dots, M-1] \leftarrow \{0, \dots, 0\}$ 
4:  $dcels[0, \dots, M-1] \leftarrow \{\text{DCEL}(N), \dots, \text{DCEL}(N)\}$ 
5:  $\text{INITWITHUNITSSQUARE}(dcels[0])$ 
6: for  $i = 0$  to  $N-1$  do
7:    $(weights, biases) \leftarrow \text{WEIGHTS\&BIASES}(0)$ 
8:    $line = \{weights[i, 0], weights[i, 1], biases[i]\}$ 
9:    $\text{ADDLINE}(dcels[0], i, line)$ 
10: end for
11: while  $depth \geq 0$  do
12:    $dcel \leftarrow dcels[depth]$ 
13:   if  $depth < M-1$  then
14:     if  $trail[depth] = |dcel.faces|$  then
15:        $trail[depth] \leftarrow 0$ 
16:        $depth \leftarrow depth - 1$ 
17:       continue
18:     end if
19:      $face \leftarrow dcel.faces[trail[depth]]$ 
20:      $trail[depth] \leftarrow trail[depth] + 1$ 
21:      $depth \leftarrow depth + 1$ 
22:      $\text{INITWITHFACE}(dcels[depth], face)$ 
23:      $(weights, biases) \leftarrow \text{WEIGHTS\&BIASES}(depth)$ 
24:     for  $i = 0 \rightarrow N-1$  do
25:        $line \leftarrow \{0, 0, 0\}$ 
26:       for  $j = 0 \rightarrow N-1$  do
27:         if  $face.lineMask \ \& \ (1 \ll j)$  then
28:            $line \leftarrow line + weights[i, j] \cdot dcel.lines[j]$ 
29:         end if
30:       end for
31:        $line.c \leftarrow line.c + biases[i]$ 
32:        $\text{ADDLINE}(dcels[depth], i, line)$ 
33:     end for
34:   else
35:      $(weights, bias) \leftarrow \text{WEIGHTS\&BIASES}(M)$ 
36:     for  $i = 0$  to  $|dcel.faces| - 1$  do
37:        $face \leftarrow dcel.faces[i]$ 
38:        $line \leftarrow \{0, 0, 0\}$ 
39:       for  $j = 0$  to  $N-1$  do
40:         if  $face.lineMask \ \& \ (1 \ll j)$  then
41:            $line \leftarrow line + weights[j] \cdot dcel.lines[j]$ 
42:         end if
43:       end for
44:        $line.c \leftarrow line.c + bias$ 
45:        $G \leftarrow G + \text{INTEGRATE}(face, line)$ 
46:     end for
47:      $depth \leftarrow depth - 1$ 
48:   end if
49: end while

```

---



**Figure 4:** An example illustrating function  $\mathbf{h}$ , mapping points in the unit triangle back to the original triangle.

$\mathcal{D}_i$ . For the sake of clarity, we will omit subscripts in the following formulas. We first define a mapping from the unit triangle to the original one:

$$\mathbf{h}: (x', y') \mapsto \mathbf{v}_1 + x'(\mathbf{v}_2 - \mathbf{v}_1) + y'(\mathbf{v}_3 - \mathbf{v}_1), \quad (4)$$

where  $\mathbf{v}_k$  are vertices of triangle  $\mathcal{T}$ , and  $(x', y')$  are (local) coordinates in the space of the unit triangle. We rewrite the vector function in Equation 4 into two scalar functions:

$$h_1(x', y') = v_{1,1} + x'(v_{2,1} - v_{1,1}) + y'(v_{3,1} - v_{1,1}), \quad (5)$$

$$h_2(x', y') = v_{1,2} + x'(v_{2,2} - v_{1,2}) + y'(v_{3,2} - v_{1,2}), \quad (6)$$

where  $v_{k,t}$  is  $t$ -th component of vertex  $\mathbf{v}_k$ . Mapping  $\mathbf{h}$  is illustrated in Figure 4. We plug Equations 5 and 6 into the integrand in Equation 3:

$$\int_{\mathcal{T}} ax + by + c \, dy \, dx = |J_{\mathbf{h}}| \int_0^1 \int_0^{1-x'} a \cdot h_1(x', y') + b \cdot h_2(x', y') + c \, dy' \, dx', \quad (7)$$

where  $|J_{\mathbf{h}}|$  is Jacobian determinant of function  $\mathbf{h}$ , which is twice an area of the triangle. Functions  $h_1$  and  $h_2$  are affine functions, and thus we can plug them into the affine function associated with  $\mathcal{D}_i$  to obtain coefficients for new affine function (with respect to  $(x', y')$ ):

$$|J_{\mathbf{h}}| \int_0^1 \int_0^{1-x'} a \cdot h_1(x', y') + b \cdot h_2(x', y') + c \, dy' \, dx' = 2A_{\mathcal{T}} \int_0^1 \int_0^{1-x'} a'x' + b'y' + c' \, dy' \, dx' \quad (8)$$

where  $A_{\mathcal{T}}$  is an area of triangle  $\mathcal{T}$ , and  $(a', b', c')$  are coefficients of the composed affine function in the space of the unit triangle. The whole integral reduces to the following expression:

$$2A_{\mathcal{T}} \int_0^1 \int_0^{1-x'} a'x' + b'y' + c' \, dy' \, dx' = \frac{A_{\mathcal{T}}}{3} (a' + b' + 3c'). \quad (9)$$

#### 4. Light Transport Simulation

In this section, we formulate several light transport problems as 2D integrations with the control variates, employing the proposed inte-

| Parameter          | Symbol                   | Encoding                | Input dim. | Output dim.  |
|--------------------|--------------------------|-------------------------|------------|--------------|
| Area light sample  | $(x, y)$                 | Identity                | 2          | 2            |
| Incoming direction | $\omega_i$               | Local hemisph. coords.  | 2          | 2            |
| Position           | $\mathbf{x}$             | Hashgrid                | 3 (2)      | $2 \times 4$ |
| Outgoing direction | $\omega_o$               | Sph. coords. / one-blob | 2          | $2 \times 4$ |
| Surface normal     | $\mathbf{n}(\mathbf{x})$ | Sph. coords. / one-blob | 2          | $2 \times 4$ |
| Diffuse albedo     | $\sigma(\mathbf{x})$     | Identity                | 3          | 3            |
| Surface roughness  | $\rho(\mathbf{x})$       | Identity                | 1          | 1            |

**Table 1:** A table showing inputs of the MLP and their encodings with input and output dimensions that we use in the light transport simulation. The inputs in the top correspond to the integration domain, while the inputs in the bottom are parameters fixed with respect to the integration.

gration method. Using the method described above, we are able to integrate an MLP over a single integration domain. In light transport, however, we typically solve many integrations simultaneously (e.g., per pixel). Our goal is to have a single MLP that could represent multiple integrands, and thus we condition the function represented by the MLP by additional parameters:  $g(x, y | \phi)$ , where  $\phi$  aggregates an arbitrary number of additional inputs to the MLP. These inputs are fixed with respect to a given integration domain, and thus after passing through the input layer (i.e., multiplying by the input weights), they are added to the bias. In other words, the MLP represents a higher dimensional function, where we integrate a 2D slice of that function for a fixed  $\phi$ . Table 1 shows an overview of inputs we use in the following applications. For the integration domain, it is important not to use any non-linear encoding that would result in non-linear borders in the planar subdivision.

#### 4.1. Ambient Occlusion

The first application is ambient occlusion, which expresses relative occlusion by the geometry itself regardless of light sources and materials. Formally, it can be described as an integral over the upper hemisphere:

$$AO(\mathbf{x} | r) = \int_{\omega_i \in \Omega} \frac{V(\mathbf{x}, \omega_i | r)(\mathbf{n}(\mathbf{x}) \cdot \omega_i)}{\pi} d\omega_i, \quad (10)$$

where  $\Omega$  is the upper hemisphere,  $r$  is a radius (a parameter), and  $V(\mathbf{x}, \omega_i | r)$  is a visibility function that returns 1 if there is no occluder from point  $\mathbf{x}$  along direction  $\omega_i$  at a distance less than or equal to  $r$ , and 0 otherwise. Notice that ambient occlusion is only dependent on the position  $\mathbf{x}$ , and the surface normal  $\mathbf{n}$  is a function of  $\mathbf{x}$ . We train an MLP to approximate the integrand which can be used as a control variate:

$$g(\omega_i | \mathbf{x}, \mathbf{n}(\mathbf{x})) \approx f(\mathbf{x}, \omega_i | r) = \frac{V(\mathbf{x}, \omega_i | r)(\mathbf{n}(\mathbf{x}) \cdot \omega_i)}{\pi}. \quad (11)$$

We use the surface normal  $\mathbf{n}$  as an explicit additional guiding input, encoded by the one-blob encoding in spherical coordinates. Since ambient occlusion depends only on geometry, we do not include the material parameters (e.g., albedo or roughness).

We need to embed the incoming direction  $\omega_i$  into the unit square to use our analytic integration. The direction is typically converted to normalized spherical coordinates in world space. Since an MLP can provide arbitrary values on the integration domain, we need to ensure that the probability density function we use for sampling is



non-zero across the whole integration domain to match the analytic integration in Equation 1. The spherical system, however, covers the whole sphere while the Monte Carlo integration used for the residual integral accounts for the upper hemisphere (e.g., using the cosine-weighted sampling). Therefore, we first transform the direction to local space, and then we convert it to hemispherical coordinates (i.e., the azimuthal angle and the height) such that the integration domain is the hemisphere in both cases. The result of the analytic integration needs to be scaled by the volume of the integration domain (i.e.,  $2\pi$  in the case of the upper hemisphere).

## 4.2. Direct Illumination

The second application is direct lighting with rectangular area light source  $A$  such as an emissive rectangle (or parallelogram) defined by an anchor vertex  $\mathbf{v}$  and two linearly independent vectors  $\mathbf{e}_1$  and  $\mathbf{e}_2$ , where each point  $\mathbf{y}$  has the form  $\mathbf{y} = \mathbf{v} + x\mathbf{e}_1 + y\mathbf{e}_2$  for  $(x, y) \in [0, 1]^2$ . This way, the problem of direct lighting can be formulated as an integral over the area of the light source:

$$L_o^{\text{direct}}(\mathbf{x}, \omega_o) = \int_{\mathbf{y} \in A} f_r(\mathbf{x}, \omega_{\mathbf{x} \rightarrow \mathbf{y}}, \omega_o) L_e(\mathbf{y}, \omega_{\mathbf{y} \rightarrow \mathbf{x}}) G(\mathbf{x}, \mathbf{y}) V(\mathbf{x}, \mathbf{y}) d\mathbf{y}, \quad (12)$$

where  $L_o^{\text{direct}}(\mathbf{x}, \omega_o)$  is outgoing radiance from point  $\mathbf{x}$  in direction  $\omega_o$ ,  $L_e(\mathbf{y}, \omega_{\mathbf{y} \rightarrow \mathbf{x}})$  is emitted radiance from point  $\mathbf{y}$  in direction  $\omega_{\mathbf{y} \rightarrow \mathbf{x}}$ ,  $f_r$  is the bidirectional reflectance function (BRDF),  $\omega_{\mathbf{x} \rightarrow \mathbf{y}} = \frac{\mathbf{y} - \mathbf{x}}{\|\mathbf{y} - \mathbf{x}\|}$  is a unit direction pointing from  $\mathbf{x}$  to  $\mathbf{y}$ ,  $G(\mathbf{x}, \mathbf{y}) = \frac{(\mathbf{n}(\mathbf{x}) \cdot \omega_{\mathbf{x} \rightarrow \mathbf{y}})(\mathbf{n}(\mathbf{y}) \cdot \omega_{\mathbf{y} \rightarrow \mathbf{x}})}{\|\mathbf{y} - \mathbf{x}\|^2}$  is the geometry term, and  $V(\mathbf{x}, \mathbf{y})$  is the visibility function indicating binary visibility between points  $\mathbf{x}$  and  $\mathbf{y}$ . Similarly as for ambient occlusion, we want to approximate the integrand by an MLP:

$$\mathbf{g}(\mathbf{x}, \mathbf{y} | \mathbf{x}, \omega_o, \mathbf{n}(\mathbf{x}), \sigma(\mathbf{x}), \rho(\mathbf{x})) \approx \mathbf{f}(\mathbf{x}, \mathbf{v} + x\mathbf{e}_1 + y\mathbf{e}_2, \omega_o) = \mathbf{f}(\mathbf{x}, \mathbf{y}, \omega_o). \quad (13)$$

$$\mathbf{f}(\mathbf{x}, \mathbf{y}, \omega_o) = f_r(\mathbf{x}, \omega_{\mathbf{x} \rightarrow \mathbf{y}}, \omega_o) L_e(\mathbf{y}, \omega_{\mathbf{y} \rightarrow \mathbf{x}}) G(\mathbf{x}, \mathbf{y}) V(\mathbf{x}, \mathbf{y}), \quad (14)$$

In contrast to monochromatic ambient occlusion, radiance is wavelength dependent, and thus the whole integrand is a vector function representing RGB values. Radiance is also view-dependent, and thus we need to include the outgoing direction  $\omega_o$  as an input encoded by the one-blob encoding in spherical coordinates. For the primary hits, the outgoing direction is uniquely determined by the position, and thus can be omitted. We also use diffuse albedo  $\sigma(\mathbf{x})$  and surface roughness  $\rho(\mathbf{x})$  as additional guiding inputs since the integrand includes the BRDF. The result of the analytic integration needs to be scaled by the area of the light source.

## 4.3. Global Illumination

Last, we discuss global illumination, described by the rendering equation, which is typically solved by path tracing:

$$L_o(\mathbf{x}, \omega_o) = L_e(\mathbf{x}, \omega_o) + \int_{\omega_i \in \Omega} f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\mathbf{n}(\mathbf{x}) \cdot \omega_i) d\omega_i, \quad (15)$$

where  $L_o(\mathbf{x}, \omega_o)$  is outgoing radiance from point  $\mathbf{x}$  in direction  $\omega_o$  and  $L_i(\mathbf{x}, \omega_i)$  is incoming (incidence) radiance at point  $\mathbf{x}$  from direction  $\omega_i$ . We approximate the integrand by an MLP:

$$\mathbf{g}(\omega_i | \mathbf{x}, \omega_o, \mathbf{n}(\mathbf{x}), \sigma(\mathbf{x}), \rho(\mathbf{x})) \approx \mathbf{f}(\mathbf{x}, \omega_i, \omega_o) = f_r(\mathbf{x}, \omega_i, \omega_o) L_i(\mathbf{x}, \omega_i) (\mathbf{n}(\mathbf{x}) \cdot \omega_i), \quad (16)$$

At first glance, everything appears to work correctly; nonetheless, there is a few caveats. The MLP can learn the radiance field quite credibly even from the noisy estimates (thanks to the Adam optimizer [KB14]). Note that in this case, we need to learn the whole 7D dimensional triple product, not only incidence radiance. However, for control variates, we also need access to  $\mathbf{f}(\mathbf{x}, \omega_i, \omega_o)$ , which is a product including  $L_i(\mathbf{x}, \omega_i)$ , which itself is a nested integral, to integrate the residual integral. Typically, we have only a one-sample estimate of  $L_i(\mathbf{x}, \omega_i)$  available, which could be very far from the actual value.

We also need to make sure that the domains of analytic integration and the residual integration are the same. In path tracing, we sample the incoming direction, for example, according to BRDF, which might not cover the whole hemisphere (e.g., highly specular surfaces). Consequently, some parts of the hemisphere have zero probability of being sampled where the original integrand has zero contribution. This might not be the case for the MLP as the MLP for such samples might be non-zero, and thus such samples are included in the analytic integration. Thus, we need to sample according to a probability density that is non-zero on the whole hemisphere to match the analytic integration. Note that for the sake of brevity, we do not consider transparent materials (i.e., refraction).

## 5. Experiments and Results

We discuss the experiments we conducted to evaluate the control variates using our geometric integration. In all experiments, we assume that  $\alpha = 1$  (i.e., perfect correlation). We use mean squared error (MSE) as an error metric as well as a loss function in all cases.

### 5.1. Analytic Functions

As a proof-of-concept, we implemented our method in PyTorch, integrating a set of analytic 2D functions [CKK18, Bur20]. We compare control variates with our geometric integration (GI-NCV) to control variates with the automatic integration [LMW21, LYZ\*24] (AI-NCV) and vanilla Monte Carlo. We use an MLP with two hidden layers with 32 neurons in each layer for both control variates methods. For GI-NCV, we use the ReLU activation function, while for AI-NCV, we use the sigmoid activation function, which has non-zero second order gradients compared to ReLU. We pre-train each MLP using 5000 epochs. The results are summarized in Figure 5. The convergence graphs show the variance estimates up to for 1024 samples, averaged over 128 trials for each sample count. GI-NCV and AI-NCV are competitive, the only difference is due to the different activation functions: GI-NCV with ReLU approximates better discontinuous step-like functions (e.g., the disk function), while AI-NCV better approximates smooth functions (e.g., the bilinear function). In Figure 6, we show the number of faces of subdivisions corresponding to the bilinear function for different

MLP configurations: the number of hidden layers and the number of neurons per hidden layer. Note that the exact number of faces depends on a particular MLP weights and biases.

## 5.2. Light Transport Simulation

We integrated our method into our in-house renderer and MLP framework to evaluate it on the light transport problems discussed in Section 4. We use an MLP with two hidden layers, each with 32 neurons, and the ReLU activation function. We employ multiresolution hashgrid encoding [MESK22] with a base resolution of 64, four levels, and two features per level to encode the position. All images are rendered at a resolution of  $1024 \times 1024$  (unless stated otherwise).

Our rendering pipeline consists of three phases. (1) We pre-train the MLP before it is actually used as a control variate. (2) Once the MLP is pre-trained, we integrate the MLP using our analytic integration. (3) We use Monte Carlo integration to integrate the residual integral. We use one training sample per pixel in each epoch. Training samples can be used either only for training or for both training and rendering, introducing correlation but not the bias (see Appendix A).

In Figure 7, we present the results when we use the training samples only for training to determine whether using control variates brings any improvement compared to vanilla Monte Carlo regardless of the training overhead. The corresponding curves in the convergence graph are parallel to those of Monte Carlo shifted by some offset in the logarithmic scale. We observe that the control variates can reduce the error for direct lighting and ambient occlusion but not for the indirect lighting (we omit the direct lighting component from the global illumination discussed in Section 4 to emphasize the indirect component) due to the noisy estimates of incidence radiance. We analyze this problem in Appendix B.

In Figure 8, we show an equal-sample-count comparison that takes into account both training samples (used for both training and rendering) and the rendering samples. We offset the corresponding curve by the number of training samples in the convergence graph (see Figure 8). Our method using control variates can achieve lower error compared to vanilla Monte Carlo. As a sanity check, we show that Monte Carlo integration of the MLP converges to our analytic integration, which is a necessary condition to avoid bias.

**Implementation Details** Both the renderer and the MLP framework are implemented in HIP (ROCm 6.3), using HIPRT 2.5 [MKVH24] and FP32 for the weights and the hashgrid latent vectors, optimized by the Adam optimizer [KB14], for the MLP. The geometric integration is implemented in a separate kernel, where each pixel is processed by a single thread. Each thread requires relatively large memory space to store DCELS in order to compute the analytic integration (see Section 3.2). To reduce the memory usage, we use persistent threads, and thus the allocated memory for one thread will be reused for multiple pixels. We use 16-bit integers for the half-edge indices since there is no more than 65k half-edges in a single DCEL. We store weights of the MLP in shared memory as the weights are common for all threads. We use an exponential moving average [MRNK21] for the weights (but

not for the latent vectors stored in the hashgrid) in order to make training more stable.

**Comparison with Automatic Integration** We implemented a simple renderer in PyTorch to compare our method (GI-NCV) with the automatic integration [LMW21, LYZ\*24] (AI-NCV), which requires autodiff. We use the same setting as in the HIP framework except the hashgrid encoding; we use frequency encoding [MST\*20] instead of the hashgrid [MESK22] for both methods. For AI-NCV, we use sigmoid as an activation function instead of ReLU for the same reasons discussed in Section 5.1, and one-blob encoding [MMR\*19] for the integration domain (spherical domain in the case of ambient occlusion). We use 5000 epochs to pre-train the MLP of each method, and the images are rendered at a resolution of  $256 \times 256$ . In Figure 9, we can observe that training the MLP directly (not via the antiderivative as the automatic integration does) using our approach can achieve better approximation, and thus lower overall error.

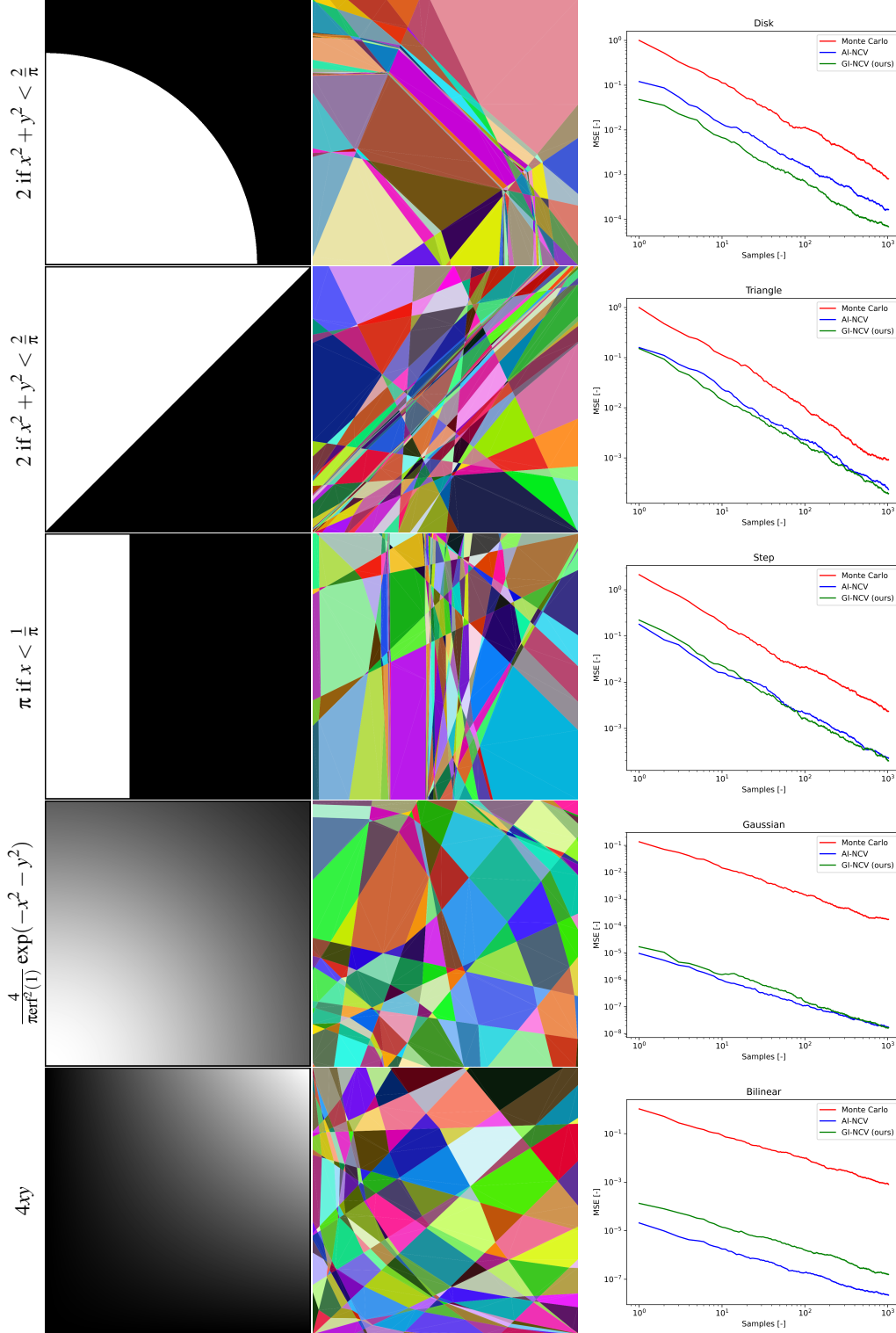
## 5.3. Discussion and Limitations

In this section, we discuss the limitations of the proposed method. Similar to the automatic integration [LYZ\*24], the geometric integration is limited to low dimensions (two dimensions in our case). In contrast to high-dimensional primary samples space control variates [MRKN20, CJMn21, SGH\*22], we can employ our method only for simplified scenarios in light transport, not to fully-recursive path tracing, nor even to simpler techniques such as multiple importance sampling or Russian roulette.

The proposed integration method is a relatively expensive (both in time and space) part in our pipeline. Therefore, we amortize its cost through using many samples such that it is evaluated only once after the MLP is pre-trained. Note that this is not possible for real-time scenarios, where the MLP is continuously trained. The time complexity depends on the resolution and the MLP architecture. To reduce the complexity, we use a smaller MLP and more complex hashgrid, that does not influence the integration algorithm.

In Table 2, we show the breakdown of absolute times needed for different phases measured on AMD Radeon PRO W7900. We achieve the same error at slightly higher times compared to vanilla Monte Carlo. With a large number of samples, the integration part becomes marginal, but the training and inference becomes dominant. We need to use significantly more training data to achieve variance reduction as we evaluate the MLP on the primary hits in contrast to the neural radiance cache [MRNK21], which hides inaccuracies by deeper bounces. We tried to use FP16 for the network weights and latent vectors in the hashgrid, but with the larger hashgrid, we can observe the problem of vanishing gradients. We leave further optimizations of training, inference, and integration as a future work.

Another limitation specific to our approach is that we cannot use non-linear encoding for a 2D integration domain to keep the lines straight. However, we can use arbitrary encoding for other inputs (e.g., a larger hashgrid). In Figure 9, we can see that our method achieves lower error even without encoding than the automatic integration [LMW21, LYZ\*24] with one-blob encoding in the spherical domain.



**Figure 5:** Comparison of the vanilla Monte Carlo method, Monte Carlo with neural control variates using the automatic integration [LMW21, LYZ\*24] (AI-NCV), and Monte Carlo with neural control variates using our geometric integration (GI-NCV), integrating simple analytic 2D functions [CKK18, Bur20]: function visualization (left), geometric subdivision of our method highlighting piecewise affine regions (middle), and the convergence graphs (right). We used 128 trials to estimate the variance and up to 1024 samples for the convergence graphs. Note that all functions are normalized such that they integrate to one.



**Figure 6:** The number of faces of subdivisions of the bilinear function for different MLP configurations: the number of hidden layers and the number of neurons in each hidden layer.

|                      | Monte Carlo          | GI-NCV (ours)        |
|----------------------|----------------------|----------------------|
| SPP [-]              | 32768                | 1024 + 18432         |
| MSE [-]              | $4.54 \cdot 10^{-5}$ | $4.53 \cdot 10^{-5}$ |
| Render time [s]      | 136                  | 78                   |
| Train time [s]       | -                    | 33                   |
| Integration time [s] | -                    | 10                   |
| Inference time [s]   | -                    | 34                   |
| Total time [s]       | 136                  | 155                  |

**Table 2:** Breakdown of times for equal-quality comparison of vanilla Monte Carlo and neural control variates with our geometric integration (GI-NCV) for ambient occlusion in the Bistro scene measured on AMD Radeon PRO W7900. For a larger number of samples per pixel, the integration part becomes marginal compared to training and inference, while for a relatively a smaller number of training samples per pixel (epochs), the training phase becomes the major bottleneck.

## 6. Conclusion and Future Work

We proposed a method for integrating the multilayered perceptron with a continuous piecewise linear activation function over a 2D domain. We formulated the integration problem as geometric subdivision of the integration domain, and we showed how to solve such a problem by means of computational geometry. The proposed method can be used in combination with control variates to reduce variance in Monte Carlo integration. We demonstrated the application of our method in physically-based rendering, achieving lower error than vanilla Monte Carlo. We also showed that our method is a viable alternative to the automatic integration [LMW21,LYZ\*24], achieving similar or even lower error.

There are several interesting directions for future work. There are two inherent problems to control variates that we have not touched in the paper: how to estimate  $\alpha$  and how to importance sample

the residual integral. We assumed that  $\alpha = 1$ , which corresponds to perfect correlation between  $f(x)$  and  $g(x)$ , which is not true in practice. We employed a variant of uniform sampling (e.g., the area light source in direct lighting) of the integration domain; however, it would be beneficial to design an importance sampling technique that would prioritize samples where the difference between the integrand and the approximation by the MLP is high. Both problems could be tackled by additional neural networks similarly to the work by Muller et al. [MRKN20]. Notice that the difference could be negative, which brings another source of variance [BXB\*24].

Another interesting idea is to use the geometric subdivision to sample the integrand itself. This would work in two steps. In the first step, we would sample a region based on the integral of the associated affine function on that region. In the second step, we sample a point within the region (the affine function is invertible and can be easily normalized). Last, we would like to evaluate the proposed method in the context of geometric processing to solve Poisson or Laplace equations, using algorithms such as walk-on-spheres [SC20] or walk-on-boundary [SCJ\*23].

## Acknowledgements

We thank Shin Fujieda, Chih-Chen Kao, and Atsushi Yoshimura for their continuous help and valuable feedback.

## Appendix A: Correlated Training and Rendering Samples

The estimator used in Section 5.2 can be defined as a weighted average of two unbiased estimators  $\langle I_{MC} \rangle$  and  $\langle I_{CV} \rangle$ :

$$\langle I \rangle = \frac{m}{m+n} \langle I_{MC} \rangle + \frac{n}{m+n} \langle I_{CV} \rangle, \quad (17)$$

$$\langle I_{MC} \rangle = \frac{1}{m} \sum_{i=1}^m \frac{f(x_i)}{p(x_i)} \quad (18)$$

$$\langle I_{CV} \rangle = G + \frac{1}{n} \sum_{i=1}^n \frac{f(x_{m+i}) - g(x_{m+i})}{p(x_{m+i})} \quad (19)$$

where  $m$  is the number of training samples (used also for rendering) and  $n$  is the number of rendering samples. We can easily show that the estimator is unbiased:

$$\mathbb{E}[\langle I \rangle] = \mathbb{E} \left[ \frac{m}{m+n} \langle I_{MC} \rangle \right] + \mathbb{E} \left[ \frac{n}{m+n} \langle I_{CV} \rangle \right], \quad (20)$$

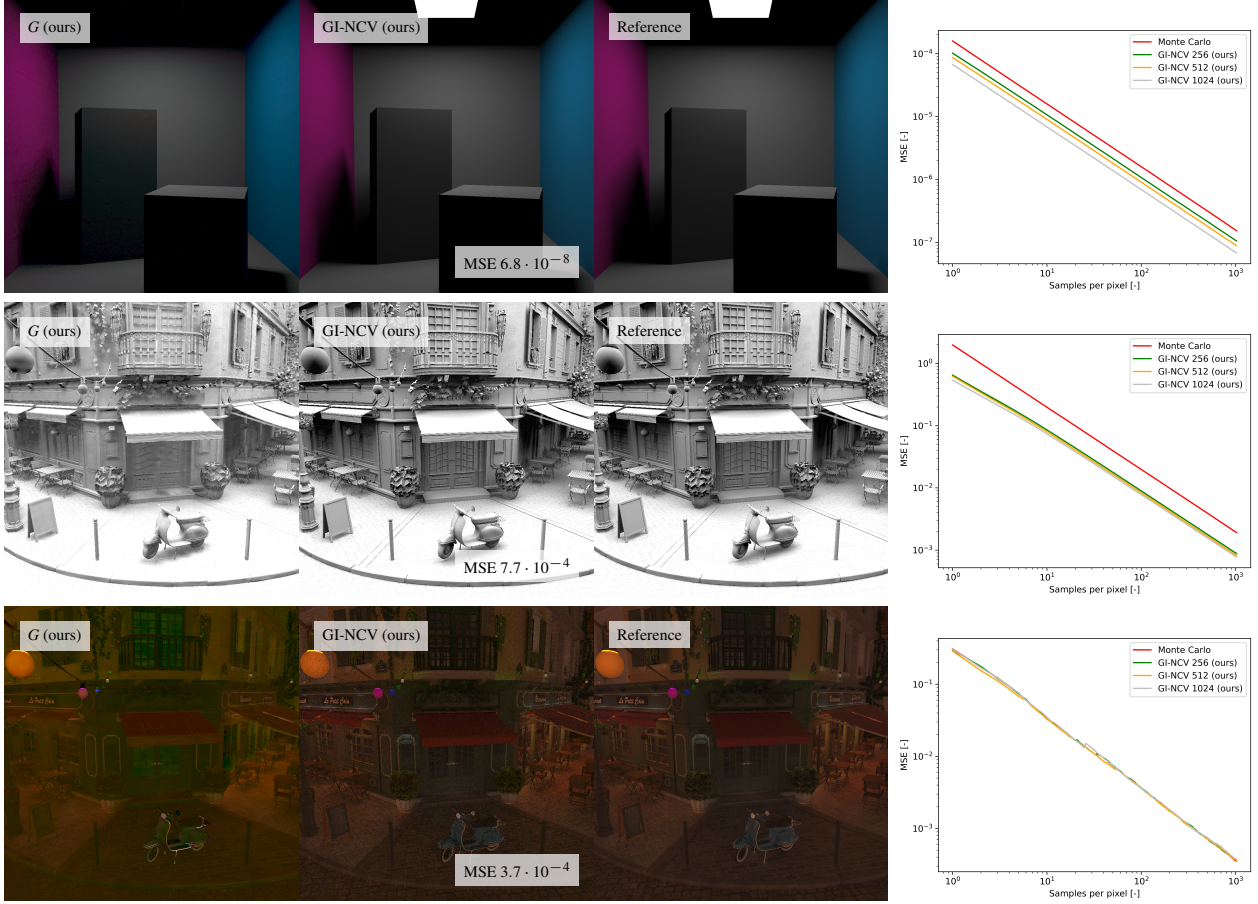
$$= \frac{mF}{m+n} + \frac{n}{m+n} \mathbb{E} \left[ G + \frac{1}{n} \sum_{i=1}^n \frac{f(x_{m+i}) - g(x_{m+i})}{p(x_{m+i})} \right], \quad (21)$$

$$= \frac{mF}{m+n} + \frac{n}{m+n} \left( G + \mathbb{E} \left[ \frac{1}{n} \sum_{i=1}^n \frac{f(x_{m+i}) - g(x_{m+i})}{p(x_{m+i})} \right] \right), \quad (22)$$

$$= \frac{mF}{m+n} + \frac{n}{m+n} (G + F - G) = \frac{(m+n)F}{m+n} = F. \quad (23)$$

Note that we use a 1D function to make notation simpler, but the same reasoning can be applied to higher-dimensional function such those used in Section 4.





**Figure 7:** Comparison of control variates with our geometric integration (GI-NCV) and vanilla Monte Carlo for different numbers of training samples per pixel: direct lighting in the Cornell box (top), ambient occlusion in the Bistro scene (center), and indirect lighting in the Bistro scene (bottom). Notice that the control variates do not help in reducing the error for indirect lighting due to the noisy estimates of incidence radiance (see Section 4). The images correspond to the result using 1024 training samples per pixel (epochs).

## Appendix B: Noisy estimates of the Integrand

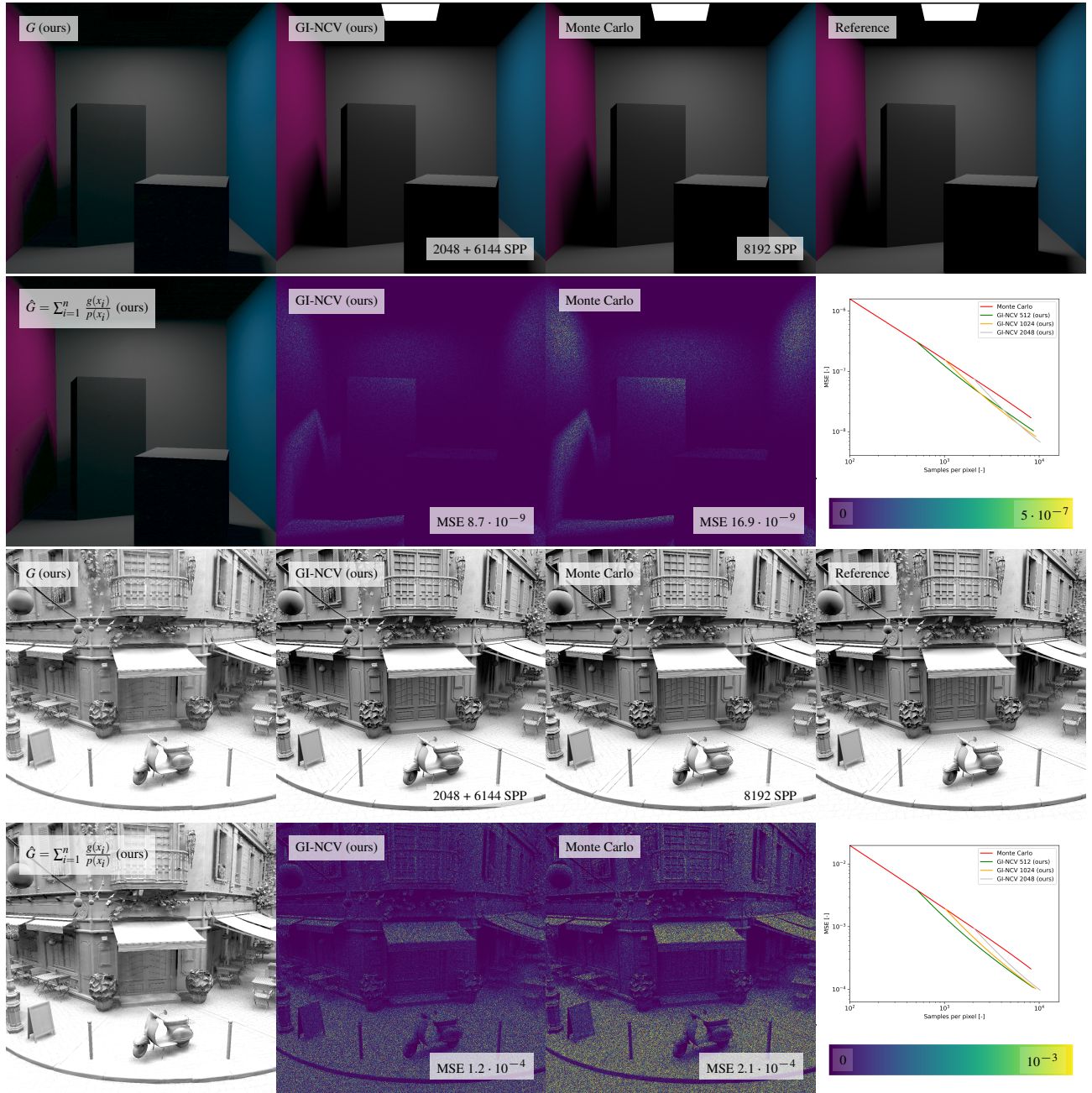
We conducted an experiment with a simple analytic function mimicking the setting in indirect lighting. We integrate function  $f(x) = \int_0^1 \int_0^1 8xyz dydz = 2x$  with its noisy one-sample estimates  $\tilde{f}(x) = 8xy_i z_i \approx f(x)$  for samples  $(y_i, z_i)$ . We use the same function scaled as a control variate  $g(x) = 0.9 \cdot f(x)$ , which is almost a perfect control variate, avoiding error due to insufficient approximation. Figure 10 shows convergence graphs with noisy estimates and ground truth. While control variates with noisy estimates provide practically no improvement compared to vanilla Monte Carlo, the ground truth function values provide expected improvement, which is aligned with our light transport experiments.

## References

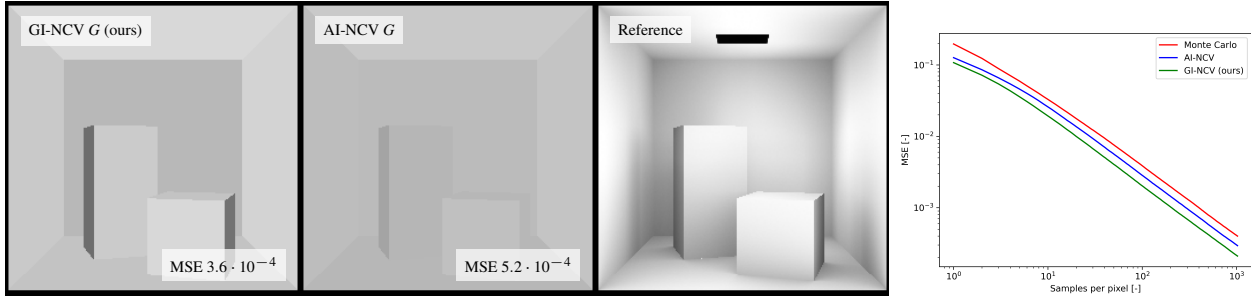
- [BCKO08] BERG M. D., CHEONG O., KREVELD M. V., OVERMARS M.: *Computational Geometry: Algorithms and Applications*, 3rd ed. ed. Springer-Verlag TELOS, Santa Clara, CA, USA, 2008. 2, 3, 4
- [Bur20] BURLEY B.: Practical hash-based owen scrambling. *Journal of*

*Computer Graphics Techniques (JCGT)* 10, 4 (December 2020), 1–20. 7, 9

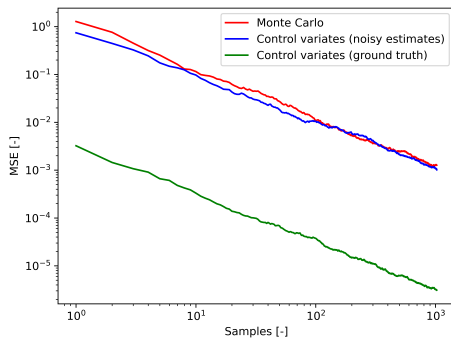
- [BxB\*24] BELHE Y., XU B., BANGARU S. P., RAMAMOORTHY R., LI T.-M.: Importance sampling brdf derivatives. *ACM Trans. Graph.* 43, 3 (Apr. 2024). 10
- [CJMn21] CRESPO M., JARABO A., MUÑOZ A.: Primary-space adaptive control variates using piecewise-polynomial approximations. *ACM Trans. Graph.* 40, 3 (July 2021). 2, 8
- [CKK18] CHRISTENSEN P., KENSLER A., KILPATRICK C.: Progressive multi-jittered sample sequences. *Computer Graphics Forum* 37, 4 (2018), 21–33. 7, 9
- [DKHD25] DEREVIANNYKH M., KLEPIKOV D., HANIKA J., DACHS-BACHER C.: Neural two-level monte carlo real-time rendering. *Computer Graphics Forum* (2025). 3
- [FKH23] FUJIEDA S., KAO C. C., HARADA T.: Neural Intersection Function. In *High-Performance Graphics - Symposium Papers* (2023), Bikker J., Gribble C., (Eds.), The Eurographics Association. 2
- [FYH23] FUJIEDA S., YOSHIMURA A., HARADA T.: Local Positional Encoding for Multi-Layer Perceptrons. In *Pacific Graphics Short Papers and Posters* (2023), Chaine R., Deng Z., Kim M. H., (Eds.), The Eurographics Association. 2



**Figure 8:** Equal-sample-count comparison of neural control variates using our geometric integration (GI-NCV) and vanilla Monte Carlo for direct lighting in the Cornell box (top) and ambient occlusion in the Bistro scene (bottom). In the left column, we show the results of the analytic integration (left - first and third rows) and Monte Carlo integration (left - second and fourth rows) of the MLP, demonstrating that Monte Carlo converges to the analytic solution. The images correspond to the result using 2048 training samples per pixel (epochs). For the sake of fair comparison, the lines of GI-NCV in the convergence graphs are offset by the corresponding training samples that are used for both for training and rendering.



**Figure 9:** Comparison of control variates with our geometric integration (GI-NCV) and the automatic integration [LMW21, LYZ\*24] (AI-NCV) for ambient occlusion in the Cornell box. Our method allows us to train the MLP directly, and thus it can approximate ambient occlusion more credibly, achieving lower error overall.



**Figure 10:** Comparison of control variates using ground truth  $f(x) = 2x$  and noisy estimates  $\tilde{f}(x) = 8xy_i z_i$ , using  $g(x) = 0.9 \cdot f(x)$  as a control variate in both cases.

- [HBBB23] HUMAYUN A. I., BALESTRIERO R., BALAKRISHNAN G., BARANIUK R.: Splincam: Exact visualization and characterization of deep network geometry and decision boundaries. In *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2023), pp. 3789–3798. [2](#), [3](#)
- [HCZ21] HADADAN S., CHEN S., ZWICKER M.: Neural radiosity. *ACM Trans. Graph.* 40, 6 (Dec. 2021). [2](#)
- [HN98] HESTERBERG T. C., NELSON B. L.: Control variates for probability and quantile estimation. *Management Science* 44, 9 (1998), 1295–1312. [2](#)
- [KB14] KINGMA D., BA J.: Adam: A method for stochastic optimization. *International Conference on Learning Representations* (12 2014). [7](#), [8](#)
- [KV90] KEMNA A., VORST A.: A pricing method for options based on average asset values. *Journal of Banking & Finance* 14, 1 (1990), 113–129. [2](#)
- [KVG\*19] KONDAPANENI I., VEVODA P., GRITTMANN P., SKŘIVAN T., SLUSALLEK P., KŘIVÁNEK J.: Optimal multiple importance sampling. *ACM Trans. Graph.* 38, 4 (July 2019). [2](#)
- [Liu24] LIU Y.: Neural networks are integrable, 2024. [2](#)
- [LMW21] LINDELL D. B., MARTEL J. N. P., WETZSTEIN G.: Autoint: Automatic integration for fast neural volume rendering. In *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2021), pp. 14551–14560. [2](#), [7](#), [8](#), [9](#), [10](#), [13](#)
- [LYZ\*24] LI Z., YANG G., ZHAO Q., DENG X., GUIBAS L., HARIHARAN B., WETZSTEIN G.: Neural control variates with automatic integration. In *ACM SIGGRAPH 2024 Conference Papers* (New York, NY, USA, 2024), SIGGRAPH '24, Association for Computing Machinery. [2](#), [7](#), [8](#), [9](#), [10](#), [13](#)
- [MESK22] MÜLLER T., EVANS A., SCHIED C., KELLER A.: Instant neural graphics primitives with a multiresolution hash encoding. *ACM Trans. Graph.* 41, 4 (July 2022). [2](#), [8](#)
- [MKVH24] MEISTER D., KULKARNI P., VASISHTA A., HARADA T.: Hipt: A ray tracing framework in hip. *Proc. ACM Comput. Graph. Interact. Tech.* 7, 3 (Aug. 2024). [8](#)
- [MMR\*19] MÜLLER T., MCWILLIAMS B., ROUSSELLE F., GROSS M., NOVÁK J.: Neural importance sampling. *ACM Transactions on Graphics (ToG)* 38, 5 (2019), 1–19. [8](#)
- [MRKN20] MÜLLER T., ROUSSELLE F., KELLER A., NOVÁK J.: Neural control variates. *ACM Trans. Graph.* 39, 6 (Nov. 2020). [2](#), [8](#), [10](#)
- [MRNK21] MÜLLER T., ROUSSELLE F., NOVÁK J., KELLER A.: Real-time neural radiance caching for path tracing. *ACM Trans. Graph.* 40, 4 (July 2021). [2](#), [8](#)
- [MST\*20] MILDENHALL B., SRINIVASAN P. P., TANCİK M., BARRON J. T., RAMAMOORTHY R., NG R.: Nerf: Representing scenes as neural radiance fields for view synthesis. In *ECCV* (2020). [8](#)
- [NRN\*23] NICOLET B., ROUSSELLE F., NOVÁK J., KELLER A., JAKOB W., MÜLLER T.: Recursive control variates for inverse rendering. *ACM Trans. Graph.* 42, 4 (July 2023). [2](#)
- [RJN16] ROUSSELLE F., JAROSZ W., NOVÁK J.: Image-space control variates for rendering. *ACM Trans. Graph.* 35, 6 (Dec. 2016). [2](#)
- [SC20] SAWHNEY R., CRANE K.: Monte carlo geometry processing: A grid-free approach to pde-based methods on volumetric domains. *ACM Trans. Graph.* 39, 4 (2020). [10](#)
- [SCJ\*23] SUGIMOTO R., CHEN T., JIANG Y., BATTY C., HACHISUKA T.: A practical walk-on-boundary method for boundary value problems. *ACM Trans. Graph.* 42, 4 (July 2023). [10](#)
- [SGH\*22] SALAÜN C., GRUSON A., HUA B.-S., HACHISUKA T., SINGH G.: Regression-based monte carlo integration. *ACM Trans. Graph.* 41, 4 (July 2022). [2](#), [8](#)
- [Sub21] SUBR K.: Q-net: A network for low-dimensional integrals of neural proxies. *Computer Graphics Forum* 40, 4 (2021), 61–71. [2](#)
- [TLG\*21] TAKIKAWA T., LITALIEN J., YIN K., KREIS K., LOOP C., NOWROUZEZAHRAI D., JACOBSON A., MCGUIRE M., FIDLER S.: Neural geometric level of detail: Real-time rendering with implicit 3D shapes. [2](#)
- [XLG\*24] XU B., LI T.-M., GEORGIEV I., HEDSTROM T., RAMAMOORTHY R.: Residual path integrals for re-rendering. *Computer Graphics Forum (Proceedings of EGSR)* 43, 4 (2024). [2](#)