Supervised Learning of How to Blend Light Transport Simulations

Hisanari Otsu, Shinichi Kinuwaki, and Toshiya Hachisuka

Abstract Light transport simulation is a popular approach for rendering photorealistic images. However, since different algorithms have different efficiencies depending on input scene configurations, a user would try to find the most efficient algorithm based on trials and errors. This selection of an algorithm can be cumbersome because a user needs to know technical details of each algorithm. We propose a framework which blends the results of two different rendering algorithms, such that a blending weight per pixel becomes automatically larger for a more efficient algorithm. Our framework utilizes a popular machine learning technique, regression forests, for analyzing statistics of outputs of rendering algorithms and then generating an appropriate blending weight for each pixel. The key idea is to determine blending weights based on classification of path types. This idea is inspired by the same common practice in movie industries; an artist composites multiple rendered images where each image contains only a part of light transport paths (e.g., caustics) rendered by a specific algorithm. Since our framework treats each algorithm as a black-box, we can easily combine very different rendering algorithms as long as they eventually generate the same results based on light transport simulation. The blended results with our algorithm are almost always more accurate than taking the average, and no worse than the results with an inefficient algorithm alone.

1 Introduction

Rendering based on light transport simulation is a popular approach for photorealistic image synthesis. Since such rendering algorithms solve the same governing equations

Hisanari Otsu · Toshiya Hachisuka
The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo, Japan e-mail: hi2p.perim@gmail.com, e-mail: thachisuka@siggraph.org

Shinichi Kinuwaki
Unaffiliated, Japan e-mail: ShinichiKinuwaki@gmail.com

(e.g., the rendering equation [Kaj86]), rendering with light transport simulation should give us the same result regardless of the choice of an algorithm. It is however well known that some algorithms are more efficient at rendering certain light transport effects. For example, photon density estimation [Jen96; HOJ08] is often efficient at rendering caustics, and Markov chain Monte Carlo algorithms [VG97; JM12] are considered efficient at resolving complex occlusions.

Because of the varying efficiency of different algorithms on different light transport effects, it is common practice to select an algorithm based on the type of light transport effect that one wants to render. In the movie industry, an artist often decomposes light transport effects into separate images, renders each with a most efficient algorithm, and composites the resulting images into the final one. Selecting appropriate algorithms and composting the results, however, can be difficult and cumbersome tasks. For selection, an artist either needs to know why some algorithms work well for some effects, or briefly tries all the available algorithms to see which one works well. For composition, an artist also needs to pay attention not to double count a certain type of paths such as caustics.

We propose a framework which automates this selection of the algorithms and composition of the resulting images. Our work is inspired by the superhuman performance of recent machine learning algorithms on classification tasks. We apply the same idea to select and composite two different rendering algorithms based on the classification of light transport effects. To be concrete, we use regression forests [Bre01] to learn the relationship between blending weights that minimize the error and the classification of light transport effects. While multiple importance sampling [VG95] also allows us to blend results of different rendering techniques, the key difference is that our framework treats each rendering algorithms as a black-box. Accordingly, our framework can be easily applied to very different algorithms such as SPPM and MLT without any algorithmic or theoretical modifications for each. To summarize, our contributions are:

- The use of machine learning to automatically blend the results of different rendering algorithms based on path types.
- A blending framework which is independent from how the underlying rendering algorithms work.
- First successful application of regression forests to light transport simulation.

2 Overview

Our goal is to blend the results of two different rendering algorithms such that the error of the blended result is as small as possible. Our algorithm is separated into two phases; the *training* phase and *runtime*. Fig. 1 illustrates the algorithm.

In the training phase, we use regression forests [Bre01] (Section 4) to learn the relationship between a feature vector of lighting effects extracted from the rendered images and the optimal weights for blending. For each training scene, we

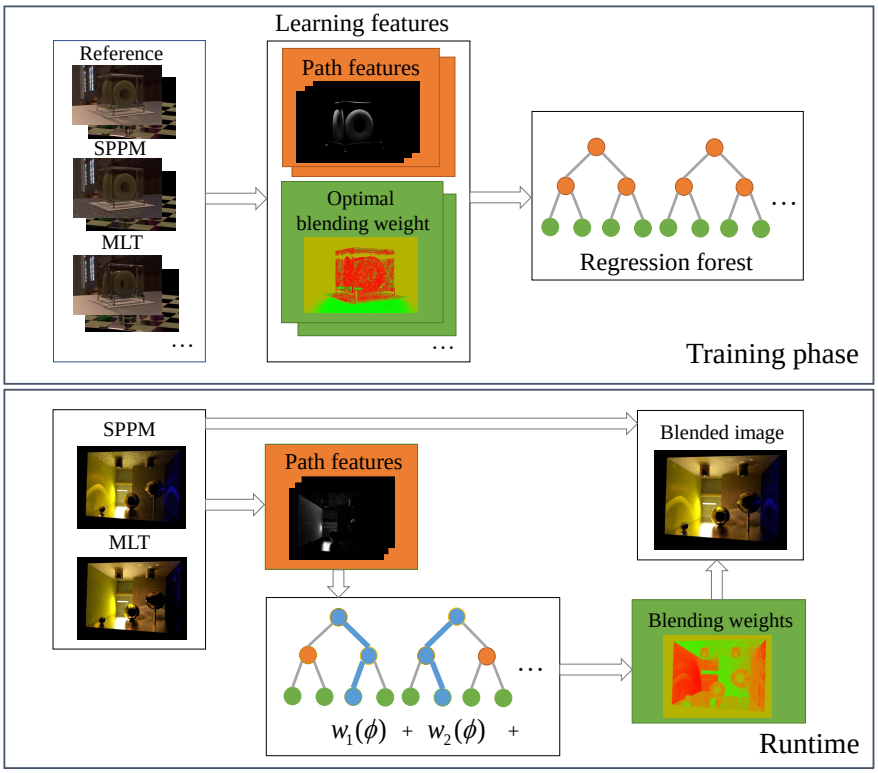

Fig. 1 General idea of blending the results of two different rendering algorithms using regression forests. In the training phase (top), we first calculate the optimal blending weight per pixel, given the reference image and rendered images with different approaches. These weights and the corresponding path features become one training sample for the regression forest for each scene. We iterate this process for various scenes. Our framework thus learns the relationship between input path features and optimal weights during this learning phase. At runtime (bottom), the trained regression forest returns approximated optimal blending weights based on path features of a new scene.

render the reference solution, and the two images with both algorithms allocating the same rendering time. Based on the rendered images, we extract *path features* as the relative pixel contributions of different light transport paths according to Heckbert's notation [Hec90]. Modern shader languages often support the same mechanism [GSK*10]. We then calculate the *optimal blending weights* based on the reference solution and the results of the two different rendering algorithms. The optimal blending weight is defined such that the error of the blended result is minimized at each pixel. A pair of path features and the optimal blending weight forms one training sample for regression forests. If a rendering algorithm is based on Monte Carlo methods (which is the case in our experiments), we generate multiple training samples for the same scene in order to avoid the influence of the randomness of rendered images.

At runtime, we use the trained regression forest to approximate the optimal blending weights for a given new scene. The path features extracted from the rendered images are used to traverse the regression forest to obtain the blending weights. The final result is a blended image with the obtained weights. Since the trained regression forest expresses the relationship between path features and the optimal weights, a blended image is expected to have small error, even for a scene that was not included in the training phase.

3 Automatic Blending with Path Features

3.1 Path Features

Our definition of a feature vector for rendering algorithms is inspired by how artists decompose a rendered image into several images with specific lighting effects for each. In order to define the feature vectors, we begin with the formulation of the light transport known as the path integral formulation [Vea98]. According to the formulation, the pixel intensity I observed at each pixel is expressed as

$$I = \int_{\Omega} f(\bar{x}) d\mu(\bar{x}), \tag{1}$$

where $\bar{x}$ is a light transport path, f is the measurement contribution function, and μ is the path measure. Ω is the space of paths of all different path lengths.

The path space Ω can be partitioned into a union of disjoint spaces according to the classification by Heckbert [Hec90]:

$$\Omega = \Omega_{\mathrm{LDE}} \cup \Omega_{\mathrm{LSE}} \cup \Omega_{\mathrm{LDSE}} \cup \Omega_{\mathrm{LSDE}} \cup \cdots, \tag{2}$$

where each Ω_* is a subspace of Ω defined with the paths represented by the Heckbert's notation $*$. For instance, the subspace Ω_{LDSE} with path length 3 is defined as a set of paths $\bar{x} = \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3$ where $\mathbf{x}_0$ is on a sensor, $\mathbf{x}_1$ is on a diffuse surface, $\mathbf{x}_2$ is on a specular surface, and $\mathbf{x}_3$ is on an emitter. A glossy interaction is classified to either D or S depending on its BRDF.

We thus define a part of the intensity I_* contributed only with the subspace Ω_* as

$$I_* = \int_{\Omega_*} f(\bar{x}) d\mu(\bar{x}). \tag{3}$$

Since the partition in Equation 2 is disjoint, the pixel intensity I is additive:

$$I = I_{\mathrm{LDE}} + I_{\mathrm{LSE}} + I_{\mathrm{LDSE}} + I_{\mathrm{LSDE}} + \cdots. \tag{4}$$

We thus define the *path features* ϕ as a vector of the intensities I_* relative to I:

$$\phi \equiv \frac{(I_{\mathrm{LDE}}, I_{\mathrm{LSE}}, I_{\mathrm{LDSE}}, I_{\mathrm{LSDE}}, \dots)}{I}. \tag{5}$$

The definition uses relative intensities such that ϕ is independent from the absolute intensity. We fixed the maximum path length to ten, which makes ϕ a $2^{(10-1)} = 512$ dimensional feature vector. The training phase uses an *estimate* $\hat{\phi}$ instead of the analytical value of ϕ for a given rendering time. We selected the number of dimensions such that all the data fits within the main memory. For instance, the scene rendered with 720p resolution requires a storage of $512 \times 1280 \times 720 \times 4$ bytes $\approx$ 1.8 gigabytes.

3.2 Optimal Blending Weights

In the training phase, we need to determine the optimal blending weight. This weight is used as an *answer* associated with a path feature vector. A pair of a path feature vector and the optimal blending weight thus becomes a training sample for supervised learning via regression forests.

We define the optimal blending weight w_{opt} that gives the minimum error as

$$w_{\mathrm{opt}} = \underset{w}{\operatorname{argmin}} \left| \left(w\hat{I}_\alpha + (1-w)\hat{I}_\beta \right) - I \right|, \tag{6}$$

where $\hat{I}_\alpha$ and $\hat{I}_\beta$ are the results of two different rendering algorithms α and β respectively, and I is the reference solution. This equation can be easily solved as

$$w_{\mathrm{opt}} = \frac{I - \hat{I}_\beta}{\hat{I}_\alpha - \hat{I}_\beta}. \tag{7}$$

If the solution of Equation 6 is not in the range of $[0,1]$, it is clamped to the nearest side such that $w_{\mathrm{opt}} \in [0,1]$. We apply this clamping such that the blending operation becomes a convex combination of the results. The blended result $w\hat{I}_\alpha + (1-w)\hat{I}_\beta$ is thus guaranteed to be more accurate than one of $\hat{I}_\alpha$ and $\hat{I}_\beta$ since

$$|w\hat{I}_\alpha + (1-w)\hat{I}_\beta - I| \leq \max(|\hat{I}_\alpha - I|, |\hat{I}_\beta - I|) \tag{8}$$

by definition if $w \in [0,1]$. Intuitively, this clamping process sets $w_{\mathrm{opt}} = 1$ when $\hat{I}_\alpha$ and $\hat{I}_\beta$ both either underestimate or overestimate I and $\hat{I}_\alpha$ is closer to I (vice versa for $\hat{I}_\beta$). If one of the $\hat{I}_\alpha$ and $\hat{I}_\beta$ underestimates and the other overestimates I, we set w_{opt} such that the blended result is exactly equal to I. Note that Equation 8 only guarantees that an error per pixel does not become worse, not the sum of errors over an image. For example, collecting pixels with worse errors (with $w = 0$ or $w = 1$) still satisfies Equation 8, but the sum of errors would increase.

The intensities $\hat{I}_\alpha$ and $\hat{I}_\beta$ are the relatively rough estimates of I in practice. If an algorithm is based on Monte Carlo ray tracing, an estimated intensity is an instance

of the random variable for each run. Using samples only from a single run of the algorithm causes overfitting to this specific run. For example, it might be that $\hat{I}_\alpha$ happens to be closer to I than $\hat{I}_\beta$ for the single run used in the training phase.

In order to deal with this issue, we use multiple training samples even for the same scene and the same algorithm. In fact, machine learning techniques (including regression forests) are naturally designed for dealing with such variations in the training data.

Problem Statement: Given the definitions above, the goal of our algorithm is to find a function w_{approx} such that

$$w_{\text{opt}} \approx w_{\text{approx}}(\phi), \tag{9}$$

for given path features ϕ and two rendering algorithms α and β. This function w_{approx} basically expresses the preference of the algorithm α over the other algorithm β for paths with a feature vector of ϕ. In order to learn w_{approx}, we use a machine learning algorithm called *regression forests*.

Difference Between Multiple Importance Sampling: Conceptually, our proposed blending approach is similar to multiple importance sampling (MIS) [VG95]. MIS also combines two or more different estimators to improve the efficiency of the combined estimator.

MIS combines multiple sampling strategies by decomposing the measurement contribution function f in Eq. 1 into a weighted sum of M different weights. The estimate of the pixel intensity I by MIS can be written as

$$I = \int_\Omega \sum_{t=1}^{M} w_t(\bar{x}) f(\bar{x}) d\mu(\bar{x}) = \sum_{t=1}^{M} \int_\Omega w_t(\bar{x}) f(\bar{x}) d\mu(\bar{x}) \tag{10}$$

$$\approx \sum_{t=1}^{M} \frac{1}{N_t} \sum_{i=1}^{N_t} w_t(\bar{x}_{t,i}) \frac{f(\bar{x}_{t,i})}{p_t(\bar{x}_{t,i})} \tag{11}$$

where $p_i(\bar{x})$ is the pdf of the i-th strategy, $w_i(\bar{x})$ is the weighting function satisfying $\sum_{t=1}^{M} w_i(\bar{x}) = 1$ for all $\bar{x} \in \Omega$ with $f(\bar{x}) \neq 0$, and $w_i(\bar{x}) = 0$ for all $\bar{x} \in \Omega$ with $p_i(\bar{x}) = 0$. In order to use MIS, however, we need to know the probability densities of path sampling techniques for arbitrary sample locations. Such information can be difficult to obtain without modifying an implementation or sometimes impossible due to the formulation of each algorithm.

4 Regression Forests

The basic idea of regression forests is to use a set of binary trees for approximating a multivariate function of the feature vector. This multivariate function expresses the relationship between feature vectors and the corresponding value. Each binary tree is called a *regression tree* where the inner nodes (split nodes) express branching

conditions on an input feature vector. Each regression tree takes an input feature vector and outputs a value associated with the corresponding leaf node. Regression forests return the average of the outputs of regression trees as the final output.

4.1 Construction

For the construction of regression forests, we need a large number of training samples which associate feature vectors (a set of path features) and output values (optimal weights). We generate these samples by rendering several training scenes. We then extract the path features and the corresponding optimal weights for each scene. The regression forest is trained to approximate the optimal weights even for a new scene, based only on the path features.

We define a training sample $t \equiv (\boldsymbol{\phi}^t, w^t_{\text{opt}}) \in \mathscr{T}$ as a tuple of path features $\boldsymbol{\phi}^t$ and the optimal weight w^t_{opt}. $\mathscr{T}$ is a set of all training samples. The construction process begins from the root node of the regression forest. Each step of the construction process recursively splits training samples into left and right nodes. We denote the subset of the training samples in the currently processed node as $T \subseteq \mathscr{T}$ and we start from $T = \mathscr{T}$. The algorithm is similar to a top-down construction of a kd-tree for ray tracing [MB90; Hav00].

Node Splitting: The construction process continues splitting the current node until the number of training samples in the current set T is smaller than a threshold, or the depth of the tree has reached the maximum depth. If the recursion terminates, the current node becomes a leaf node. Each leaf node stores the average over the set of the optimal weights in this node as w_{leaf}. This average weight approximates the optimal weight at runtime.

If the recursion continues, we split the current set of samples T into two disjoint subsets T_L and T_R according to a threshold θ and an index k of the path features:

$$T_L(\theta,k) = \{t \in T | \boldsymbol{\phi}^t(k) \geq \theta\} \tag{12}$$

$$T_R(\theta,k) = T \setminus T_{\text{L}}(\theta,k) \tag{13}$$

where $\boldsymbol{\phi}^t(k)$ is the k-th element of the path features $\boldsymbol{\phi}^t$. The threshold θ and the index k at each step are defined as $(\theta,k) = \text{argmax}_{\theta',k'} V(\theta',k',T)$, where $V(\theta',k',T) = \text{Var}(T) - \text{Var}(T_L(\theta',k')) - \text{Var}(T_R(\theta',k'))$. Here $\text{Var}(T)$ is the variance of the optimal weights in T. The function V is used to define the most discriminative pair of the threshold θ and the index of the path feature k according to the variance

4.2 Runtime

In our framework, we first render a given new scene with two different algorithms $\hat{I}_\alpha$ and $\hat{I}_\beta$ with the same computation time. We also extract the path features $\boldsymbol{\phi}$ according to the definition by Equation 5. Using these path features, we can now evaluate each trained regression tree by traversing down the tree according to the branching

condition defined in Equation 12, which eventually reaches a leaf node and the weight w_{leaf} is recorded in the leaf node. By repeating this process for all regression trees in the trained regression forest, we obtain a set of weights w_{leaf} recorded in the leaf nodes for each tree. We define $w_r(\phi)$ as the output of the r-th tree in the trained regression forest, given the path features ϕ. The approximated optimal weight $w_{\text{approx}}(\phi)$ with M trees is given as

$$w_{\text{approx}}(\phi) = \frac{1}{M}\sum_{r=1}^{M} w_r(\phi). \tag{14}$$

Blending at each pixel is $w_{\text{approx}}(\phi)\hat{I}_\alpha + (1 - w_{\text{approx}}(\phi))\hat{I}_\beta$. This evaluation process is repeated for all the pixels. The use of forests can alleviate the discontinuity of the resulting weights. Even if one tree suddenly returns a totally different value due to hard classification, it is likely that other trees still return similar weights. As a result, returning weights will be smoothly changing.

4.3 Refinement

A trained regression forest is sometimes too optimized for given training samples. In order to reduce overfitting, we follow the refinement technique for regression forests proposed by Ren et al. [RCWS14] and Ladický et al. [LJS*15]. The main idea is to split a set of training samples into two subsets and use one for constructing the structure of each tree while using the other for defining the outputs. After the construction step, we first discard the values w_{leaf} assigned to the leaf nodes while keeping the tree structure. The refinement process then updates w_{leaf} using the additional training samples.

For each additional training sample ϕ, we execute the evaluation of the tree until the evaluation process reach to the leaf node. After collecting the set of training samples Φ reached to the leaf node, the updated weight w_r^* can be computed as

$$w_r^* = \frac{1}{|\Phi|}\sum_{(\phi, w_{\text{opt}})\in\Phi} w_{\text{opt}}(\phi). \tag{15}$$

We iterate this refinement process for each tree in the regression forest using the sample training set for the refinement. Since the training samples are taken from the different portion of the training set independent of the samples assigned for the initial construction, the final weights associated to the leaf node could become more generic, which alleviates overfitting to the initial training set.

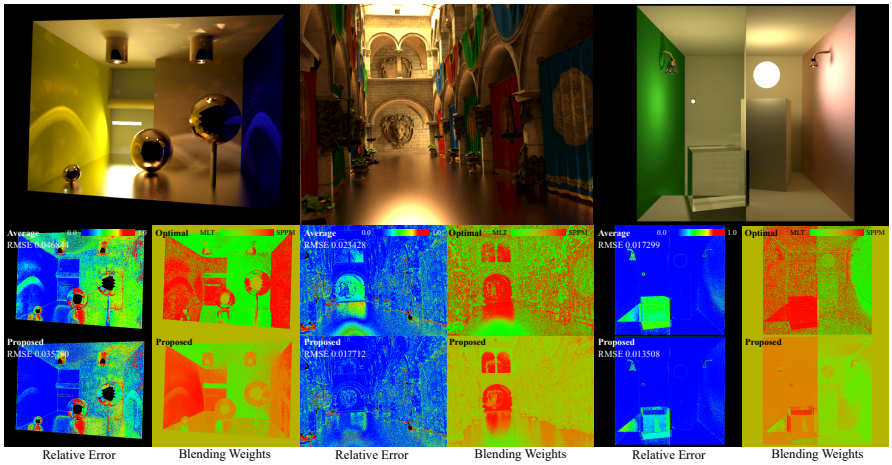

Fig. 2 Equal-time comparison (20 minutes) of the average and our automatic blending of the images rendered by SPPM [HJ09] and MLT [VG97] with manifold exploration [JM12]. We highlighted three scenes with different characteristics from our test cases (box, cryteck-sponza, and water). The top row shows the reference images. The bottom two rows visualize errors, the optimal blending weights, and the output blending weights of our framework. Depending on the types of lighting effects, the optimal blending weights for SPPM and MLT that result in the minimal error vary significantly. Simply taking the average of SPPM and MLT thus produces a suboptimal result in terms of RMS error.

5 Results

We selected the two combinations of the rendering algorithms to show the effectiveness of our framework: (1) stochastic progressive photon mapping (SPPM) [HJ09] and Metropolis light transport (MLT) [VG97] with manifold exploration [JM12] shown in Fig. 2, (2) SPPM and bidirectional path tracing (BDPT) [LW93; VG94] shown in Fig. 3. We chose these algorithms because both the algorithm and the performance are distinguishably different. One famous characteristic of SPPM is the ability to handle specular-diffuse-specular paths efficiently. A caustic that can be seen through a water surface is an example of such paths. MLT is based on Markov chain Monte Carlo sampling which utilizes a sequence of correlated samples that forms a Markov chain. The sequence of the samples is generated such that the resulting sample distribution follows an arbitrary user-defined target function such as the measurement contribution function. MLT is known to be effective for the scenes with complex occlusion. BDPT can utilize various sampling technique by the combination of paths traced from the sensor and the lights. These sampling techniques are combined with multiple importance sampling [VG95]. The combination of SPPM and BDPT would exhibit the good trade-off because BDPT is not efficient at handling specular-diffuse-specular paths [KD13] and while being more efficient at rendering diffuse surfaces [HPJ12; GKD*12].

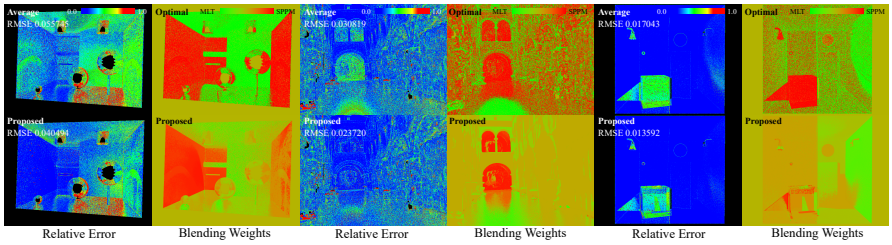

Fig. 3 Comparison of errors, the optimal weights and the approximations by our framework for the combination of BDPT and SPPM. The selection of the scenes and meaning of the images are same as Fig. 2. Similar to the combination of MLT and SPPM, our framework generally captures the preference to the scene according to the characteristics of the scenes, although some difference can be observed, e.g., preference to the scene dominated with specular material is weaker (*box* scene).

For the implementations of rendering algorithms, we used the Mitsuba renderer [Jak10]. Mutation techniques used for MLT are bidirectional, lens, caustic, multi-chain, and manifold perturbation [JM12]. All the images except for the reference images are rendered on a machine with Intel Core i7-4720HQ at 2.6 GHz. The training phase is computed with a machine with Intel Core i7-3970X at 3.5 GHz and 16 GB of main memory. We utilized only a single core for rendering in order to alleviate the difference of performance between SPPM and MLT according to the parallelization. In order to facilitate the future work, we publish our implementation on our website.

Training Samples: Our training set consists of 10 scenes with various characteristics in order to cover as many types of paths as possible. We render all the scenes with each rendering algorithm for 5, 10, 15, and 20 minutes. Each scene is rendered five times, in order to alleviate overfitting as discussed in Section 3.2. Given this whole training data, we generate a regression forest for each scene by excluding the scene from the training data. We thus have 10 different regression forests as a result. Each forest is tested against the corresponding scene that was excluded from its training. It is essentially leave-one-out cross-validation in machine learning.

While it is possible to have a single forest for all the training scenes and test this forest against the same set of scenes, we found that this kind of experiment is prone to overfit to the training scenes. Our regression forest consists of five trees and the maximum depth of each tree is 15. The construction time of the regression forest is 30 minutes.

Approximated Optimal Weights: Fig. 4 shows blending weights and RMS errors for selected five scenes with the combination of SPPM and MLT. Fig. 2 shows such results with visualization of the error per pixel for three other scenes. We compare approximated optimal weights via a trained regression forest with the average of five different runs for each scene. The blending weight is fixed to 0.5 when a pixel has no information on path features (e.g., background images). We blended two images rendered by SPPM and MLT by taking the average (Average) or by

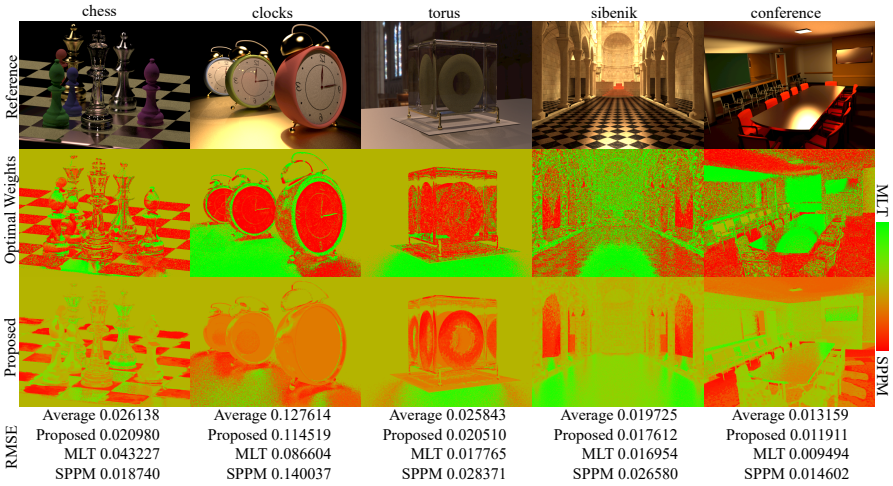

Fig. 4 Comparison of the optimal weights and the approximations by our framework for the selected five scenes combining MLT and SPPM. The first row shows the reference images. The bottom two rows visualize the optimal weights and the approximated weights via trained regression forests. For many scenes, our framework largely reproduces the optimal weights, without any information other than rough estimates of path features per pixel. The RMS errors between the blended images and the references are improved compare to taking the average (Average). We also show RMS errors for MLT and SPPM with the same total rendering time.

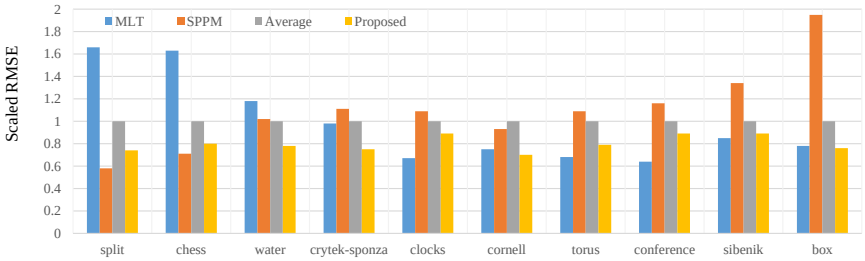

Fig. 5 Scaled RMS errors of MLT, SPPM, Average, and blending with our framework over 10 scenes. All the methods use the total rendering time of 20 minutes. The average and our blending spends 10 minutes for both MLT and SPPM, keeping the total rendering time equal to 20 minutes. We scaled RMS errors such that the average is always one. The scenes are sorted roughly according to the difference of RMS errors between MLT and SPPM.

using the approximated optimal weight per pixel (Proposed). The running time of our framework is less than 50 msec for all the scenes. The storage cost of our regression forest is 100 KB. Both the running time and the storage cost are independent of the geometric complexity of the scenes. We can see that optimal weights and weights suggested by our framework are very similar to each other in almost all the cases. Our framework thus successfully learned the preference of an algorithm only based on path features.

RMS Errors: Fig. 5 shows RMS errors for 10 scenes for the combination of MLT and SPPM. We plot RMS errors of MLT, SPPM, their average, and our blended result for each scene with the total rendering time of 20 minutes for all the methods. The plots are scaled such that the values for the average is one. We can observe that our blending is superior to the average in all scenes. The reduction of error by our blending is larger when the difference of RMS errors between SPPM and MLT is large. Moreover, the blended solution by our framework sometimes outperforms a better algorithm with the same total rendering time. Such a result is not trivial since our framework spends only half of the total rendering time for each algorithm. We should also note that just taking the average can in fact increase the error for the same reason (e.g., *Cornell* scene). In contrast, we did not find any such cases using our framework. This result supports that our framework can improve the robustness of light transport simulation in practice.

Effect of Tree Depth: The images in Fig. 6 show the approximated optimal weights for the *box* scene with different depths of the regression trees in the runtime. As the depth increases, we can observe that the preference to each technique becomes more explicit. Yet another observation is that the approximated weights are converged around the tree depth of 15. The graph in Fig. 6 shows the RMS error between the optimal weight and the approximated weight with our framework for this scene. We can observe that the RMS error converges around the depth of 15, and we found that it is similar for the other scenes as well. Along with the saturation of the weights, we thus conservatively set the tree depth to 15 in our experiments.

6 Discussion

6.1 Alternative to Blending

While we found that blending is a practical approach to combine different rendering algorithms, it is tempting to try *selecting* one of the different algorithms instead of *blending* such that we can spend all the allocated rendering time to one algorithm. This alternative solution, however, is not feasible for two major reasons. Firstly, as shown in Fig. 2, a better algorithm can change even within a single image. Even though MLT looks converged in many regions, it can entirely miss certain lighting effects such as specular reflections of caustics. As such, resolving all the effects by a single algorithm can take a significant amount of rendering time as compared to combining the results of two algorithms. Recent work on robust rendering algorithms are based on the same observation [HPJ12; GKD*12].

Secondly, defining useful features for this selection is not trivial and algorithm-dependent. In order to select an efficient algorithm for a specific input scene, we would need a feature vector of a whole configuration of the rendering process. This information includes parameters of each rendering algorithm that affects the performance, which in turn makes the whole framework algorithm-dependent. It

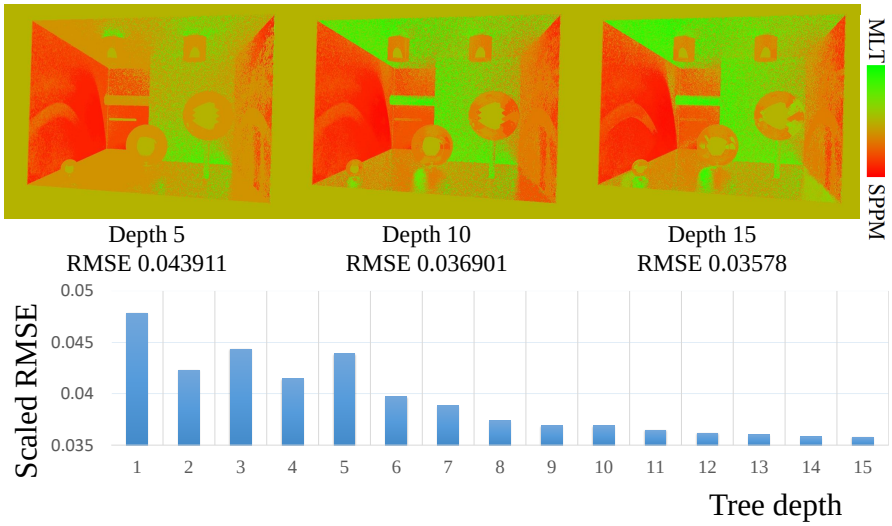

Fig. 6 Visualization of approximated optimal weights for the *box* scene with different tree depth (top) and the corresponding plot of the approximation errors (bottom). The RMS errors of the blended images are shown under each image. As the depth of the tree increases, the color indicating the preference to MLT becomes a bit more explicit, but not significantly after a certain depth. The plot of the variance shows how approximation errors of the optimal weights change according to the tree depth, which also stops converging around the depth of 15.

is also not obvious how to encode input scenes as feature vectors. Unlike images, which contain a set of pixels in a structured manner, scene data contains a set of very different information such as material data, textures, and triangle meshes. There is no single data structure common to all of data necessary to define input scenes. This lack of a common structured input form is a striking differences to applications of machine learning for images.

One might also consider finding a distribution of total rendering time, such that we do not spend too much computation for an algorithm with small weights. This deceivingly obvious improvement, however, is not possible since our regression forest is trained under the assumption that each algorithm spends the same rendering time. Even if we can find such a distribution of rendering time somehow, optimal blending weights are now different from those at the training phase since rendering time for each algorithm is also different. To implement this idea, we would need to have multiple regression forests for all the possible distributions of total rendering time, which is likely infeasible.

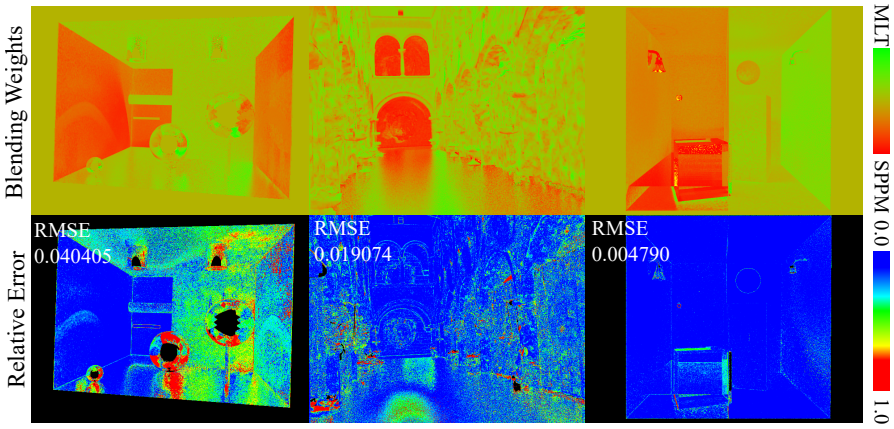

Fig. 7 Approximated blending weights (top) and the relative errors (bottom) for the selected three scenes (*box*, *crytek-sponza*, and *water*) by replacing regression forests via a neural network.

6.2 Comparison to Neural Networks

We used regression forests as a machine learning technique to learn the relationship between path features and the optimal blending weights. One possible option is to replace it by neural networks. Given its success in the computer vision community, a deep neural network [HOT06] is a possible candidate. We tested replacing regression forests by a fully-connected four layer's neural network using Caffe [JSD*14] on GPU as additional experiments. As shown in Fig. 7, we found that a neural network can achieve similar performance to regression forests. We discarded this approach in the end since even its running time is multiple orders of magnitudes slower (three minutes) than regression forests (60 msec) without much improvement in terms of RMS errors.

6.3 Limitations

Preparing Training Scenes: In general, a machine learning technique needs a large number of training samples to avoid overfitting. While we carefully designed a set of training scenes, it is not guaranteed that the prepared training scenes are indeed sufficient for learning. This situation is in contrast to the computer vision community; there are several standardized large datasets such as ImageNet [RDS*15]. Although we used some standard models and scenes often seen in other rendering research, it would be interesting as future work to generate training scenes based on procedural modeling. This procedural modeling should include not only shapes, but also materials, lighting, and camera parameters.

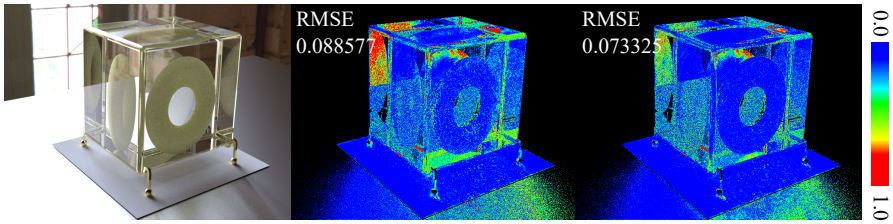

Fig. 8 RMS errors for a test scene that is only slightly different from a training scene. This test scene is made by changing the environment light and the camera configuration while retaining the geometry and materials of the *torus* scene in Figure 2. For this experiment, we used the original *torus* scene for training, and the modified *torus* scene at runtime.

Dependency on Training Scenes: We found that our method works especially well if there are only slight differences between training scenes and test scenes. Fig. 8 shows the *torus2* scene which uses the same geometry and materials as the *torus* scene in Figure 2, but with a slightly different camera configuration and an environment map. For this experiment, we used only the *torus* scene for the training phase, and rendered the *torus2* scene. We can observe that reduction of RMS error is significant in this case. This experiment indicates an interesting use case of our framework in practice: when an artist is modeling a new scene based on existing ones, we can train a regression forest with existing scenes beforehand.

7 Related Work

Light Transport Simulation in Rendering: Since the development of path tracing [Kaj86], the number of light transport simulation algorithms have been developed. Among many rendering algorithms, we used the two representative approaches in our tests: SPPM [HJ09] and MLT [VG97] with manifold exploration [JM12]. We chose these two approaches because their algorithms are completely different and have different characteristics as rendering algorithms. SPPM works by tracing a number of light paths and estimates density of light path vertices at a visible point through each pixel. MLT on the other hand traces a whole path by a Markov chain from the previously generated path and estimates the histogram of this Markov chain at all the pixels. SPPM is generally considered good at rendering caustics, while MLT is considered efficient at resolving complex visibilities from light sources. Our framework however is not restricted to use very different algorithms, since it is independent of how each algorithm works internally.

Machine Learning in Rendering: Several researchers have already applied machine learning to rendering. One popular application of machine learning in rendering is regression models. Among others, Jacob et al. [JRJ11] utilized unsupervised online-learning of a Gaussian mixture model (GMM) to represent a radiance distribution in participating media. Vorba et al. [VKŠ*14] also used online learning of

GMM to represent probability density functions for importance sampling. Ren et al. [RWG*13] introduced a realtime rendering algorithm using non-linear regression to represent precomputed radiance data. The precomputed radiance data is modeled as a multi-layered neural network [Hay98]. The idea is to learn the relationship between scene configurations and the resulting radiance distribution based on off-line rendering with random attributes. While we also use machine learning for regression, we propose to use machine learning to combine existing rendering algorithms without any modification to them. Our framework thus can be applied on top of any of the previous work mentioned above. More recently, Nalbach et al. [NAM*17] showed how to use CNN to approximate screen-space shaders. While the goal of their work is completely different from ours, their work demonstrate the powerful potential of applying machine learning to rendering.

Kalantari et al. [KBS15] recently proposed a image filtering technique to reduce Monte Carlo rendering noise based on the multilayer perception [Hay98]. The idea is to learn the relationship between the scene features such as a shading location or texture values and a set of filtering parameters. Our work is inspired by their successful application and we also use machine learning to find the relationship between path features and the optimal blending weights. The difference is that their work focuses to improve the result of a single image by filtering, while we consider a situation where there are multiple rendering algorithms available for a user.

The aim of our work is to use machine learning to blend the results of different rendering algorithms. Such blending is often done by multiple importance sampling [VG95], and there have been many recent works on this approach [HPJ12; GKD*12]. Our work differs from multiple importance sampling in that we treat each rendering algorithm as a black-box and does not require any detailed algorithmic information such as path probability densities.

Regression Forests: Regression forests [Bre01] are actively used in many applications. One famous example is Kinect body segmentation [SFC*11]. By simply fetching neighboring depth values and parse the regression forest, this algorithm can label each pixel by 31 different body parts quite accurately in realtime. For face recognition, Ren et al. [RCWS14] showed that regression forests can be used to detect major features such as eyes, a mouth, and a nose. Tang et al. [TYK13] used regression forests to extract a skeletal hand model from an RGB-depth image.

For applications in computer graphics, Ladický et al. [LJS*15] used regression forest for fluid simulation and achieved x200 speed up. They trained a regression forest via position-based fluid simulation by defining several features around each particle. The trained regression forest is used to update the state of particles at the next time step, without relying on costly simulation. Inspired by the success of regression forests in many applications, we also utilize regression forests instead of a more popular convolution neural network [HOT06]. As far as we know, our work is the first application of regression forests in rendering.

8 Conclusion

We presented a framework to automatically blend results of different light transport simulation algorithms. The key idea is to learn the relationship between a class of light transport paths and the performance of each algorithm on each class. For classification of paths, we introduced a feature vector based on relative contributions from different types of paths according to Heckbert's notation. We then calculate optimal blending weights such that a resulting image has minimal errors on average after blending. Using regression forests, we approximate a function that takes a feature vector of light transport paths and outputs the optimal blending weight per pixel. The resulting framework is independent from how each algorithm works, which makes it easily applicable to different rendering algorithms.

References

KD13. KAPLANYAN A. S., DACHSBACHER C.: Path Space Regularization for Holistic and Robust Light Transport. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering) 32*, 2 (2013), 63–72.

MB90. MACDONALD J. D., BOOTH K. S.: Heuristics for ray tracing using space subdivision. *The Visual Computer 6*, 3 (1990), 153–166.

GSK*10. GRITZ L., STEIN C., KULLA C., CONTY A.: Open Shading Language. *ACM SIGGRAPH 2010 Talks* (2010), Article 33.

Bre01. BREIMAN L.: Random forests. *Machine Learning 45*, 1 (2001), 5–32.

GKD*12. GEORGIEV I., KRIVANEK J., DAVIDOVIC T., SLUSALLEK P.: Light transport simulation with vertex connection and merging. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 31*, 6 (2012), Article 192.

Hav00. HAVRAN V.: *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.

Hay98. HAYKIN S.: *Neural Networks: A Comprehensive Foundation*, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1998.

Hec90. HECKBERT P. S.: Adaptive radiosity textures for bidirectional ray tracing. *Computer Graphics (Proc. SIGGRAPH) 24*, 4 (1990), 145–154.

HJ09. HACHISUKA T., JENSEN H. W.: Stochastic progressive photon mapping. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 28*, 5 (2009), Article 141.

HOJ08. HACHISUKA T., OGAKI S., JENSEN H. W.: Progressive photon mapping. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 27*, 5 (2008), Article 130.

HOT06. HINTON G. E., OSINDERO S., TEH Y.-W.: A fast learning algorithm for deep belief nets. *Neural computation 18*, 7 (2006), 1527–1554.

HPJ12. HACHISUKA T., PANTALEONI J., JENSEN H. W.: A path space extension for robust light transport simulation. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 31*, 6 (2012), Article 191.

Jak10. JAKOB W.: Mitsuba renderer, 2010. http://www.mitsuba-renderer.org.

Jen96. JENSEN H. W.: Global illumination using photon maps. *Proc. Eurographics Symposium on Rendering* (1996), 21–30.

JM12. JAKOB W., MARSCHNER S.: Manifold exploration: A Markov chain Monte Carlo technique for rendering scenes with difficult specular transport. *ACM Transactions on Graphics (Proc. SIGGRAPH) 31*, 4 (2012).

JRJ11. JAKOB W., REGG C., JAROSZ W.: Progressive expectation–maximization for hierarchical volumetric photon mapping. *Computer Graphics Forum (Proc. Eurographics Symposium on Rendering) 30*, 4 (2011).

JSD*14. JIA Y., SHELHAMER E., DONAHUE J., KARAYEV S., LONG J., GIRSHICK R. B., GUADARRAMA S., DARRELL T.: Caffe: Convolutional architecture for fast feature embedding. *CoRR abs/1408.5093* (2014). URL: http://arxiv.org/abs/1408.5093.

Kaj86. KAJIYA J. T.: The rendering equation. *Computer Graphics (Proc. SIGGRAPH)* (1986), 143–150.

KBS15. KALANTARI N. K., BAKO S., SEN P.: A machine learning approach for filtering Monte Carlo noise. *ACM Transactions on Graphics (TOG) (Proceedings of SIGGRAPH 2015) 34*, 4 (2015).

LJS*15. LADICKÝ L., JEONG S., SOLENTHALER B., POLLEFEYS M., GROSS M.: Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (Proc. SIGGRAPH Asia) 34*, 6 (2015), 199.

LW93. LAFORTUNE E., WILLEMS Y. D.: Bi-directional path-tracing. *Proc. Compugraphics* (1993), 145–153.

NAM*17. NALBACH O., ARABADZHIYSKA E., MEHTA D., SEIDEL H.-P., RITSCHEL T.: Deep shading: Convolutional neural networks for screen-space shading. *Computer Graphics Forum (Proc. EGSR 2017), 36* (2017), 4.

RCWS14. REN S., CAO X., WEI Y., SUN J.: Face alignment at 3000 fps by regressing local binary features. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2014).

RDS*15. RUSSAKOVSKY O., DENG J., SU H., KRAUSE J., SATHEESH S., MA S., HUANG Z., KARPATHY A., KHOSLA A., BERNSTEIN M., BERG A. C., FEI-FEI L.: ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV) 115*, 3 (2015), 211–252. doi:10.1007/s11263-015-0816-y.

RWG*13. REN P., WANG J., GONG M., LIN S., TONG X., GUO B.: Global illumination with radiance regression functions. *ACM Transactions on Graphics (Proc. SIGGRAPH) 32*, 4 (2013), 130:1–130:12.

SFC*11. SHOTTON J., FITZGIBBON A., COOK M., SHARP T., FINOCCHIO M., MOORE R., KIPMAN A., BLAKE A.: Real-time human pose recognition in parts from a single depth images. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2011).

TYK13. TANG D., YU T.-H., KIM T.-K.: Real-time articulated hand pose estimation using semi-supervised transductive regression forests. In *The IEEE International Conference on Computer Vision (ICCV)* (2013).

Vea98. VEACH E.: *Robust Monte Carlo methods for light transport simulation*. PhD thesis, Stanford University, USA, 1998. AAI9837162.

VG94. VEACH E., GUIBAS L. J.: Bidirectional estimator for light transport. *Proc. Eurographics Symposium on Rendering* (1994), 147–162.

VG95. VEACH E., GUIBAS L. J.: Optimally combining sampling techniques for Monte Carlo rendering. *Proc. SIGGRAPH '95* (1995), 419–428.

VG97. VEACH E., GUIBAS L. J.: Metropolis light transport. *Proc. of SIGGRAPH 97* (1997), 65–76.

VKŠ*14. VORBA J., KARLÍK O., ŠIK M., RITSCHEL T., KŘIVÁNEK J.: On-line learning of parametric mixture models for light transport simulation. *ACM Transactions on Graphics (Proc. SIGGRAPH) 33*, 4 (aug 2014).