Imperial College London

MEng Individual Project

Imperial College London

Department of Computing

One Shot Radiance
Convolutional Autoencoders Deeper in Ray Tracing

Author:
Giulio Jiang

Supervisor:
Dr. Bernhard Kainz

Second Marker:
Dr. Abhijeet Ghosh

June 17, 2018

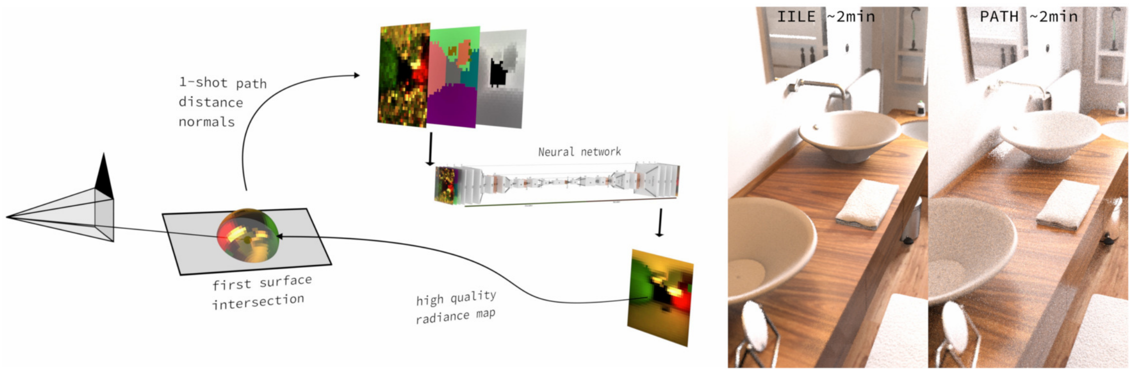


Figure 1: One-Shot Radiance (OSR) evaluates a low quality radiance map (one-shot), distance and normals at the first intersection, and uses a neural network to obtain a higher quality radiance map containing all indirect lighting. The *Bathroom* scene from PBRTv3's scenes [20] shows OSR compared to a same-time path traced image. OSR produces a noise-free result with convincing global illumination after 30 seconds, and progressively refines quality afterwards.

Abstract

Rendering realistic images with Global Illumination (GI) is a computationally demanding task. A recent project used Generative Adversarial Networks (GAN) to predict indirect lighting on an image level, but is limited to diffuse materials and requires training on each scene.

We present One-Shot Radiance (OSR), a novel machine learning technique for rendering Global Illumination using Convolutional Auto-encoders. We predict radiance maps at the first path tracing intersection level using the Neural Network, which uses as inputs a One-shot radiance map (1 sample/pixel intensity), distance and normal maps. Indirect illumination values are interpolated to offer high performance GI rendering while supporting a wide range of material types, without any offline precomputation on the scene. The independent direct lighting layer preserves details and shadow correctness.

OSR has been evaluated on scenes of interiors, and is able to produce high-quality images within 180 seconds.

Acknowledgements

I would like to thank

- My supervisor Dr. Bernhard Kainz for the feedback and suggestions he has provided me during the project
- My parents for the unfaltering support they have given me during these university years
- All my friends, who have shared with me every memorable moment of the degree

Contents

1 Introduction

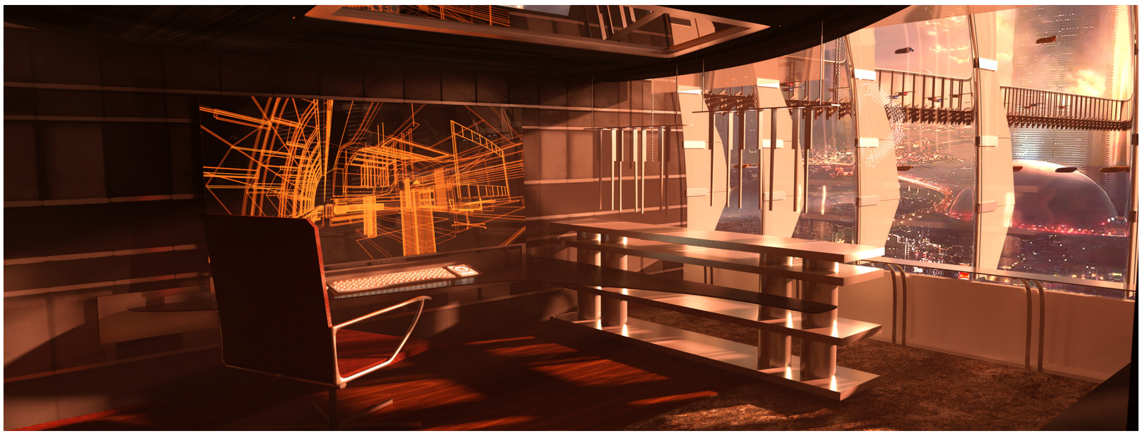


Figure 2: The *Futuristic Tower* scene took 50 CPU-days to render using the Metropolis Light Transport Bi-directional Path Tracer implemented by LuxRender. A more efficient rendering algorithm could allow a much faster workflow on this scene, or be cheap enough to allow creating an animation.

Ray Tracing is capable of producing photorealistic images virtually indistinguishable from real pictures. Progressive refinements on rendering algorithms, such as Bi-Directional Path Tracing [45] and Metropolis Light Transport [67] have increased the efficiency of rendering engines in scenarios in which light paths are difficult due to the high amount of indirect lighting and GI. Complex lighting conditions are however still highly expensive to resolve, and most algorithms require long rendering times to clear out the noise from Monte Carlo sampling.

Recent research has attempted to use machine learning techniques to accelerate rendering of GI effects. The *Deep Illumination* [63] approach uses a GAN to translate diffuse albedos, normals and depth maps to a global illumination component layer, and obtained good success at predicting indirect illumination in real time for diffuse materials. The network requires specific training for each scene to be rendered, but is able to extrapolate and adapt to dynamic objects and new shapes introduced.

To overcome the limitations of diffuse-only materials, and to make the renderer easily usable without the need of offline training, we attempt a more general approach to resolving GI based on the estimation and caching of radiance maps for indirect illumination only. Indirect lighting is the primary cause of strong noise in Monte Carlo rendering, and is much more difficult to clear than first bounce lighting due to the high dimensionality of the paths. Rendering can therefore be accelerated by predicting approximate but noise-free radiance maps that can be interpolated and used to obtain the indirect illumination component of the final image. Thanks to the slowly changing nature of indirect illumination, artifacts and bias are not very visible in the general case, while the overall predicted GI looks convincing and noise-free. With OSR we propose to work not on the final rendering image level, but to use Path tracing to find the first intersection in the scene, where a Convolutional Autoencoder predicts a high quality radiance map from a path traced map rendered at just 1 sample per pixel (one-shot map), a depth and a normals map. Using the high quality radiance map we can compute all indirect illumination contributed from path tracing bounces beyond the first one. We use a neural network to predict high quality radiance maps to accelerate rendering of GI effects. Our approach works with a wide range of material types, and does not require any offline pre-computation or per-scene training.

2 Background

In the background section we present an overview of graphics and ray tracing in Section 2.1, Machine Learning in Section 2.3, and some recent related work in Section 2.5.

2.1 Ray tracing

In computer graphics, there are two major techniques that can be used to generate high quality images from a geometric model of the scene and of the lights: surface mesh rasterization rendering pipelines and ray tracing.

The rasterization based rendering pipeline is commonly implemented in all modern graphics cards and their APIs, and widely used by real-time 3D visualization systems to provide a low-latency output to the user in the form of a constant stream of frames. Graphics pipelines such as the one described by OpenGL transform a 3D scene into a perspective projection through geometric manipulation of all the objects, and can be programmed by writing shaders to provide illumination properties and other effects that help achieve visual correctness. The pipelines are built and programmed with efficiency in mind: transformations expressed in matrix form can be applied to arbitrary numbers of scene elements in parallel using hardware acceleration, and processing time for many of the rendering stages scale linearly with the number of geometric primitives present in the viewing frustum and with the target display resolution. The fixed layout and structure of such rendering pipelines however limit the expressiveness and flexibility of the graphical effects they can output. While shading languages offer full programmability to developers, the major benefits from hardware acceleration come from the fixed functionalities in the pipeline, such as the rasterization stage which processes geometry triangles into screen-space fragments.

Ray tracing is a different approach to the task. Rendering is not performed by transforming geometry from world space to screen space and applying lighting effects, but by simulating light rays according to physical laws. For example, a lamp in a room shoots light rays that bounce on walls until they reach the camera film's individual pixels.

Ray tracing simulations perform numerical integration on the illumination contribution of every object present in the scene. As long as the behaviour of light rays is simulated in an accurate and correct way according to surface and volume properties, ray tracing promises to be able to produce images that are scientifically realistic and virtually indistinguishable from real photographs[1] .

While it is intuitive to think about simulating rays starting from the light sources, the most common ray tracing algorithms simulate rays in the opposite direction, starting from the camera. Helmholtz Reciprocity [8] states that incoming and outgoing light rays are perfect reversals of each other, and it is perfectly valid to simulate light behaviour from the opposite direction. Practical experiments such as *Dual Photography* [60] have shown that it is possible to exchange positions of light source and camera, and to produce photographs from the viewpoint of the projector as if it were the camera.

Many ray tracing algorithms exploit Helmholtz reciprocity by shooting light rays starting from the camera instead of the light sources. The common pinpoint camera model employs a point

[1]In order to obtain a fully physically accurate render, a few more requirements need to be met. Most rendering engines use the three standard RGB channels to create a colored picture. While RGB is usually sufficient to accurately model the visible part of the light spectrum, it doesn't fully simulate all light path properties that occur in nature. For this reason some physically accurate ray tracers perform all simulations in the full spectrum of light frequencies. Such renderers are called spectral renderers [21], and can for example create extremely realistic images of dispersive refraction. Additionally to spectral rendering, light physics also include quantum effects such as light diffraction, phosphorescence and fluorescence, but they are typically disregarded as they would only increase the complexity of the simulations without improving the quality of the output or offering effects that could be otherwise achieved in different ways.

aperture that has no physical dimension. Using forward tracing of rays towards a camera sensor would result in an infinitesimally small probability of hitting the pinpoint camera's aperture, while using reverse simulations it is possible to start rays from the camera and find light sources that have a physical dimension.

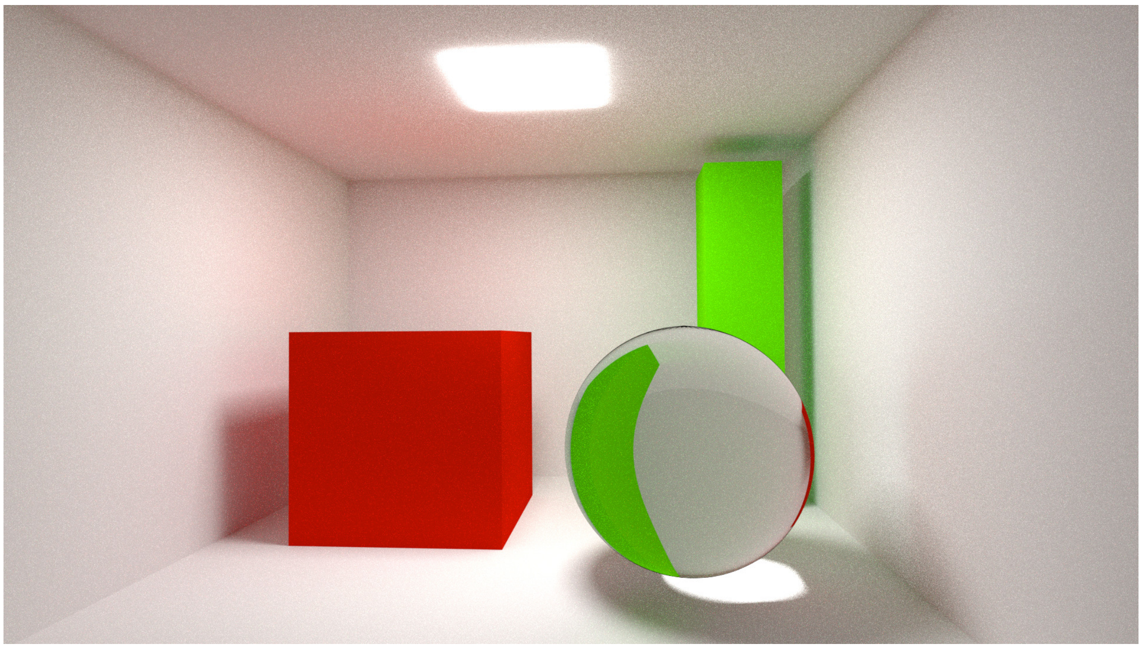


Figure 3: A simple scene rendered with Path Tracing. The effects of global illumination are clearly visible on the walls and in the shadows.

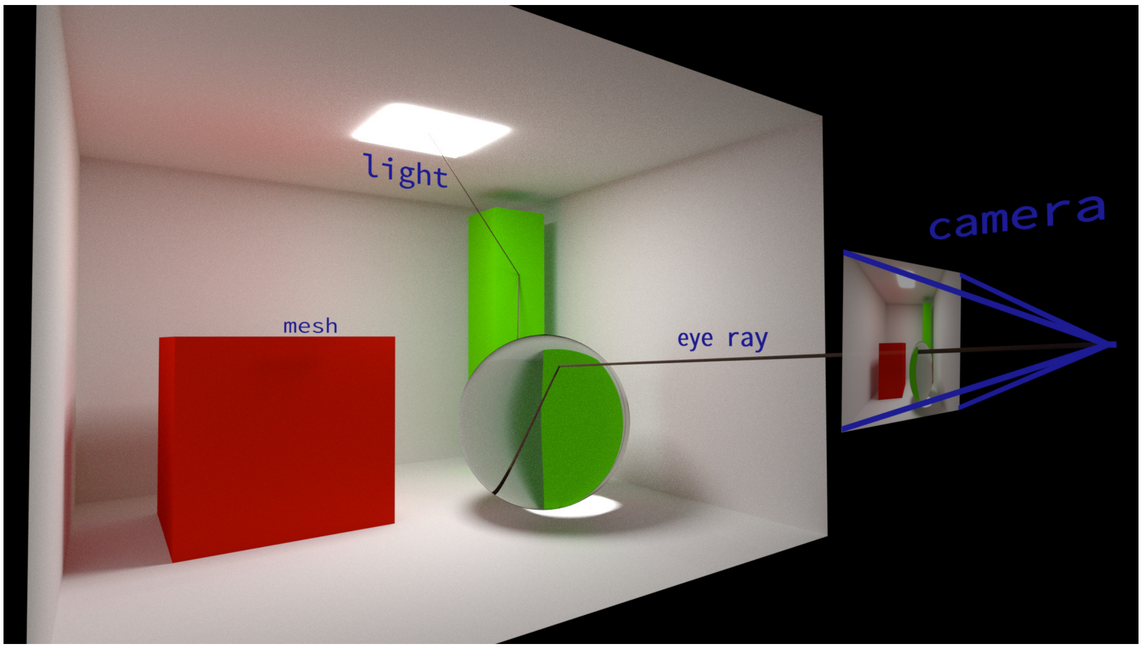


Figure 4: Visualization of the different elements of a 3D scene: camera, mesh, and light source. A hypothetical eye ray has been traced through the environment.

We now present the basic components of a 3D scene.

2.1.1 Camera

The most common camera model used in computer graphics is the *pinhole* model. The aperture the camera is a single point, and the camera film is placed in front of the focal point. This simple model is easy to implement in software, and represents a perfect real world camera with infinitely small aperture. In Figure 4 the camera is positioned in front of the scene, and the focal point is the origin of all the eye rays that are shot into the scene. The pixel that will be updated is the intersection point between the eye ray and the camera film plane.

The pinhole model can be extended to support more realistic camera features. Depth of Field effects can be modelled by introducing a non-zero *aperture*: the origin point of the rays has some surface area, and the film plane is positioned at the distance of the focal plane.

Other common kinds of cameras are the *Orthogonal projection camera*, and the *Fish-eye* camera.

The *Environment* camera can be used to obtain a 360 degree view of the scene. It is similar to the pinpoint camera, with a single point focal point. The film, however, is not a plane, but a sphere centered around the focal plane. Any object that is at the focal plane therefore is in focus, and one that is behind or in front of it becomes blurrier the farther away it is.

2.1.2 Mesh

Often referred to as `Objects` or `Geometry`, the mesh of a scene is the visible and concrete part. Mesh is composed of vertices, edges and surfaces, and it is the main source of interaction for light paths.

The geometry of a scene can be defined and stored in different ways. The most common representation is based on vertices, edges and triangles.

2.1.3 Lights

In order to have any visible element, a scene requires at least one light source. While light sources are treated in a particular way, there isn't necessary a clear distinction between *light* objects and *mesh* objects, as *mesh* object materials can be also configured to emit light. With Global Illumination effects, all surfaces can be treated as light sources.

The lights of a scene generate all the light rays, and are the endpoints where camera-to-light ray tracing simulations terminate if they are not terminated early for efficiency reasons.

Several different kinds of lights exist:

- `Point light` - a source that has no physical dimension. While point lights are easy to model in a rasterization pipeline, they are not realistic in unbiased ray tracing as there is zero probability of finding a point light with rays starting from a camera. Point lights are however useful in modeling simplified illumination configurations, such as in the Instant Radiosity method [39].
- `Spot light` - similar to the Point light, a Spot light has no physical dimension but only emits light in a cone towards a specific direction.
- `Area light` - a source that emits light from a 2D surface. Area lights can be used to generalize other types of lights, and provide uniform emission across the surface. Area lights are capable of generating realistic illumination by casting soft shadows and smooth transitions between the illuminated parts of the scene. An emitting object can be modeled in a raytracing software as a collection of area lights corresponding to the faces of the object.

- `Environment light` - a source that has no precise position coordinate in the 3D environment, mapped at *infinity* distance in the world. Environment lights are used for sun and sky lighting, and for Image Based Lighting [31] with HDRI and Environment maps. Sampling of environment lights has been subject of several research efforts, and is highly optimized in modern rendering engines.

2.1.4 The visibility problem

One of the primary goals of any rendering technique is to obtain an appropriate 2D projection of a 3D scene or model. This is the `visibility problem`.

In order to project a scene as an image, a rendering system is required to obtain a viewpoint of the objects depending on the relative position of the camera, and to calculate which objects are visible and which are occluded.

The visibility problem has several different solutions:

- `Rasterization and Z buffering` are implemented in many rendering pipelines. Each triangle in the scene is rasterized by calculating the pixels needed to draw it, generating fragments. Fragments contain a Z value representing a distance from the viewing point that can be used to determine which fragments are visible and how to blend them.
- `Ray tracing` is not only a rendering technique but also a solution to the visibility problem itself. The visibility problem is solved by computing the intersection between an eye ray and every surface in the scene, and taking the closest intersection, for each pixel.

2.1.5 Local, Direct and Global illumination

Local illumination models are the simplest ones because they work on each object or primitive independently, and only consider the relationship between surfaces and light sources. Local illumination models, such as the Phong Model [54], can compute view-dependant highlights and illumination. A very simple illumination solution for computer graphics can be based on three distinct components: *Diffuse*, *Specular* and *Ambient*. The *Diffuse* term contains the primary color of the surface material, and does not depend on the viewing direction. The *Specular* contains view-dependant highlights produced by glossy materials. The *Ambient* component is a rough approximation of the total amount of indirect light received by all other surfaces. Local illumination models are easy to implement in shader languages and can be computed at interactive rates, although they offer a limited amount of realism.

Direct illumination models will also consider all the other objects in the scene, and can generate shadows and some ambient occlusion. Shadows are computed by tracing additional rays, called `shadow rays`, between surface points and light sources to determine whether an object is illuminated or not. Direct illumination is a considerable step forward as it can produce accurate shadowing, including self-shadowing, while still being reasonably cheap to compute.

Global Illumination (GI) computes the lighting of each object considering not only the contribution of light sources, but also the indirect lighting of all other non-emitting surfaces in the scene. No object in reality is able to absorb all the light it receives, and it will therefore produce some reflected illumination. While this statement is obviously true for glossy materials, even matte materials reflect a significant amount of light. The effect is visible in figure 3, where there are some red and green halos on the walls closer to the objects, caused by indirect lighting.

In the absence of Global Illumination, shadowed locations are completely black. This is unrealistic, and Local illumination models often use an *Ambient light* component to approximate the overall contribution of all indirect lighting, and *Ambient Occlusion* to approximate the darkness that is typical when objects are very close to each other, such as in corners.

Global Illumination is expensive to compute as it requires to integrate over all the visible surfaces, with an arbitrary number of light ray bounces. Brute force computation of Global Illumination is impossible due to the infinite integration domain in both the spatial and ray depth domains. Monte Carlo integration rendering methods are commonly used to obtain physically correct rendering of Global Illumination.

2.1.6 Photometric Quantities and Light Transport Equation

Photometric quantities provide measures of *brightness* and *power* that are weighted with respect to how humans perceive light. Symbols referring to photometric quantities are typically distinguished from radiometric quantities by a subscript *v*.

Luminous Flux is the amount of luminous energy that leaves a light source, and it is measured in Watts (W).

$$\Phi_v$$

The amount of luminous energy that leaves a source in a particular direction is the Luminous Intensity. It is the Flux per unit of solid angle, and it is measured in $\frac{W}{sr}$

$$I_v = \frac{d\Phi_v}{d\omega}$$

The Irradiance is the density of incoming flux on an area. It is the ratio between incoming flux and surface area, and it is measured in $\frac{W}{m^2}$

$$E = \frac{d\Phi_v}{dA}$$

Radiance is the intensity, emitted in a certain direction, orthogonally projected on a surface, where s is the surface area

$$L_v = \frac{dI_v}{ds \cdot cos\theta}$$

The Rendering Equation [35] expresses the fundamental behaviour of luminous energy and is at the core of the mathematical problem that rendering engines attempt to solve or approximate

$$L_o(x, \vec{\omega}_0) = L_e(x, \vec{\omega}_0) + \int_{\Omega_{4\pi}} K_L(x, \vec{\omega}_0, \vec{\omega}_i) L_i(x, \vec{\omega}_i) d\omega_i$$

In the *local* form, the Light Transport Equation is expressed in terms of a specific location x . Output radiance L_o of a location in a direction is the sum of its directly emitted light L_e plus the integral over the sphere of the scattering of the light that it receives. K_L is the kernel and describes how light is scattered. L_i is the light received from a different direction $\vec{\omega}_i$.

The reflectance function of a surface is

$$K_L(x, \vec{\omega}_0, \vec{\omega}_i) = f_r(x, \vec{\omega}_0, \vec{\omega}_i) cos(\theta_i)$$

f_r is the BRDF (Bidirectional Reflectance Distribution Function) and θ_i is the angle between $\vec{\omega}_i$ and the surface normal. The BRDF encodes the surface material properties: the amount of energy that is reflected in a direction given an incoming light direction. The BRDF is defined over the hemisphere, while a BSDF (Bidirectional Scattering Distribution Function) is defined over the sphere and can also model volumetric scattering.

Only numerical methods can solve the integral in the Light Transport Equation, and an infinite number of samples on all scene surface points is required in order to obtain a correct answer. The task is extremely computationally expensive, and modern rendering techniques attempt to find an efficient numerical solution to the Equation, or a visually convincing approximation.

2.1.7 Bias

Given the theoretical ground truth given by fully evaluating the Light Transport Equation, a rendering algorithm is unbiased if and only if, given an infinite amount of computational time, the output does not have a systematic error. Therefore, as the number of samples tends to infinity, the expected difference from the ground truth should be zero.

2.1.8 Ray tracing algorithms

These are some of the most common ray tracing algorithms:

- `Direct lighting` is a simple kind of ray tracing. Its main purpose is to solve the visibility problem, and it does not attempt to compute global illumination. Eye rays are shot from the camera to the first intersection, from which a single shadow ray is used to determine whether the object is in shadow. Although the shadow ray is shot in a deterministic direction, Direct lighting can produce noise and requires a number of samples per pixel in order to correctly render reflective and refractive surfaces, as well as area lights and environment lights.
- `Path tracing` is usually considered the most basic form of unbiased ray tracing able to compute global illumination. In `path tracing` all rays are simulated starting from the camera, and solve the visibility problem by simply finding the closest intersection with a scene object along the ray's travel path. At each intersection with a surface, additional random secondary rays are generated according to the material properties, and their contribution is added together. Due to the exponential explosion of secondary rays with an increasing number of bounces, Monte Carlo methods are employed to sample a single secondary ray at each intersection. Path tracing is a powerful and unbiased general purpose algorithm, although it shows its performance weakness in scenarios where the majority of light is indirect, such as indoor scenes.
- `Photon mapping` [34] consists of two separate passes. In the first pass, photons are shot from the light sources into the scene, creating a cache of photons in the 3D space. In the second pass, path tracing is used to explore the photon map and compute the illumination values of the visible surfaces. Opaque surfaces are rendered in the Final Gather process, in which illumination is collected from the second bounce to avoid obtaining a patchy look. Caustics are rendered from an independent photon map that specifically targets refracted light. Photon mapping is biased due to the finite number of photons generated. Stochastic Progressive Photon Mapping [42] improves Photon Mapping by making the rendering algorithm unbiased and reducing its memory usage.
- `Bi-directional path tracing` is an extension of path tracing. Rays are traced at the same time from the camera and from the light sources, and each of the intersection points obtained from the two directions are joined together by an additional ray, which may or may not create a valid path. Importance sampling is used to combine and weight the contributions of each connecting path. Bi-directional path tracing is more effective than standard path tracing when there is a large amount of indirect lighting thanks to its ability to obtain valid paths even from hidden light sources. [45]

2.2 Ray tracing inefficiency

One of the most important motivations that have pushed past research efforts into developing more advanced and performant renderers is the general inefficiency of ray tracing algorithms.

Ray tracing has seen wide adoption thanks to both its simplicity in its ease of implementation and usage [2] and for it being extremely powerful in simulating realistic light effects [14].

[2] A minimal ray-tracer can be written in less than 100 lines of code [27], or even fit on a business card [59]

Ray tracing performs numerical integration on the light that each point in the scene is emitting towards a given viewing direction. At each intersection point, the secondary rays generated will depend on the properties of the surface. For example, a simple matte surface can shoot secondary rays in completely random directions to sample a uniform space, and use the average amount of gathered ray to calculate its diffuse color and intensity, while a perfect mirror generates secondary rays in exactly one direction given the viewing direction.

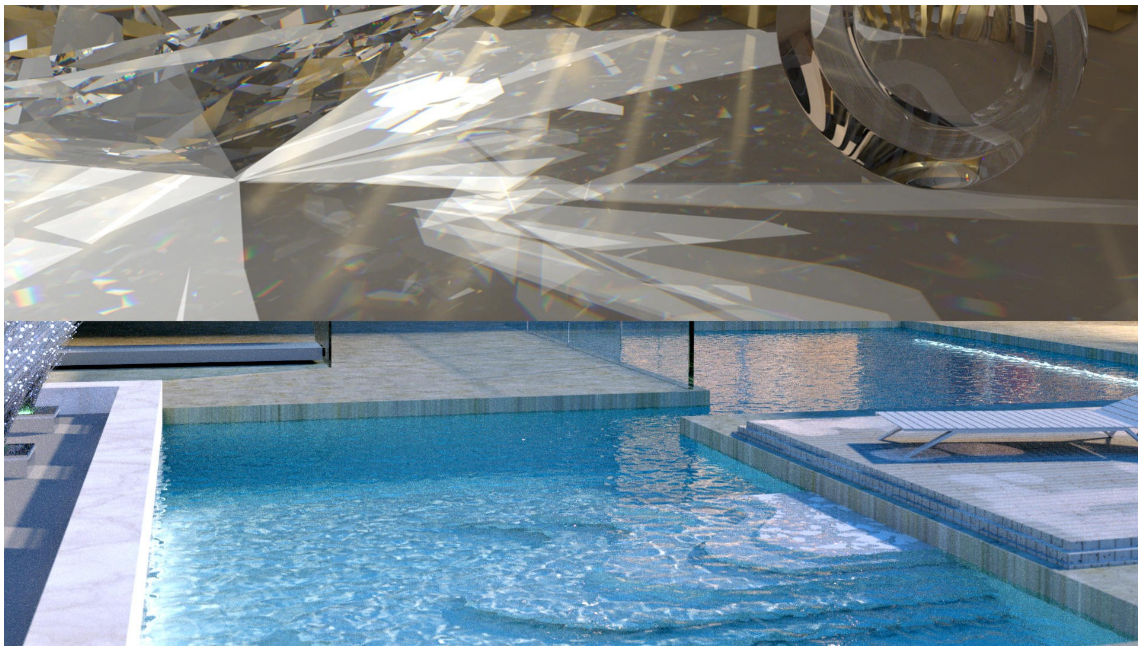


Figure 5: Caustics typically take a very long time to converge. Top image: physically correct color dispersion requires an accurate model of the material properties and a large number of samples to converge to the correct color. Bottom image: lighting effects at the bottom of the pool require solving Diffuse-Specular-Diffuse paths that are challenging for Path Tracing and Bi-Directional Path Tracing.

With GI, all light contributions are taken into consideration, not only those that come from direct interaction with the light source. Effects such as indirect lighting and color bleeding are possible.

An immediate consequence of GI is that rays needs to interact with every object in the scene and not only with the light sources. It is generally not possible to make assumptions about the position of the lights anymore. A simple but correct implementation of a diffuse surface integrator in a GI context would be to shoot secondary rays in random directions, in order to consider all contributions from any part of the scene.

Such an approach leads to difficulties in obtaining quick convergence towards the ground truth of the scene, the main reason being the small probability, in the average case, that a randomly generated ray from any surface in the scene will hit the light source. As computers do not have infinite computational power, a large number of pixel samples will likely bounce among different surfaces in the scene without hitting any light source and not contribute to the final pixel level in an effective way.

The small probability of discovering light sources in a scene where rays are shot randomly leads to pronounced noise in the rendered scene, and consequently to the necessity of using a large number of samples in order to obtain a clear image. An especially undesirable category of noise is typically referred as `fireflies` [10]. A firefly, such as the one in figure 6 is a type of noisy pixel

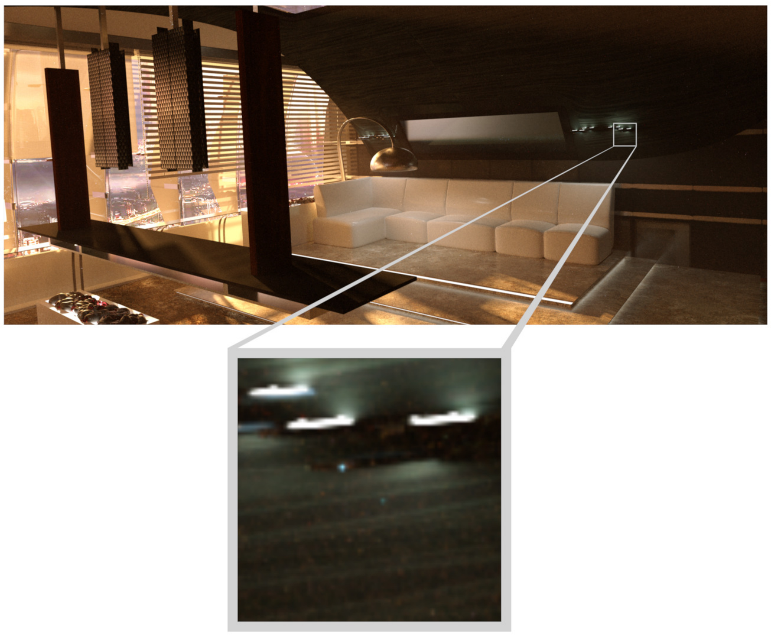


Figure 6: Fireflies show themselves as very bright pixels and can be present even after a large number of samples per pixel have been rendered. In this picture, the overall noise level is very low, but the bright blue fireflies still persist.

that distinguishes itself for being sparsely distributed in the final image and for being extremely bright, typically clipping at the limit of an RGB pixel. Fireflies are caused by unlikely ray paths that have hit a hard to find light path with high contribution. At a low number of samples, they are very clearly visible as most of the pixels will not have sampled the same kind of contribution, and it would take a long time to average the sample out to its correct value.

2.3 Machine Learning

Machine Learning studies how to make computers learn in a way that is more similar to how humans and animals do it. Cristóbal Esteban [25] explains that until Machine Learning was introduced, humans would teach computers how to perform certain tasks using programming languages. Mathematical problems can be specified in software, so that computers could perform the operations far faster than any human would be able to. The same way of teaching operations and activities also works among people, when we teach each other how to do specific actions, recognize patterns and so on. While descriptive languages based on *ifs* and rules are effective to precisely describe many scenarios, they are not the only way we can learn. Certain abilities are gained not through explicit teaching, but by gathering examples and experience over time. An example would be the capability of recognizing subjects from images, an activity that humans and other animals excel at doing, but that is very hard to describe in either words or programming languages to someone else. Machine Learning introduced a new way of teaching knowledge domains that are difficult to express using conventional languages, allowing computers to learn patterns and tasks in a way that is far more natural for humans.

Supervised Learning is based on showing the machine examples together with the correct label, allowing a model to extract features associated with labels to be able to classify unseen examples during operation. Unsupervised Learning does not provide labels, and lets the machine autonomously extract features from the examples. Reinforcement Learning trains the system towards optimal decisions by using a rewards system.

Machine Learning systems can solve different kinds of problems:

- *Clustering* - input data is grouped based on similarities and features learned autonomously by unsupervised learning systems
- *Classification* - after supervised training, the model predicts the class to which a new example belongs, for example to determine the species of an animal in a picture
- *Regression* - the system learns to compute new values based on some input data

2.3.1 Regression and Neural Networks

We focus on Regression problems in the context of the project.

The simplest kind of regression is *Linear Regression* [11]. Linear Regression attempts to find a best-fit linear relationship between the input data and the expected output, in such as way as to minimize the error between the predicted output and the training expected values. Linear Regression therefore is based on the analysis of correlation between the input and output distributions. While Linear Regression is a simple and effective model for many scenarios, it is highly susceptible to outliers in the datasets, and to multicollinearity, which occurs when multiple input predictors are correlated to each other, causing skew in the predictions. Linear Regression is not able to model complex relationships or any kind of non-linear models.

Non Linear Regression can be used to fit more complex functions to the input and output data, and can be useful to model some events.

Neural Networks have been an important focus of recent research. Neural Networks are inspired by the way neurons are organized in the brain in order to be able to recognize patterns in information and carry on computation. Typical neural networks are organized in layers. Each layer contains neurons that take inputs from previous layers and send information forwards to the next layer. Artificial neural networks use *Forward* propagation to obtain outputs, and *Backpropagation* to learn from examples.

Gradient Based Learning Methods [47] compute the error gradient from the expected output and the predicted one in a forward pass of the network, and backpropagates the derivatives of the error to previous layers to update the weights and achieve a smaller error at each step. *Gradient Descent* is the training process of progressively reducing the error rate in order to find a minimum, and it is the way a neural network learns from its training data.

2.3.2 Multilayer Perceptron

The *MultiLayer Perceptron* (MLP) is a simple and very common type of Neural Network. Individual neurons are organized in layers, and each layer is fully connected to the previous one: a neuron can read the output of each neuron in the previous layer. Figure 7 shows the visualization of an example MLP with a single hidden layer.

The network can be trained by changing the weight (and bias) assigned to each of the connections. For example, the output of the first neuron in the hidden layer would be:

$$h_1 = act(w_{11} * in_{11} + b_{11} + w_{12} * in_{12} + b_{12} + w_{13} * in_{13} + b_{13} + w_{14} * in_{14} + b_{14})$$

Where act is an activation function, w_i is a weight value, in_i is an input of a connection, b_i is a bias value. Activation functions, such as *Sigmoid function* or *Tanh* introduce non-linearity to the neural network, as without them the network would only be a linear combination of its inputs.

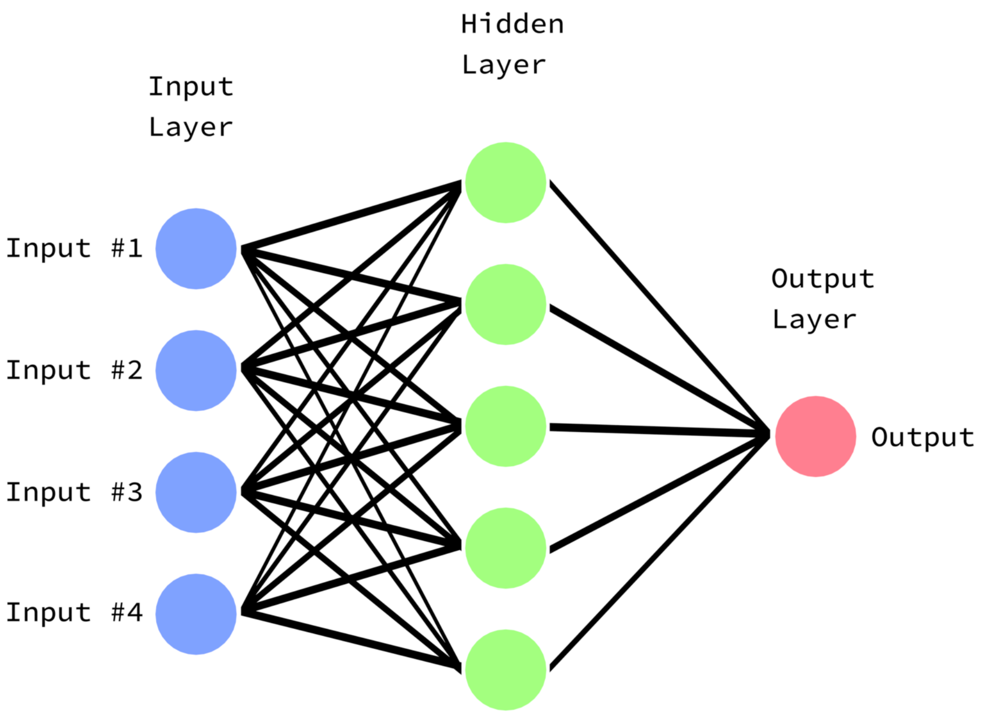


Figure 7: A Multilayer Perceptron has an input layer, an output layer, and one or more hidden layers. Each layer is fully connected to the previous one.

2.3.3 Activation Functions

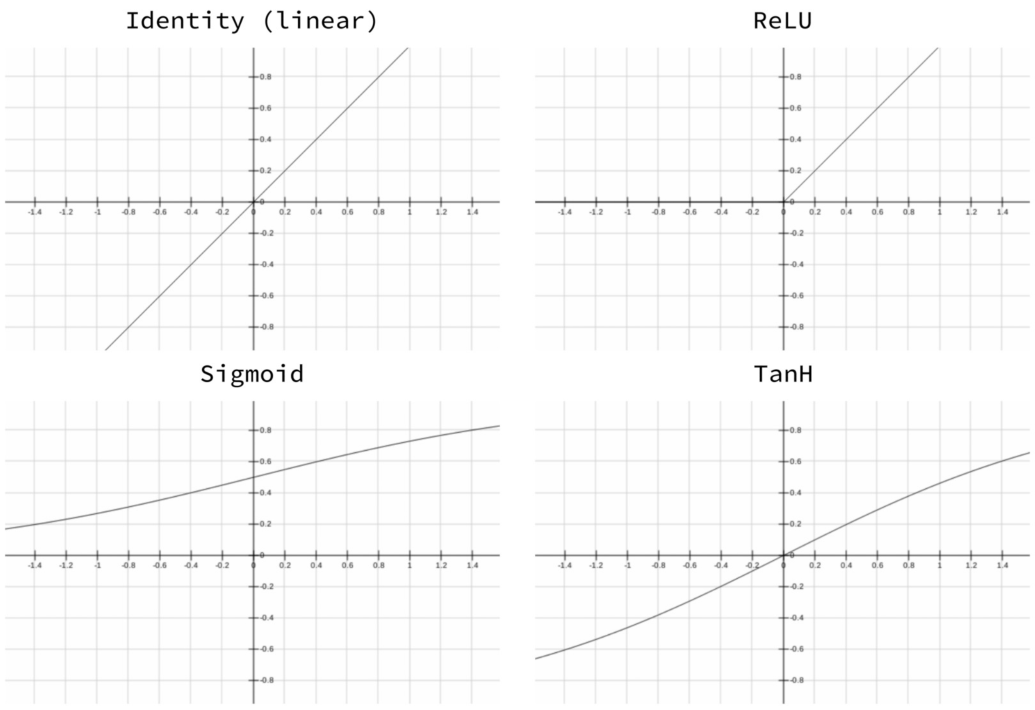


Figure 8: Plots of some activation functions

Activation functions are placed at the output of each layer, and have the purpose of introducing nonlinearity in the network behaviour. If a model has no activation functions, it acts as a linear combination of the input layer data no matter how many layers and neurons are present. Activation functions allow to model much more complex functions. Figure 8 shows plots of some of the most commonly used activation functions.

The *Identity* (Linear) function is the same as using no activation function at all, as it doesn't introduce any non-linearity to the system.

ReLU (Rectified Linear Unit) has a linear output if the input is positive, and zero otherwise. This is one of the most useful and popular activation functions, because it allows the network to react to relevant information. Compared to many other functions, it also offers performance advantages thanks to its simplicity.

Sigmoid is useful in classifier networks, as it boosts the relevance of values that are slightly larger or smaller than zero, allowing to discriminate more precisely between categories. The entire output space is positive.

TanH has a shape similar to the Sigmoid, but the output space is now ranging between -1 and +1.

There are no rules about what activation function should be used in any scenario, and experimentation and intuition are needed to select the most appropriate ones.

2.3.4 Loss Functions

The network needs to be able to estimate the error between its prediction and the ground truth in order to be able to determine the direction in which the gradients need to be pushed during learning. The Loss Function has the role of defining what the sign and magnitude of the error is, to allow backpropagation to update the parameters.

The *Mean Squared Error* (MSE), or L2 Loss, computes the sum of the squared errors between predicted z and expected y outputs. Intuitively, MSE computes an Euclidean distance between two vectors in a classification problem.

$$MSE := \frac{1}{n}\sum_{t=1}^{n}(y_t - z_t)^2$$

The *L1* Loss computes the sum of the absolute differences. L1 is less susceptible to outliers in the dataset, is simple to compute and understand, but it is not differentiable at the point 0.

$$L1 := \frac{1}{n}\sum_{t=1}^{n}|y_t - z_t|$$

The *Huber* Loss, or Smooth L1, uses a quadratic shape near zero values to make the L1 loss smoother and differentiable.

2.3.5 Convolutional Neural Networks

Convolutional Neural Networks (CNN) gained huge importance with the success it achieved in ImageNet classification [43].

The primitive component of a CNN is the *convolution*. The convolution is a mathematical filter containing parameters that are multiplied with the input in a dot product. With the input having more data points than the convolution does, the filter slides step by step to cover the entire input space. The purpose of a convolution is to recognize and highlight particular shapes and patterns,

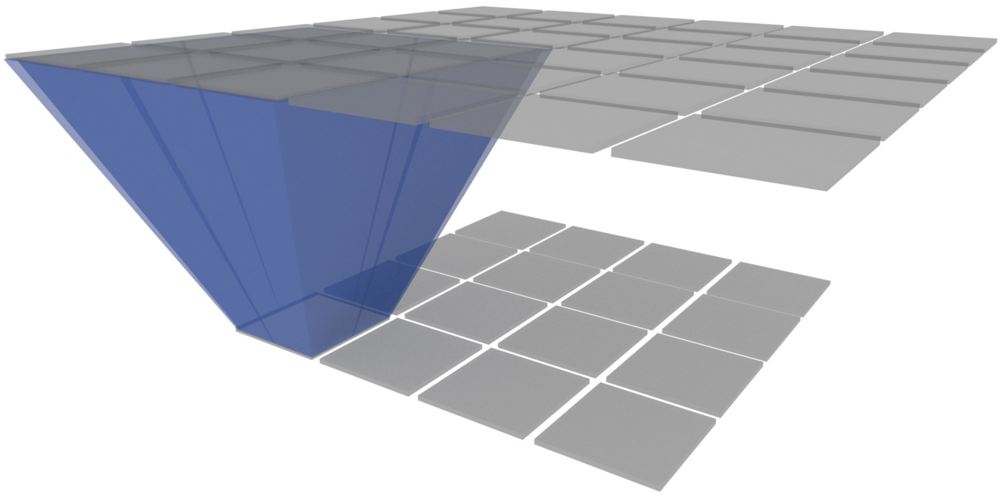


Figure 9: A single convolution, with a filter size of 3x3. As the filter slides across the input data, a 6x6 input is reduced to a 4x4.

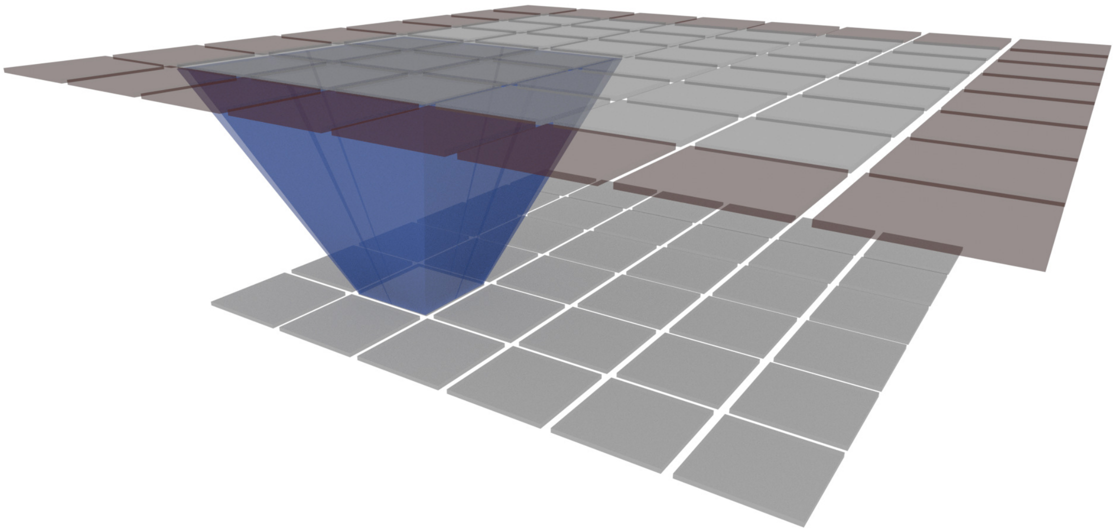


Figure 10: A single convolution, with a filter size of 3x3. Padding of 1 is added to each side of the input array. The output has the same dimension as the input, 6x6.

as a matching shape between the input and the convolution creates a larger output. Additionally, the convolution is space-independent when it slides across the input, and is able to recognize the feature no matter where it is located.

A convolution has three important parameters: filter size, padding and stride. Figure 9 shows a simple convolution step on 2D data arrays. The 3x3 convolution filter is applied to all possible locations in the input, and generates a smaller 2D output array. Therefore applying this kind of convolution would progressively reduce the dimensionality of the data, and not be suitable for very deep configurations. A common way to keep the same dimension after the convolution is to use *padding*: blank values are added to the sides of the input array to allow the filter to produce larger outputs. Figure 10 shows the same convolution as before, but the padding make it possible to obtain the same output dimension. The third important parameter is the stride step size. We assumed until now a stride of 1: the convolution filter is applied to every possible location in the input array without skipping any position. Increasing the stride makes the convolution skip some of the positions, generating smaller outputs. Some CNNs use striding instead of pooling to decrease the size of the data.

CNNs are inspired by the visual cortex, stimulated when exposed to particular shapes. A typical network is structured in a stack of layers, each performing a convolution. The dimensionality of the data can be reduced using pooling operations that summarize small patches with the maximum or average value, and the final compressed values are processed by a fully connected layer to output classification classes.

A study by [70] analyzes and explains how CNNs are capable of recognizing complex objects when structured in a deep stack. After training, convolutions in the first layer are maximally excited by simple geometric shapes, such as lines in various orientations and simple geometrical elements. Going deeper into the network, where pooling operations have compressed the data, the convolutions specialize into recognizing more and more complex items, progressively going towards higher levels of knowledge, until they become specialized for specific classes, such as certain animals or objects. The pooling and compression layers are essential to allow the network to compress low level features into higher level ones.

CNNs have been the subject of many more projects. The GoogLeNet [62], based on *Inception* modules and a total of 22 layers, introduces parallel processing to CNNs. Layers are not simply stacked one on top of each other, but the data flow can fork and go through different convolutions and pooling systems at the same time before being rejoined together and passed to the next module. The Microsoft ResNet [32] uses an even deeper network of 152 layers, also with non-linearity in the dataflow introduced by skip connections.

2.3.6 Autoencoders

An Autoencoder is a class of neural network that learns to output the original input data in presence of noise or missing values. Models can be trained using Unsupervised Learning, as the only data required for the loss function is the original input without added noise.

A Denoising Autoencoder by [68] is capable of reconstructing the original input given a noisy and partially occluded image from the MNIST dataset [15].

2.3.7 Overfitting and Dropout

Overfitting occurs when a model performs well on training data, but significantly worse or even randomly on unseen test data. This phenomenon can be caused by an excessive degree of freedom in the machine learning model. A network can learn to adapt well to the fixed set of examples it encountered during training, but perform badly on new data due to the excessive

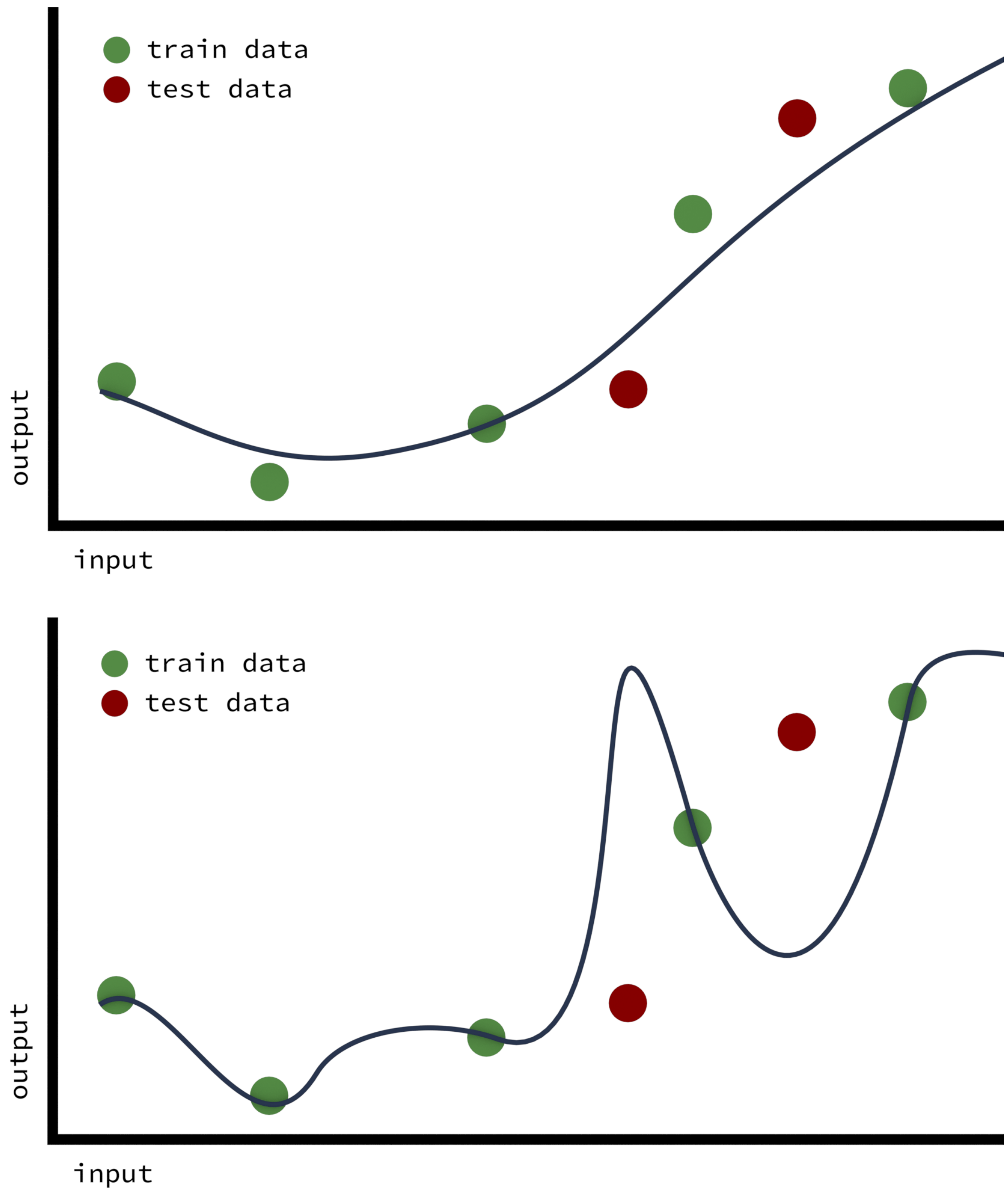


Figure 11: A visualization of overfitting. Top: a function is fit to data loosely, but is able to predict good output given new, unseen inputs. Bottom: the same data points are used to fit a much more precise function. However, it is not able to give good results on new data.

level of specialization. Figure 11 visualizes a machine learning model as a function of its input data. The model overfits by accurately matching its training data points.

Overfitting can be easily detected by monitoring loss values and other quality metrics during training on the test set, and can be caused by excessively long training, complex models with a large number of trainable parameters, or limited amounts of data. One of the most effective ways to combat overfitting is to increase the quality of the training dataset by adding more examples and by artificially generating new combinations using data augmentation techniques such as random crops, rotations, flips and distortions.

A simple and effective way to limit overfitting in classification neural networks is to use Dropout [61]. Dropout layers randomly set a fraction of the activation values to zero during training (but not during evaluation). The removal of information either on the inputs or in hidden layers forces the network to build up resistance to missing information and redundancies in its pattern recognition capabilities, reducing the probability of overfitting to a limited set of data. Although the training process becomes slower due to the smaller amount of information available, Dropout makes it generally possible to train for longer periods without overfitting.

2.4 Existing methods to increase ray tracing efficiency

Ray tracing is an image rendering technique that is typically very inefficient in terms of computational power. It requires to shoot a large number of rays for each pixel to obtain global illumination, and each ray will need to intersect every geometrical element present in the scene.

There have been many past efforts to increase the efficiency of ray tracing.

2.4.1 Optimizing sampling

A most naïve implementation of Monte Carlo Path Tracing would take random bounces from surfaces and towards light sources. The model is simple and unbiased, but extremely inefficient due to the fact that most light sources are highly concentrated in small parts of the scene, which translates in very high variance in the sampled intensities.

Let's consider a simple material: a perfect mirror, which allows an incoming light ray to be reflected in exactly one outgoing direction. If a path tracing algorithm always produces random bounces from an intersection point, the probability of an outgoing ray to have the same direction as the mirror's reflection is infinitesimally small, and while the algorithm is still theoretically unbiased, it's extremely inefficient. A better approach is to generate the new reflected rays according to the surface properties; in this case to be always perfect reflections. A similar scenario happens when sampling light sources: when the primary source of light is a sky dome with a clear sun, the majority of the luminous contribution is created by the small sun disk. The rest of the sky is also a light source, and although it has a huge area it produces far less flux than the sun. Randomly sampling the entire dome would work, but weighting the samples towards the sun would be far more efficient.

From these examples follows the intuition that weighting the proposed sampling distribution to match that of the BRDF or that of the incoming light energy can drastically improve the speed of raytracers. Both BRDF and incoming radiance in fact appear in the Rendering Equation (Section 2.1.6), and the optimal proposal distribution is that of the product distribution of BRDF and incoming radiance. We formalize the concepts of estimate variance and importance sampling.

In a noisy image, the high variance of the samples makes it necessary to obtain a larger number of samples to clear out the noise, but it is time consuming and computationally inefficient due to the quadratic increase in the number of samples required for a visible improvement in noise reduction.

In a Monte Carlo estimate

$$I_N(f) = \frac{1}{N}\sum_{i=1}^{N} f(x_i)$$

The variance is

$$var(I_N(f)) = \frac{1}{N} var(f(x_i))$$

The variance of the estimate is inversely proportional to the number of samples taken, and the final image converges slowly towards the ground truth.

Past research has focused on `Importance Sampling` to optimize the variance of the drawn samples.

Importance Sampling provides an unbiased estimate of an unknown function $I(f)$ by using a proposal probability distribution $p(x)$

$$I(f) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{f(x_i)}{p(x_i)}$$

The variance of the Importance Sampling estimate is minimized when the variance of $\frac{f}{p}$ is low. Therefore it is optimal to choose the proposal distribution p to be similar to that of f.

Importance Sampling requires drawing random samples from a probability distribution. When a *probability mass function* is known, it is necessary to compute a *cumulative distribution function* and invert it to obtain the samples.

While the probability distribution of the incoming radiance is generally not known during rendering, other distributions might be. In a ray tracer, it is safe to assume that the BRDFs of the surfaces are well defined, and that the intensity distribution of environment maps is also known. These two distributions can be used together using Multiple Importance Sampling methods, originally proposed by Veach and Guibas [65] [66], which alternate and weight the sampling from multiple distributions. Bidirectional Importance Sampling [28] samples directly the product distribution using Rejection Sampling or Sampling Importance Resampling to obtain the optimal proposal distribution.

2.4.2 Image level noise reduction

Operating on the final rendered image is the simplest way of reducing noise. As some ray tracing algorithms generate a kind of noise which resembles the noise captured by digital camera sensors, general purpose noise reduction algorithms can have a good level of effectiveness.

However, this is not a real solution to the amount of noise produced by ray tracers, and it will also not work very well in the rendering of animations.

2.4.3 Clamping

Clamping is an effective way of reducing fireflies. The value of a pixel contribution is limited to a maximum amount to avoid cases where extremely bright outliers create excessively bright pixels that take a long number of samples to average out. Clamping can be defined in absolute terms, or as a ratio of rejected-to-accepted samples.

In most cases clamping will create a visible difference in the final image when rendered to an infinite number of samples, and it will therefore make any ray tracing technique biased. Clamping the brightness of rays makes the overall image darker than the ground truth. The energy removed by light rays by applying clamping is permanently lost and it is not added back to the scene, making ray tracing engines that use clamping physically inaccurate.

Clamping is widely deployed as an option to reduce fireflies in popular ray tracing engines [22] [18].

2.4.4 Portals

Portals [13] are abstract objects that indicate surfaces through which all external light should pass. In interior scenes with bright external lighting, such as direct sunlight, path tracing and even bi-directional path tracing can struggle to reach small openings, such as windows, from which most of the light comes. Portals are useful to guide light rays into the scene.

Using carefully placed portals can dramatically increase rendering performance of interior scenes, but require manual input and can be hard to optimize in case of many small openings.

2.4.5 Russian Roulette

A practical Monte Carlo Path Tracing algorithm cannot trace paths for an infinite number of bounces for obvious efficiency reasons: the contribution of paths after a large number of bounces can rapidly fall to become almost insignificant, while being very expensive to compute. A simple solution is to cut off the maximum number of bounces to a predefined depth level, and return no light contribution if the maximum depth is reached.

Limiting the maximum recursion depth introduces bias to the algorithm, as long paths that have been cut off can still contribute to the brightness of the scene. The Russian Roulette [64] solves this problem by randomly terminating paths with a fixed probability $1-p$, and increasing the contribution of non-terminated paths by $1/p$. This method is mathematically proven to be unbiased.

The Russian Roulette increases the variance of the estimator, but it is useful to increase the efficiency of the Path Tracer by reducing the amount of computation required per sample, as many rays will be terminated early. The majority of visible light only requires approximately three bounces in a typical scene, and increasing the variance of longer paths that contribute less to the scene is a good tradeoff.

2.4.6 Metropolis Light Transport

Metropolis Light Transport [67] is a sampling technique that typically performs much better than random sampling in scenes with strong indirect illumination. Instead of attempting to obtain a uniform sampling over the global sampling space of all paths, Metropolis sampling is based on mutations of existing paths. A starting path is necessary, and each subsequent path depends on the previous one. As the number of paths generated goes to infinity, the covered sampling space is unbiased.

The advantages that Metropolis Light Transport brings are clear if we consider that the initial path found using standard Bi-directional path tracing is a valid connection between the camera and a light source. Mutating an existing, possibly difficult to find path, can produce other difficult paths, allowing the ray tracer to solve efficiently difficult lighting situations. The use of a single initial path makes it necessary to use different copies of the algorithm rendering the same image in order to reduce variance on the image level. Additional characteristics of Metropolis Light Transport are also the patchy areas visible on low quality renders caused by the local search mechanisms, and the flickering of the frames of rendered animations containing variance on the image level, compared to the pixel-level flickering of animations rendered with Path Tracing.

2.5 Related Work with Machine Learning

Along with progressive algorithmic refinements on ray tracing techniques, and the introduction of new methods that allow faster convergence or lower variance, recent research has made several attempts at applying Machine Learning to computer graphics.

These studies and proposals have operated on different levels, such as post-processing of the output image, generation of ray sampling directions, or direct derivation of illumination values, and have demonstrated various degrees of success and applicability.

2.5.1 A Machine Learning Approach for Filtering Monte Carlo Noise

A Machine Learning Approach for Filtering Monte Carlo Noise [36] is an early successful attempt in using machine learning to augment the quality of the output of a Monte Carlo ray tracer by removing the noise produced by a Path tracer.

The authors used a neural network to generate optimal feature-based filter parameters to denoise the output of a MC path tracer. Previous work had already shown that denoising filters that used additional feature could easily outperform those that only relied on pixel RGB values. The approach used a ray tracer to generate *primary features*. Primary features are data values that can be computed during a standard rendering pass for each pixel: world coordinates, surface normals, texture values, illumination visibility. When generating primary features for an image generated with several samples per pixel, the feature values are averaged for each pixel. The illumination visibility, a binary value, is encoded into the fraction of samples in which shadow rays were not occluded.

Secondary features are computed from the primary features, and include metrics such as gradients, mean deviation, and mean absolute deviation. A total of 36 secondary features is computed for each pixel.

The neural network is a Multi Layer Perceptron which takes on the input layer the secondary features and outputs filter parameters.

The approach is able to produce high quality results, and shows its ability to outperform all other existing denoising algorithms. The inclusion of texture and illumination visibility data in the input layer shows its usefulness in preserving fine detail that would otherwise be lost in the denoising process, and the network is able to work quite well when extrapolating outside of its training limits, such as when presented with glossy materials.

The system also shows good performance, averaging a few seconds to denoise a single image at 1200x800, and around one minute for a spatio-temporal filtering of an animation frame.

While this method proves to be successful in many situations, it requires the use of a modified ray tracer based on PBRTv2 [53] and is not easily applicable to other rendering engines. Furthermore it does not solve underlying weaknesses of path tracing, such as solving difficult paths involving multiple transmission layers required for interior lighting through windows and caustics.

2.5.2 Filtering Monte Carlo Noise in Ray Traced Images

A Monte Carlo denoiser that is not closely tied to a specific rendering engine, and that performs direct image to image translation, was proposed by [49]. Deep neural networks are trained to remove noise from a render. The project has obtained good success, often managing to effectively eliminate all visible noise from the picture and demonstrating itself to be much more effective than general-purpose noise removal tools for photography.

In order to preserve fine detail, the system takes as inputs the rendered image, and an OpenGL material rendering which is used to improve the quality of sharp edges and textured surfaces.

Being a noise removal tool, the neural network cannot re-create information that is not present in the partially rendered image, and can occasionally generate artifacts in difficult situations.

This method is a post-processing noise removal filter specifically trained for Monte Carlo ray traced images. It shows the usefulness of allowing a neural network to learn some domain specific information, in this case how to remove the typical noise produced by Monte Carlo ray tracers. However, it shows significant limitations in the quality of the output compared to the same scene rendered with a high number of samples, and it has difficulties in adapting to noise produced by different rendering engines. Metropolis Light Transport (Section 2.4.6) and Photon Mapping (Section 2.1.8), for example, produce noise patterns that are very different from those produced by a Path Tracer, and would require a different training set to achieve good results.

While this approach shows some limitations in complex scenes, its extreme simplicity and adaptability are important strengths. It does not require the modification of any rendering engine to produce additional primary features, and it works with any ray tracer that is capable of producing a material preview or direct illumination pass.

2.5.3 Kernel-predicting Convolutional Networks for Denoising Monte Carlo Renderings

The work by [26] further improves denoising neural networks. Deep Convolutional Neural Networks (CNN) are used to denoise ray traced images in their output space.

Diffuse and specular components of the low quality render are separated and processed by two separate deep CNNs. This approach allows to achieve high quality results because it separates components that present very different levels of dynamic range and noise characteristics. The filtered components are recomposed into a color image in a post processing step. An additional step useful to increase the stability of the network in presence of high dynamic range is a log transform applied to the specular component. The authors have found out that applying a log transform to the input of other methods can improve quality as well.

Training of the network used images rendered at low quality (16, 32, 128 Samples/px) in supervised learning. This method currently yields state-of-the-art noise reduction quality in production renders at a low to medium number of samples.

Similarly to the machine learning method described in Section 2.5.2, this method is highly optimized for a specific kind of renderer, and makes assumptions about the input quality level of the image. It is harder to apply in general with arbitrary rendering applications, and it heavily depends on the quality of the training set. Some kinds of fine details in the scene can be easily mistaken for noise and removed, and specific effects such as volumetric smoke cannot be dealt with properly if the training set doesn't have enough examples containing this kind of elements.

The CNN yields state of the art denoising results, but it shares some of the weaknesses of the previous denoising methods: it requires the use of a modified rendering engine, it is not universaly applicable, and it does not solve some fundamental limitations of ray tracers.

2.5.4 Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder

A different approach to applying machine learning post-processing to rendered images is to reconstruct details that are missing, rather than removing noise from the picture. In an animation, successive frames retain much of the information from their neighbours, and computation can be highly optimized by learning temporal features from different frames and reconstructing missing details.

A Path Tracing render with a single sample per pixel often results in an image which is black for the majority of pixels, except for those that were generated by a path connecting to a light source. While the image may look much darker than the reference, the average brightness is unbiased because most of the individual pixels are much brighter than the maximum dynamic range of the output.

The method by [29] proposes the application of denoising autoencoders to process Monte Carlo images rendered with a single pass.

The autoencoder network is trained to reconstruct the full input data given a more compact intermediate representation. The noisy image, in the encoder stage, is converted to an internal representation in the network, and the decoder stage up-samples the representation to obtain an output image which is as close as possible to the reference. An autoencoder network should also be stable: if fed as input the result of its own processing, the new output should not present many differences. Each stage uses recurrent connections to retain information from previous frames.

This reconstruction method yields highly believable images with global illumination despite the small amount of information that the network can work with. Combined with a fast path tracer, a noise free image can be obtained in interactive times, in the order of a few hundreds of milliseconds. While this system does not always yield highly accurate results, either due to the lack of training examples or of fine details in the low quality render, it does offer a realistic, noise-free preview with global illumination, and proves the capability of networks to solve difficult cases in which data is scarce.

2.5.5 Deep Scattering

The work by [37] is a rare example of Machine Learning applied to a level different from that of the output image. Neural networks are used to estimate Radiance values in volumes with high scattering values. Dense volumes, such as clouds and fog, are very slow to simulate with standard Monte Carlo Path Tracing because each light ray bounces many times within the volume with unpredictable directions before exiting and reaching a light source. Monte Carlo integration is used to integrate both on the sphere of ray directions, and on the length of the light path. Effects such as dark patches and silverlining in clouds are highly desired for a realistic image, but they are hard to produce with approximating solutions.

The Deep Scattering approach encodes the 3D structure of the cloud in a hierarchical point stencil model able to capture both the fine details near the central point, and the overall shape of the cloud on a large scale. The hierarchical model is oriented towards the direction of the sunlight source, and a neural network is trained with several examples of clouds to predict radiance values.

The system is able to render realistic clouds that are virtually indistinguishable from the ground truth rendered with Path Tracing, while being several orders of magnitude faster.

The project shows how effective a domain-specific application of machine learning can be in computer graphics, although the system has considerable difficulties in dealing with any kind of volume that doesn't resemble a cloud, such as unusual shapes, smoke, or high density volumes.

2.5.6 Reinforcement learning importance sampling

An interesting modification of a general-purpose Path Tracer by Dahm and Keller [30] shows that reinforcement learning and the rendering equation are closely related, and that it's possible to significantly improve path tracing performance by learning and storing importance information on the scene surfaces. Reinforcement Learning is a Machine Learning technique which attempts to achieve optimal results by assigning rewards for good solutions, and penalties for bad ones, allowing the system to progress towards an optimal answer.

Hemispherical importance maps are stored on voronoi cells on each scene surface. Each importance map is progressively updated using Reinforcement Learning, and is used to obtain ray tracing paths that are more efficient and more likely to carry important light contribution. The aim of the learning process is to guide rays towards areas with higher radiance emission.

At each light-surface intersection, the importance map associated with the surface point is used to generate the next sample. The importance values can be used to guide the sampling of secondary rays: higher importance directions are sampled more frequently, and the energy transported is reduced proportionally to the probability of sampling the direction.

This approach shows very good performance in the general case, because the Reinforcement Learning system adapts quickly to new data, and the use of importance maps to guide light rays is effective. The importance maps are used for every bounce along a light path, therefore the system can adapt well to difficult scenes where indirect light is only reachable after several bounces. Compared to Metropolis Light Transport, the Reinforcement Learning approach shows more uniform noise patterns, because each pixel is sampled independently. The average path length and number of zero-contribution paths are drastically reduced compared to Path Tracing as light rays are guided towards places that emit more light and are more unlikely to go to dark spots that generate little contribution.

Reinforcement Learning Light Transport has demonstrated to be a capable ray tracer in difficult situations, but it presents challenges and possible areas of improvement. Storage of surface patches for the importance maps needs to scale with the size of the scene or the resolution of the patches. The number of hemispheres has additional implications on the learning rate of the system, because each hemisphere needs to be updated independently. Due to the necessity to apply updates to the hemispheres, it is a challenge to reduce memory usage by merging similar patches, as it is not easy to predict whether two patches have similar behaviour with respect to the light they receive. Reinforcement Learning requires a period of learning at the beginning of the rendering, and can have an impact on images rendered at a very low number of samples per pixel and before the importance maps have converged to to be optimal.

2.5.7 Deep Shading

Some recent projects have focused specifically on the integration of Global Illumination and other effects using neural networks. *Deep Shading* [52] uses primary geometrical features obtained from an OpenGL rasterization pass and a Deep Convolutional Neural Network to generate a variety of screen-space effects, including Ambient Occlusion, Motion Blur, Anti-Aliasing and Diffuse Indirect Illumination. The target effects do not try to be physically correct, but are comparable to the output of some real-time algorithms commonly employed in video games.

The main attributes used are: positions, normals, depth, depth from focal plane, material diffuse and glossy coefficients, scattering coefficients, and pre-computed direct light.

The network is a U-shaped CNN. The downscaling and upscaling branches are symmetric and work with power of 2 sizes.

The approach is capable of generating convincing screen-space effects at interactive rates. The OpenGL attribute generation is fast and easily implemented in a standard rendering pass, and the forward pass of the network was also re-implemented as a shader to eliminate the communication overhead between the rendering engine and the process managing the neural network. The use of a U-shaped network structure allows to efficiently process per-pixel data, and to gracefully scale to higher resolution inputs.

The outputs can match in quality and speed some of the state-of-the art real-time algorithms typically used in video games, such as HBAO ambient occlusion and FXAA anti-aliasing. An interesting capability of Deep Shading is the possibility to train a single network that combines several screen-space effects, effectively reducing the evaluation time for any given number of effects to a constant.

2.5.8 Deep Illumination

Deep Illumination [63] takes a different approach in the estimation of screen-space global illumination effects. The project focuses the efforts on computing high-performance indirect illumination from diffuse surfaces, aiming at interactive frame rates.

The approach is different from *Deep Shading* as the network is not trained on general features, but is specialized on a per-scene basis. The method effectively stores radiance transfer functions of diffuse indirect bounces in a neural network.

Conditional Generative Adversarial Networks (CGAN) [51] are used to obtain global illumination effects given surface normals, distances, diffuse constants and direct illumination. While a standard Generative Adversarial Network (GAN) aims at generating new content from random noise, a CGAN produces outputs based on input characteristics. The generator network is based on 2D convolutions and attempts to generate global illumination images, and the discriminator network learns to distinguish real global illumination images from those that were obtained form the generator. The two networks play a Minimax game until it is not possible to distinguish the generator's output from the ground truth. Reference ground truth examples are obtained from Path tracers and other methods capable of Global Illumination effects. Additionally to the discriminator's output, an additional L1 loss function is used to ensure that the generator produces images coherent with the given input data.

This method is capable of producing high quality Global Illumination effects even when new shapes and objects are introduced in the scene, and is temporally stable for use in animations and video games.

While the method is fast and temporally stable, we believe that the support of only diffuse materials and the necessity of training the network for each scene as a preprocessing step are limiting factors. Deep Illumination targets GI effects in a very restricted environment with a single scene and very few moving objects, and fails to model Ambient Occlusion correctly.

3 Motivation

Ray tracing is inefficient, but there is a large amount of unused information in the scene that could be useful to accelerate rendering convergence. Existing applications of machine learning in computer graphics mostly work on the final image level to either remove some sampling noise or add some post-processing effects including Global Illumination.

Current approaches to add GI effects on the image level have limitations: Deep Shading adds simple real-time effects such as Ambient Occlusion and cannot process highly complex lighting configurations due to the limited amount of information generated by the OpenGL rendering engine, and Deep Illumination requires per-scene training, only supports diffuse materials, and cannot produce Ambient Occlusion.

We believed many of these limitations could be overcome by pushing the use of Machine Learning to a level deeper than that of the final rendered image. A lower level of information can make an approach easier to generalize, and the use of data that is more closely related to the underlying geometrical structure of the scene can help achieve more accurate results under complex lighting conditions.

We propose One Shot Radiance (OSR) to accelerate the rendering process by making use of information contained in the 3D scene description to efficiently calculate indirect light contributions.

3.1 Objectives

In this project we combine ray-traced radiance information with geometrical data such as surface positions and normals to directly predict indirect illumination for global illumination renders. The method is suitable for scenes with a large amount of indirect lighting, and aims to efficiently produce a low-noise indirect illumination layer that can be combined with a direct illumination pass for a complete image.

We train a deep neural network capable of estimating partial indirect illumination from a few light samples and geometrical information, and create a proof of concept rendering engine that produces global illumination by combining the predicted indirect illumination with a direct illumination pass.

We focus on the following objectives for OSR:

- Obtain smooth and noise-free indirect illumination estimation.
- Achieve good performance on the indirect illumination pass.
- Achieve low bias on typical indoor scenes with high indirect illumination.
- Support a wide range of standard non-scattering materials.
- Remove the necessity of pre-processing and per-scene training.

These objectives require to:

- Train a deep neural network that predicts indirect illumination on a given scene point starting from some illumination samples and the local scene geometry.
- Use the predictions of the neural network to obtain an indirect-illumination only pass of the final image.
- Integrate OSR in an existing ray tracer to obtain a full global illumination rendering engine.

4 Method

We illustrate the intuition and method of One Shot Radiance, and a high level overview of its implementation. Section 4.1 presents some general intuitions, Section 4.2 shows how OSR evaluates the Rendering Equation, Section 4.3 illustrates the interpolation and refinement strategy, Section 4.5 describes the neural network model.

4.1 Scene knowledge

An average 3D scene can contain millions of triangles, and provides a very large amount of information that a typical ray tracing algorithm only partially exploits. Randomly bouncing secondary rays around the scene, while being correct and simple to implement, is not a very efficient way to obtain an image.

Some of the rays, like the one in figure 12, can go in areas that don't generate any contribution. These rays can be completely avoided.

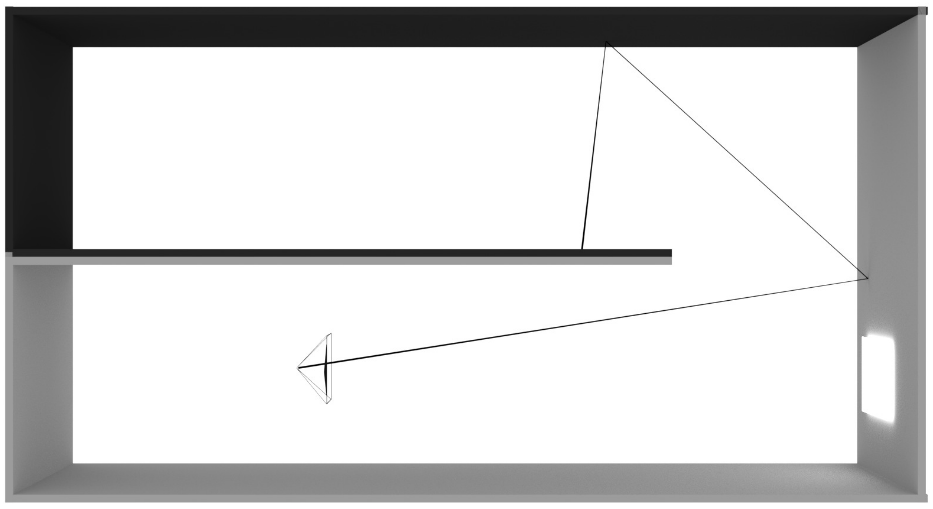


Figure 12: In path tracing some paths will lead to areas that do not generate any contribution. The highlighted camera ray bounces on a diffuse surface, and the secondary ray only hits black surfaces.

Others can be very unlikely, but generate highly visible effects, such as caustics. These rays are required to be sampled in order to obtain a correct image, even if the probability of obtaining them randomly is extremely low.

Others can be important, but similar to many other rays, such as all secondary rays that hit a large uniformly-illuminated matte surface. The number of these rays can be reduced without affecting the quality of the final picture.

These simple examples indicate that the 3D scene contains surface and light information that can be used to improve the efficiency of rendering.

The idea in OSR is to use geometric scene information local to an intersection point, together with a few path-traced samples, to obtain a small radiance map centered around the intersection containing indirect illumination. This radiance map contains all illumination that is received at the surface from the upper hemisphere by light which bounced on another surface. Evaluating the radiance map with respect to the BRDF of the intersection surface point material yields in the final image the total indirect lighting of the pixel. A small number of samples is evaluated with Multiple Importance Sampling [65]: a few sampling directions are chosen according to the BRDF probability density distribution of the surface, and a corresponding set is chosen from the radiance map. Each sample uses the light information of the selected radiance map pixel and returns the final radiance value according to the BRDF of the surface and the viewing direction. The radiance map is mapped at infinity distance around the intersection point, and visibility tests are skipped as the radiance map contains visibility information. The radiance map does not contain any direct illumination, such as rays hitting light sources or environment maps directly, therefore first order bounces and shadow rays are excluded from the final image.

A standard path tracer eventually fully evaluates each hemispherical radiance map at every intersection point in order to converge, and OSR accelerates the process of evaluating the total contribution of higher depth bounces using a neural network.

The first bounce direct lighting component is still evaluated using a normal path tracing algorithm, but it is not a significant source of complexity or noise due to its limited path depth (as all higher depth paths are handled by the neural network), and can be optimized using existing importance sampling and visibility testing methods.

4.2 OSR Rendering

OSR splits rendering of direct illumination and indirect illumination. The indirect illumination component is evaluated using cached radiance maps, obtained efficiently using a deep neural network that reads local geometrical information of the scene.

A standard Path tracer evaluates the Rendering Equation [35] recursively

$$L_o(x, \vec{\omega}_0) = L_e(x, \vec{\omega}_0) + \int_{\Omega_{4\pi}} f_r(x, \vec{\omega}_0, \vec{\omega}_i) cos(\theta_i) L_i(x, \vec{\omega}_i) d\omega_i$$

Figure 13 shows that a ray in Path tracing bounces on surfaces in the scene until a light is hit, or the path is terminated. The recursively evaluated radiance, the BRDF f of the material and the viewing direction $\vec{\omega}_i$ are used to obtain the final pixel value. OSR approximates the evaluation of the Rendering Equation by collapsing all the light bounces beyond the first in a single step. The ray tracer shoots a ray to find the first intersection point, and evaluates on its hemisphere a One-Shot Radiance map, a Depth map, and a Normals map. These three images are the inputs of our Convolutional Autoencoder, which returns a higher quality Radiance map. The output Radiance map is placed back on the hemisphere around the first intersection point, and contains an approximation of all the recursive indirect radiance that path tracing would evaluate over time. The Rendering Equation therefore becomes

$$\begin{aligned} L_o(x, \vec{\omega}_0) = L_e(x, \vec{\omega}_0) + \\ \int_{\Omega_{4\pi}} f_r(x, \vec{\omega}_0, \vec{\omega}_i) cos(\theta_i) R_i(\vec{\omega}_i) d\omega_i + \\ \sum_{l \in lights} f_r(x, \vec{\omega}_0, \vec{\omega}_i) cos(\theta_i) L_e(l, \vec{\omega}_i) V(x, l) \end{aligned}$$

We replaced the recursive radiance term with R, the high quality radiance map obtained from the neural network, and evaluate it in the same way according to the BRDF f of the surface and

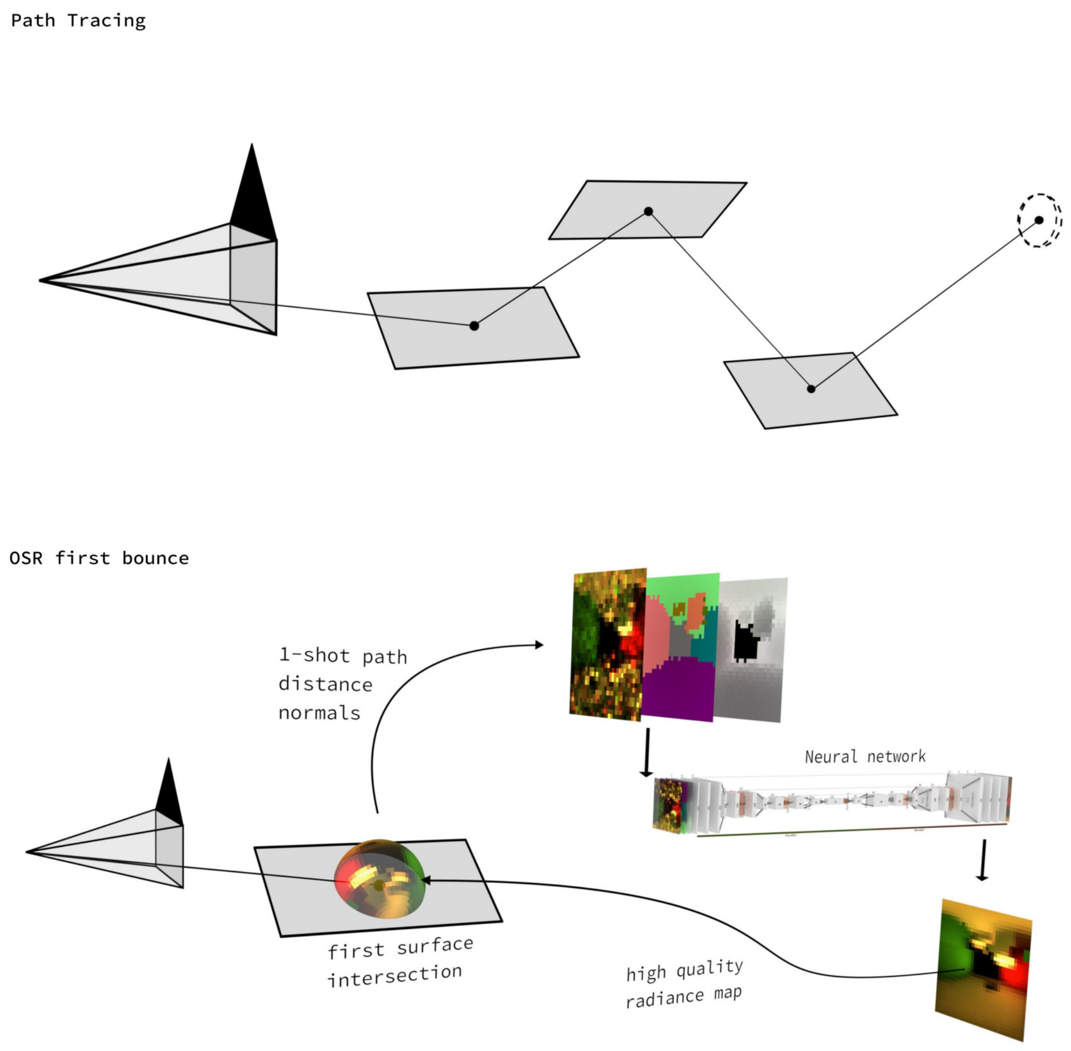


Figure 13: Path tracing evaluated multiple bounces until a light source is hit or the path is terminated by the maximum depth or the russian roulette. OSR only uses the first intersection to sample low quality intensity, distance and normals and uses a neural network to estimate a radiance map of all the indirect light received at the point. The radiance map is evaluated to compute indirect lighting, and direct light paths are added.

the viewing direction $\vec{\omega}_i$.

Evaluating the entire Radiance map R would still be expensive. Our implementation uses Multiple Importance Sampling [65] with a small fixed number of samples to obtain indirect illumination values. We alternate sampling from R as if it were an Environment map with perfect visibility, and from the probability distribution of the BRDF.

To include direct illumination, the first intersection point also generates shadow rays to light sources (not shown in the figure). It is important to exclude direct light information from the Radiance maps, as they don't have a resolution high enough to produce accurate shadows.

4.3 Progressive refinement and interpolation

The interpolation method of OSR does not use any precomputation to determine the best sampling locations, but is based on a regular grid on the image film. At each vertex a radiance map is evaluated, and interpolation is used within the grid. After each complete pass on the image, the grid size is reduced by a constant factor, allowing the algorithm to progressively refine the resolution of the indirect illumination.

The interpolation strategy is based on both the distance from the sampled radiance maps and the orientation of the surface normal. The weight of each radiance map for pixel i is:

$$w = w_p \cdot w_n + w_p + \epsilon$$

Where the position weight w_p is:

$$w_p = dist(p_i, p)/r$$

$dist(p_i, p)$ measures the distance in pixels on the film plane between the pixel being evaluated and the cached radiance map, and r is the distance between two vertices in the grid.

The normals weight w_n is maximal when the pixel's intersection point and the cached radiance map's normals point in the same direction, and becomes zero when their dot product is negative:

$$w_n = 1 - max(0, (\frac{n_i}{|n_i|} \cdot \frac{n}{|n|}))$$

The weights w of the radiance maps are normalized and converted into sampling probabilities for the radiance sampling.

This interpolation method here presented can still cause light bleeding and softness when surfaces are at different distances from the camera and have the same orientation. We introduce a third component to complete the 3D dependencies of the interpolation algorithm, the distance from the camera:

$$w_d = max(0, 1 - \frac{|z_i - z|}{z})$$

Where z is the distance of the intersection point from the camera's origin, and z_i is the distance of a cached sample from the camera's origin. We integrate w_d in the overall weighting system as:

$$w = w_p \cdot w_n + w_p \cdot w_d + w_p + \epsilon$$

4.4 Algorithms

Algorithm 1 Evaluating indirect illumination for a pixel

```
1: procedure PixelIndirect(Pixel)
2:     $Ray \leftarrow Camera.Ray(Pixel)$    ▷ Generate ray from main camera
3:     $Intersection \leftarrow Scene.intersect(Ray)$    ▷ Obtain first intersection point
4:     $IntCamera \leftarrow newHemisphericCamera(Intersection)$    ▷ Create a hemispheric camera at intersection point
5:     $Intensity \leftarrow IntCamera.renderPath(1)$    ▷ 1 Sample/px Path
6:     $Normals \leftarrow IntCamera.renderNormals()$
7:     $Distance \leftarrow IntCamera.renderDistance()$
8:     $Predicted \leftarrow Network.predict(Intensity, Normals, Distance)$    ▷ Run neural network
9:     $L \leftarrow newSample$
10:    $N \leftarrow 0$
11:    for all $P \in Predicted$ do
12:        $L \leftarrow L + Intersection.BRDF(P.direction, P.value)$
13:        $N \leftarrow N + 1$
       return $L/N$
```

Algorithm 2 Evaluating indirect pass

```
1: procedure RenderIndirect(Scene, Camera)
2:     $Radius \leftarrow InitialRadius$
3:     loop
4:         $Tasks \leftarrow Scene.split(Radius * TaskSize)$
5:         for all $T \in Tasks$ do
6:             $Tiles \leftarrow T.split(Radius)$
7:             for all $Pixel \in Tiles$ do
8:                 $L \leftarrow PixelIndirect(Pixel)$
9:                 $Camera.addIndirectSample(L)$
10:        $Radius \leftarrow Radius * RadiusUpdateRatio$
```

We provide high level algorithms of the OSR rendering process.

Algorithm 1 describes the evaluation of a pixel value using the radiance map.

Algorithm 2 describes the progressive refinement updates. The subdivision of the image space into `tasks` allows to limit the amount of memory and computation that is taken by OSR at each step, allowing the algorithm to gracefully scale and be able to operate on large scenes and high resolution. The parameters `InitialRadius` and `RadiusUpdateRatio` define the starting influence range of an OSR sample, and the ratio by which the radius is updated after a full pass is completed on the image. With a `RadiusUpdateRatio` less than 1, the radius slowly converges towards 1, and the algorithm progressively reduces the size of each tile and refines the resolution of the output.

4.5 Neural Network

The Neural Network is one of the core components of the OSR project. The objective of the network is to predict a smooth and accurate radiance map from primary features that can be computed quickly by a raytracer.

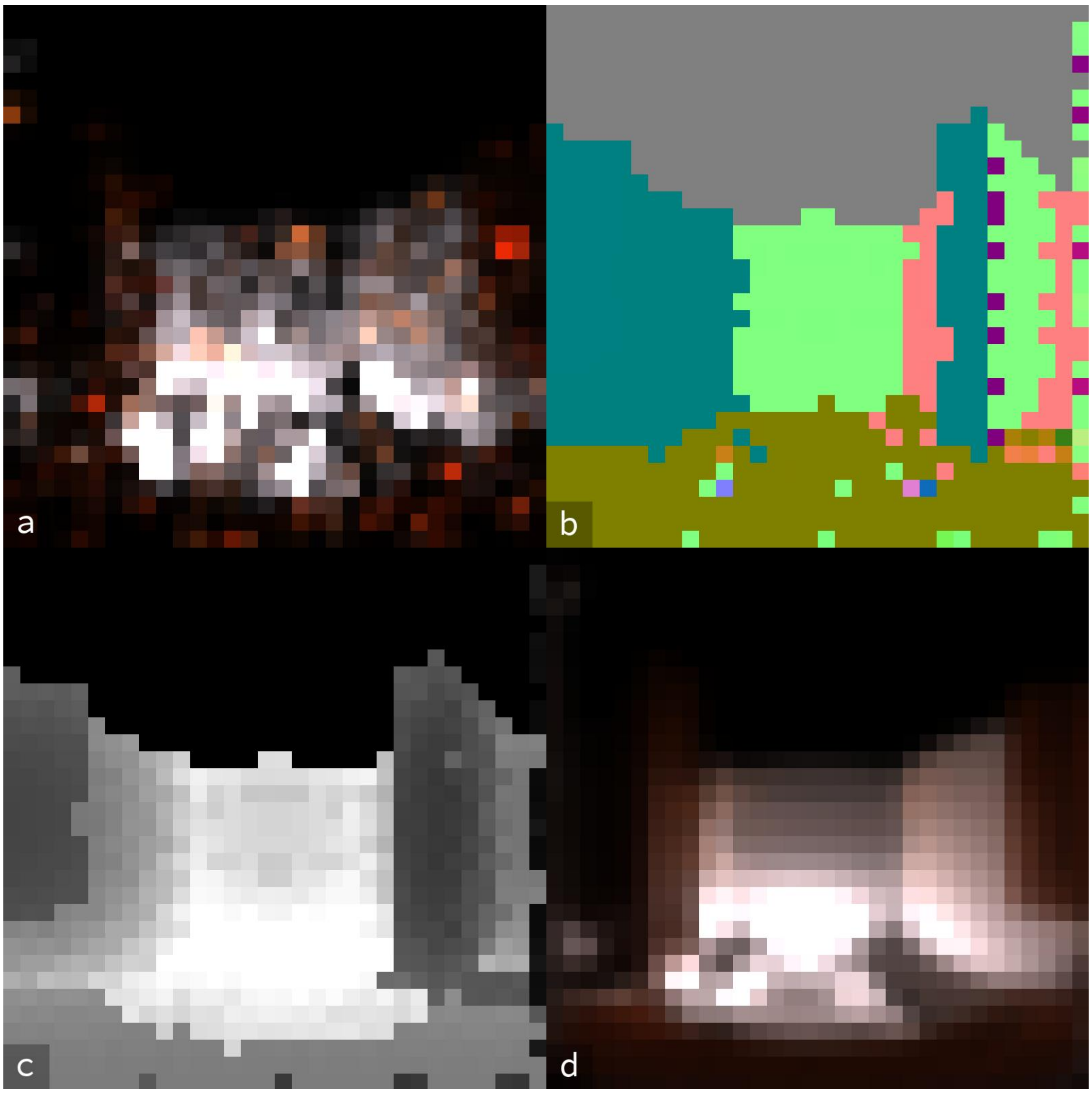


Figure 14: Examples of latitude-longitude hemispherical maps generated by OSR. This sample was generated from one intersection in the scene `bathroom`, part of the PBRTv3 example scenes [20]. Image (a) is a path traced intensity map at 1 sample/pixel. Image (b) is a normals map. Image (c) is the distance map. Image (d) is the same intensity map, traced at 4096 samples/pixel, used as reference ground truth during training.

4.5.1 Input and Output Data

The output data of the network is a radiance map that will be used for the subsequent rendering stages.

The output is a 32x32 pixel color image with individual Red, Green and Blue channels. Each data value is a 32 bit floating point, and there are a total of 3072 floating point values for one output image.

The inputs are three 32x32 pixel maps.

The *one-shot* intensity map is produced by a path tracer using 1 sample per pixel. The *normals* encode the surface normals at the first intersection. The *distance* map is the distance between the main camera and the intersection point. There is a total of 7 layers.

Each of the hemispherical maps in both inputs and outputs is a square 32x32 pixel hemispherical latitude-longitude equirectangular projection [7]. The coverage of a hemispherical map is not the full sphere, but only the upper hemisphere visible from a scene intersection point, with the surface normal aligned towards the Z direction of the latitude-longitude map. The `up` vector used in the hemispherical maps is the same as the one specified in the global scene, and it is rotated by 90 degrees towards the global Y vector only when the surface normal also points towards positive Z.

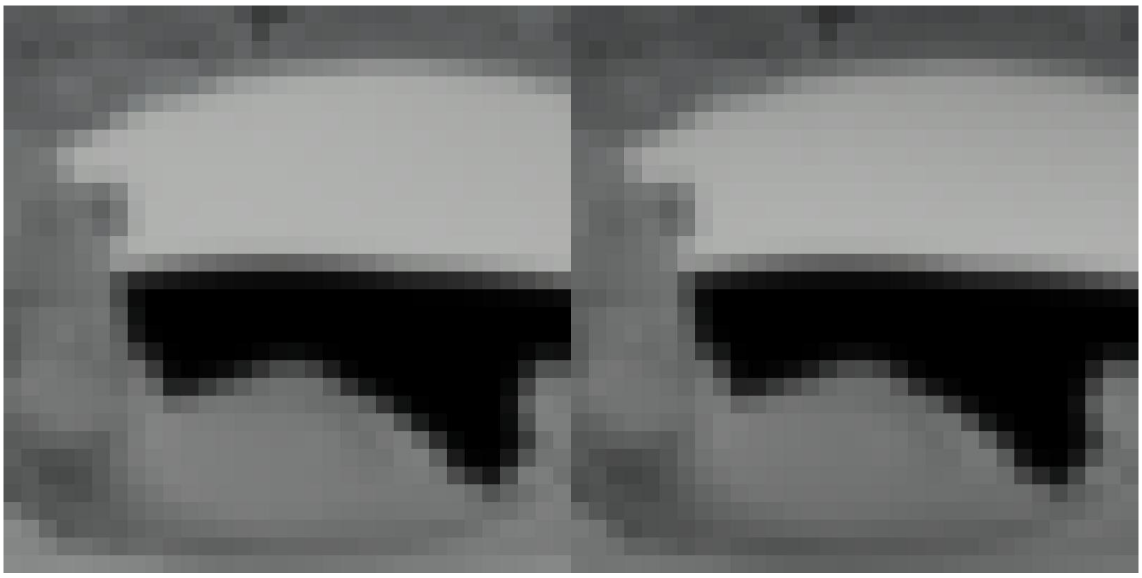


Figure 15: Left: Original latitude-longitude image. Right: The same image, with intensity correction on latitude. The corrected image presents slightly darker intensity near the top and the bottom. Being a hemispherical map instead of a full environment map, the correction is not very strong because the top and bottom edges have a latitude of $\pi/4$ instead of $\pi/2$ in a full environment map.

None of the images processed by the neural network account for the latitudinal intensity adjustment required to ensure that the cumulative intensity is correct [9]. In an equirectangular projection the areas are not preserved, and it is necessary to adjust the intensity maps to ensure that sampling from a lat-long mapping preserves the original intensity values. The areas near the poles are stretched out horizontally and have a larger area than originally, requiring a multiplication by $sin(\Theta)$ of the intensity before computing any integral over it. This correction is handled by the rendering software at a later stage:

$$I(\Theta, \Phi) = P(\Theta, \Phi)sin(\Theta)$$

The choice of using a latitude-longitude format for the mappings derives from the simplicity of converting between cartesian and polar coordinates, and because PBRT already implements an `Environment Camera` that uses an equirectangular projection. In the OSR project the `Environment Camera` has been modified into the `Hemispheric Camera` to provide only a single hemisphere field of view. Although the lat-long format was easy to implement, it is not the most compact representation, and presents a significant amount of surface distortion at the poles.

4.5.2 Data Normalization

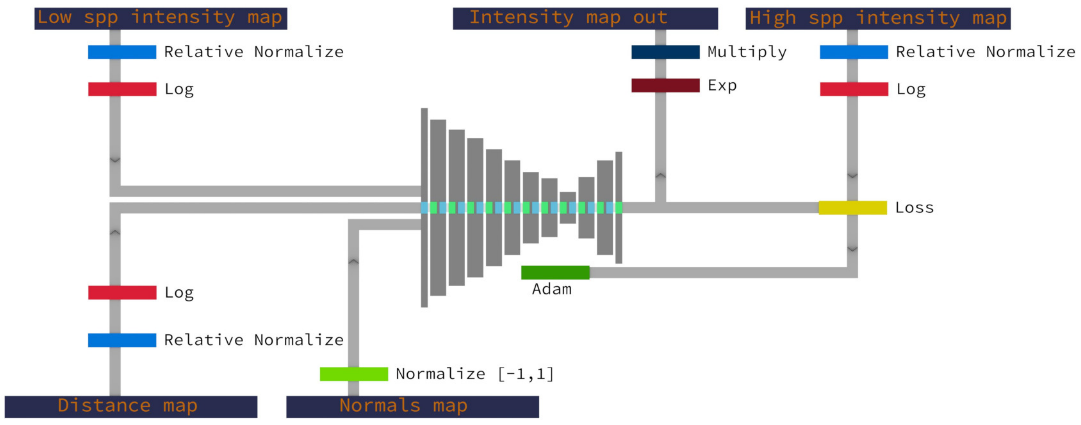


Figure 16: Detailed normalization and transformation steps for input and output data

With the exception of the normals data that always encodes normalized vectors, the hemispherical maps have an unknown magnitude.

Intensity values in the RGB channels can have several orders of magnitude in a single scene, and differ even more across different scenes. It would be extremely ineffective to feed the neural network with raw intensity data from a path tracer, as the magnitude of the values is unknown and the network would be unable to place them on a relative scale. Training using raw data can yield disappointing results due to the inability of most loss functions to work on un-normalized data.

In the OSR training process, several input normalization methods have been attempted:

- `Standard normalization` - The mean of each example is centered around 0, and each value is divided by the standard deviation. This is a common normalization strategy proposed by LeCun [47].
- `Per-scene normalization` - Each example is scaled according to a probabilistical maximum computed per training scene, and not for each example. The values are provided by the PBRT engine and are available for both training set and during typical usage.
- `Asymmetric normalization` - The input of the neural network is normalized on a per-frame basis, but the output of the network is compared to the ground truth normalized with a different scheme.

Both intensity and distance maps are converted from their unknown original magnitude to a relative magnitude by dividing each value by the mean of the map. Intensity values need to be handled in a non-linear way. Light intensity is perceived to increase only as the actual intensity value increases exponentially. Human vision is sensitive to luminance rather than energy in photons [55]. A logarithmic transform follows to reduce the high dynamic range peaks. *Kernel-predicting Convolutional Networks for Denoising Monte Carlo Renderings* (Section 2.5.3) showed that log transforms can significantly improve network performance. Interestingly, even existing models show significant improvement when used with logarithmic intensity data.

The *Batch normalization* [33] method is applied in the network to handle internal covariate shift and eliminate the necessity of dropout.

4.5.3 Loss Function

We considered several loss functions for our neural network, including the Relative Mean Squared Error (RelMSE) [58] [36] and a Relative L1 (RelL1).

$$E_{RelMSE} = \frac{n}{2} \sum_{q \in \{r,g,b\}} \frac{(p_{i,q} - c_{i,q})^2}{p_{i,q}^2 + \epsilon}$$

n is the number of samples per pixel, $p_{i,q}$ are the predicted pixels, and $c_{i,q}$ are the ground truth pixels.

$$E_{RelL1} = \sum_{q \in \{r,g,b\}} \frac{|p_{i,q} - c_{i,q}|}{|p_{i,q}| + \epsilon}$$

Both of these loss functions attempt to handle high dynamic range inputs gracefully. In our final model we use the much simpler L1 loss, which outperformed the other functions and works well with our normalized dataset.

4.5.4 Training Examples

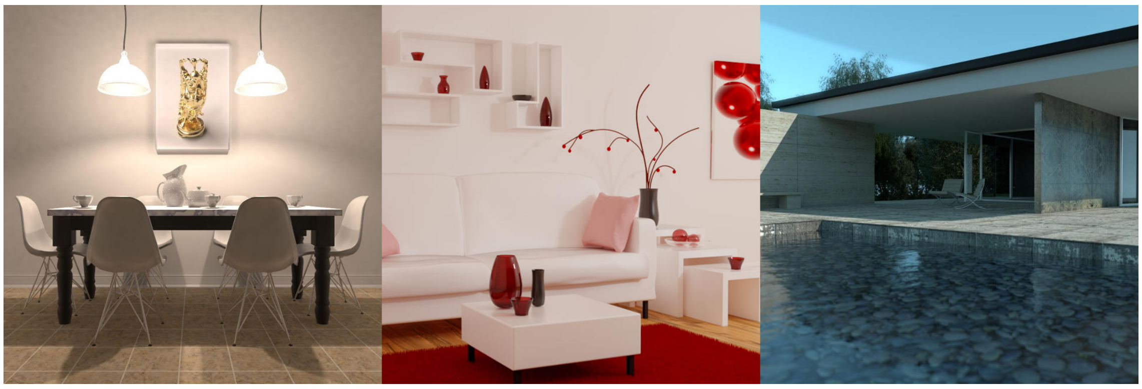


Figure 17: Some scenes part in the training set (cropped to fit)

A set of 43 scenes have been selected from the PBRTv3 example scenes (`http://pbrt.org/scenes-v3.html`), Benedikt Bitterli's resources (`https://benedikt-bitterli.me/resources/`), and our own custom created scenes. As OSR focuses on rendering indirect illumination, the selection of the training scenes has been weighted to include mostly interior scenes with difficult illumination, and scenes with strong global illumination.

Each scene has been processed to generate a set of hemispherical maps. Similarly to the approach taken by Kalantari [36], we change sampling algorithms and seeds to prevent the network from overfitting to specific noise patterns rather than higher level features: we use multiple samplers, including Sobol and Random, to produce varied noise patterns.

Over 16000 individual training examples has been collected.

4.5.5 Model

Our network architecture is shown in Figure 18 and is similar to the U-Net architecture [57], a convolutional autoencoder with skip connections and regular dimensionality progression. The encoder and decoder have symmetrical structure: each encoder stage uses two 3x3 convolutions and doubles the dimensional depth, while each decoder stage has two 3x3 deconvolutions and reduces the depth by half.

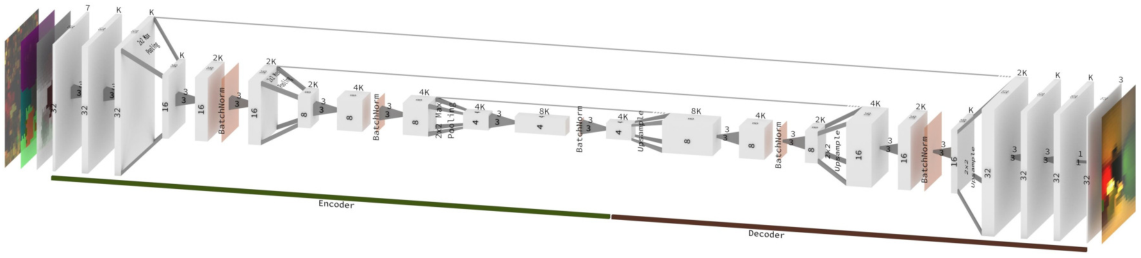


Figure 18: Network layout

All intermediate stages use batch normalization and LeakyReLU activation functions. The output stage has two 3x3 deconvolutions with LeakyReLU, and a final 1x1 convolution with ReLU activation to output the final data. Each downsampling stage uses a 2x2 Max Pooling, and the upsampling stages use 2x2 Bilinear Upsampling. We do not use any Dropout layer.

5 Implementation

We present the implementation details of OSR: the attempted Neural Network models in Section 5.1, the PBRT extension in Section 5.3.

5.1 Neural Network Training Attempts

We experimented different Neural Network models during the development of the project. We present here our attempts and observations.

1: Per scene normalization

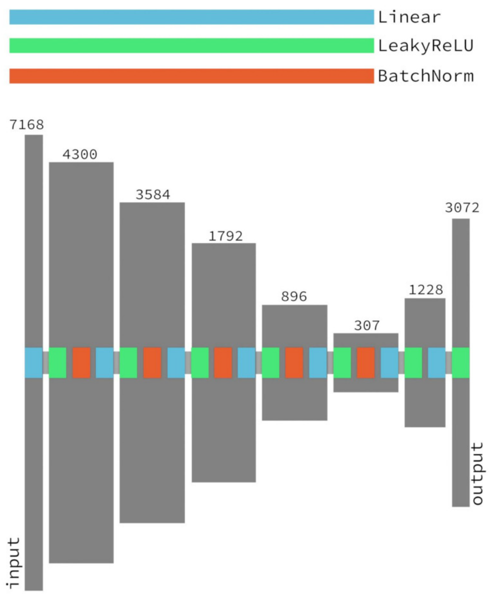


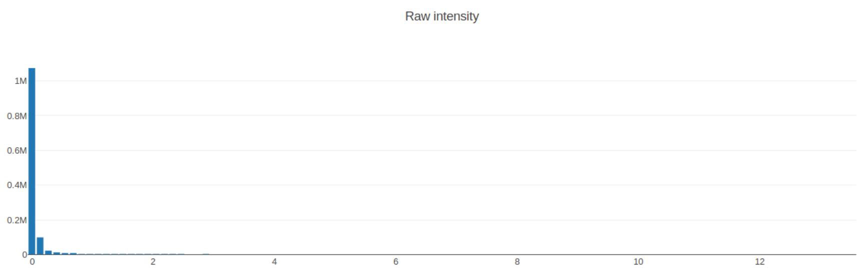
Figure 19: Network layout for Models 1, 2, 3

The initial model for the neural network is a deep, fully connected Multilayer Perceptron (MLP).

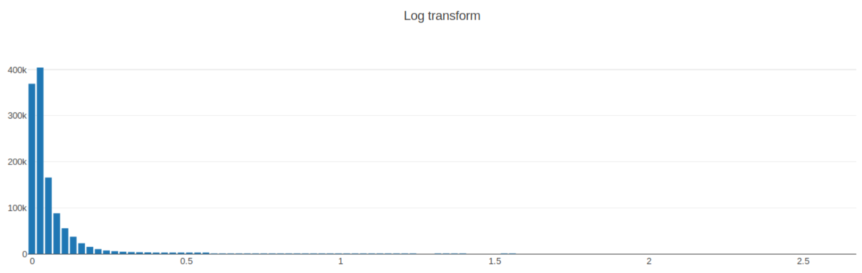
The network has 5 downscaling layers, and 2 upscaling layers, and processes a flattened array of the inputs. The 7168 input values are downscaled progressively to 307 values before being scaled up to the output image size of 3072.

Each of the layers uses a LeakyReLU activation function [50], which is similar to a Linear Rectifier, although it doesn't zero negative values but multiplies them by a small constant, chosen to be 0.2 in our case.

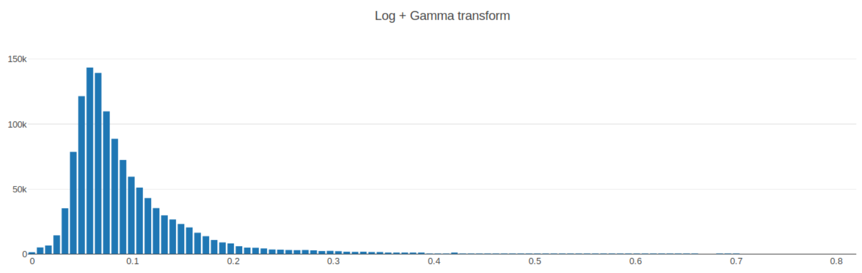
Our first approach uses per-scene normalization. For each scene, we estimate a single normalization value for each of our high dynamic range attributes: the single-pass path traced intensity map, and the distance map. Due to computational requirements, the estimation of the normalization values is performed only once per scene, and using a limited number of samples. We describe in detail our normalization estimation approach in Section 5.3.1. The normalization values are used with the *logarithmic* and *square root* transforms to obtain normalized maps for intensity and distances, which are used on the input layer of the neural network.



(a)



(b)



(c)

Figure 20: Plots of the pixel intensity distributions from the example scene `Breakfast` from PBRT. (a) Intensity values distribution of the raw intensity hemispherical maps pixels. Most of the pixels reside in a very small region of the total dynamic range. (b) Moving to a logarithmic scale, linearized with respect to the perceived brightness, improves the spread of the distribution significantly, although significant skew is still present. (c) Applying the Gamma correction significantly improves the distribution characteristic.

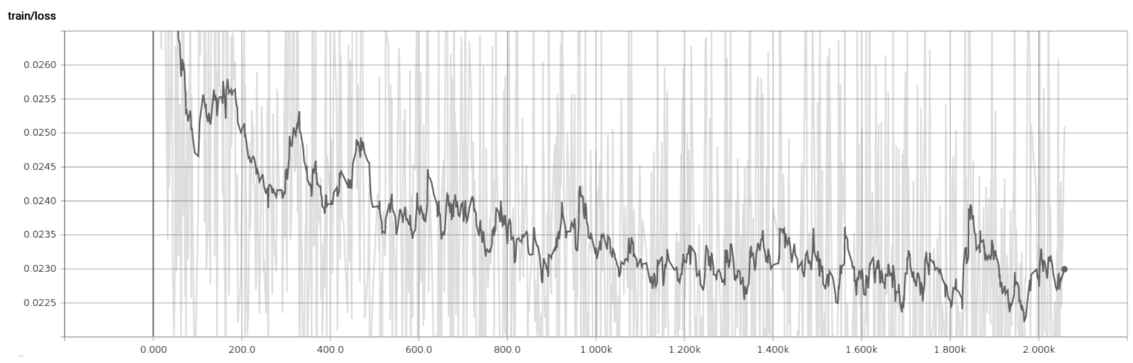


Figure 21: Model 1: Per scene normalization, batch normalization and RPROP optimizer

After applying the logarithmic transform, a histogram visualization of the data still shows that most of the data points fall within the lower end of the spectrum, with very few points on the higher side. Even when working with transformed data that presents linear perceived intensity response, the typical distribution of lighting in a scene is not well distributed. There are few points in most hemispherical maps being brightly illuminated.

We approach the issue by applying a further layer of transformation: a Gamma correction. Gamma correction is an important function, widely applied in image processing, displays and camera response calibration systems to handle the non-linear intensity response typical of many CRT, TFT and Plasma display technologies [48]. The Gamma function $f(x, gamma) = x^{1/gamma}$ applied to a dataset normalized to $[0, 1]$ produces a positive lift which is stronger in the darker regions. The lift boosts the output response of smaller values, and fits well the distribution of intensities of a typical hemispherical map used by OSR.

Figure 20 shows that the combination of a log transform and a gamma correction can significantly move the distribution of pixel intensities, allowing the network to achieve more stable learning.

To further take into account the high dynamic range of the inputs, we use *Batch Normalization* [33] in the first 5 layers of the network.

We set up the training loop to use the RPROP optimizer [56], which computes gradients based on the sign of the loss and discards the magnitude, with a learning rate of 0.0001. The loss function is the *L1*.

Figure 21 shows the loss during the first few thousands of iterations in training of this model. The model is highly unstable, presents very high loss values, and has low learning speed. A visual analysis of the test examples after 100 epochs of training shows very little relationship between the ground truth and the predicted images, except for a generic colored halo.

2: Per scene normalization, tweaking parameters

In an attempt to check whether we were working with an excessive learning rate, causing large instability in the loss values across different iterations, the second model uses the same network layout, based on the Multilayer Perceptron, but adjusts the learning rate to a more conservative 0.00005.

Figure 22 shows a comparison between this model and the previous one. The lower learning rate achieves better results during the first iterations, although the visual analysis of the longer training process at 100 epochs still reveals that the network is unable to return any usable result.

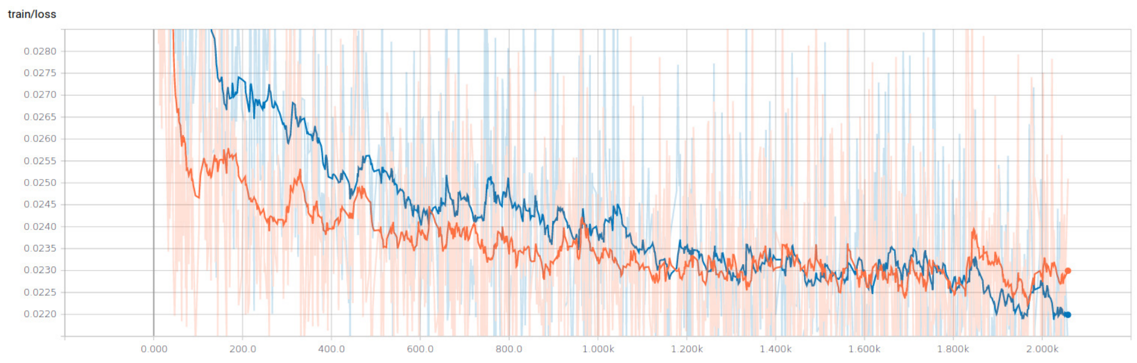


Figure 22: Orange: Model 1. Blue: Model 2: Per scene normalization, batch normalization and RPROP optimizer, lower learning rate.

There could be several reasons because the first two models were not successful. The network might not have enough learning power due to its relatively shallow depth and high compression ratio in the hidden layers, and the per-scene normalization system might not be appropriate for the input data. The estimated normalization values can be inaccurate due to the way they are sampled, and can cause in some cases large amounts of data to be clipped out of range.

3: Per Frame Mean and Standard Deviation Normalization

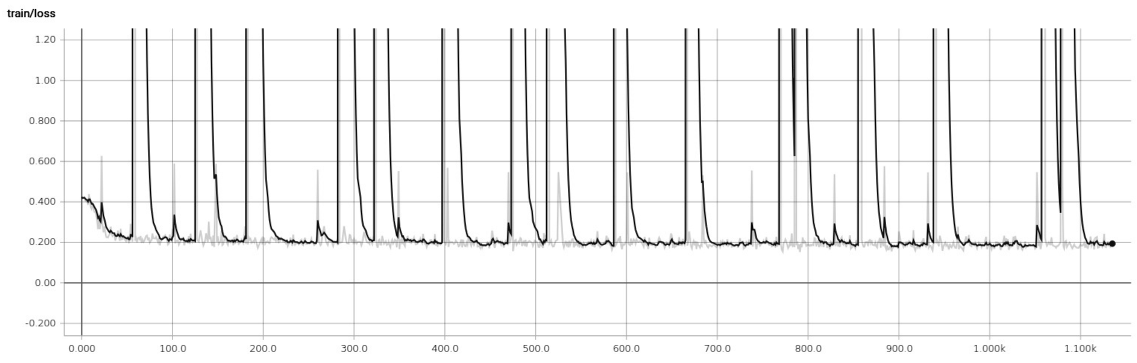


Figure 23: Model 3: Per frame normalization

In the third model we keep the same neural network layout based on fully-connected layers and batch normalization, and change the way we normalize the input data.

The input low quality intensity map is normalized by subtracting the mean and dividing by the standard deviation. This method, proposed by LeCun [47], is a popular general-purpose normalization mechanism.

To obtain meaningful comparison data from the ground truth images, we normalize the high-quality path traced intensity maps in the same way, and using the mean and standard deviation computed on the corresponding input intensity.

The distance maps are normalized in the same way as before, using a per-scene distance normalization value on the root of the distances, followed by a gamma correction curve.

Figure 23 shows the training loss of the third model during the first few iterations. Large spikes in the loss are clearly visible in the plot, and the network stops learning after a very small number of iterations.

In the longer training session the network fails to learn additional information, and the visual inspection shows mostly random images at the network's output, losing the blurred colored halos that models 1 and 2 had.

After we attempted training this model, we realized that the normalization mechanism was flawed. While it is viable to apply LeCun normalization on the input data, it was not a good idea to use the same mean and standard deviation to normalize the ground truth. In a path traced image, the average brightness of a pixel is unbiased: averaging pixel samples in a noisy render yields approximately the same average intensity compared to a higher-quality one. However, the standard deviation highly depends on the amount of noise that is present. An image generated by a path tracer at 1 sample per pixel in the presence of large amounts of indirect illumination usually contains a few very bright pixels over a dark background. Therefore the standard deviation would be typically very high compared to its high quality counterpart. Using the standard deviation from the input intensity map to normalize the ground truth causes the comparison image to heavily depend on the quality of the input. This is undesired behaviour, and causes the network to behave erratically.

4: Improved Per Frame Normalization

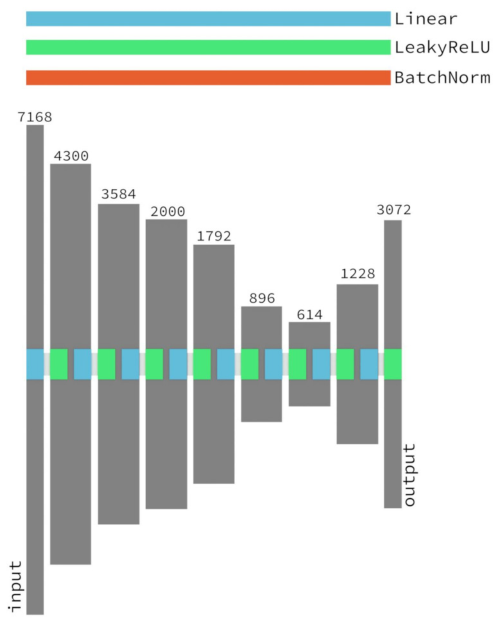


Figure 24: Network layout for Model 4

To verify the conclusion of Model 3, Model 4 applies independent normalization to the input intensity map and the ground truth intensity map. The idea is that these two intensity maps, although one the refined version of the other, do not need to be normalized in the same way. The neural network can be allowed to learn how to translate one normalized format into another. It is more important to use a robust normalization system that works independently of the target scene and its data values distribution. As long as it is possible to uniquely obtain a post-processed intensity map given the output of the neural network, any normalization methodology should work.

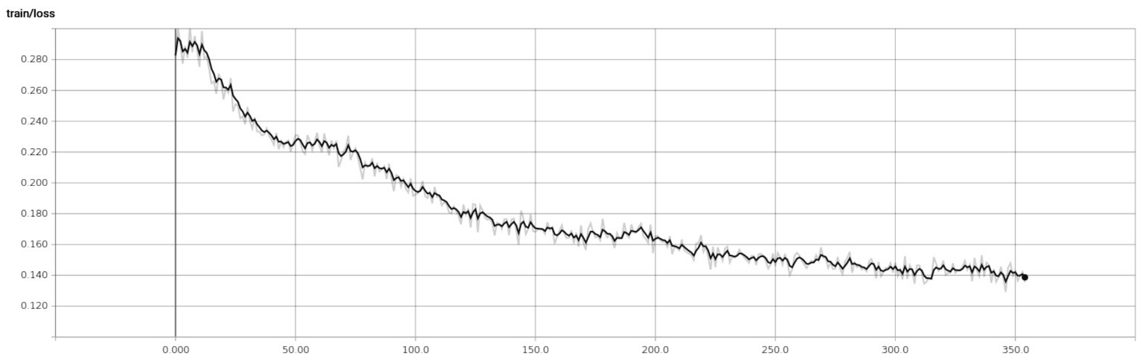


Figure 25: Model 4: independent per frame normalization

In this model we compute for each intensity map its average intensity value, although we expect them to be similar as path tracing yields unbiased results. Using the mean intensity, an intensity map's values are divided by $2 \cdot mean$ to obtain an average intensity of 0.5. Afterwards, a *log* transform is applied to linearize the illumination response.

The distance maps are now also handled on a per-frame basis, using a similar normalization scheme. The mean distance is used to scale the average value to 0.5, and a *square root* transform follows.

The model is similar to the previous one. There are 6 downscaling layers, including the input layer, and 2 upscaling layers, including the output layer. Each layer uses a LeakyReLU activation function with a factor of 0.2, and there are is no batch normalization.

Training is based on the *Adam* optimizer [40]. *Adam* is a stochastic optimization algorithm that maintains a dynamic, per-parameter learning rate capable of adapting to changes in gradient magnitudes over time. These properties make *Adam* a good choice for large parameter sets in the presence of noisy data.

Learning rate is set at 0.00005 and the loss function is L1.

Figure 25 shows the loss during the first few iterations of training. It is clear that compared to the previous models, Model 4 shows much higher learning stability, with all scenes contributing to the convergence of the network.

After 100 epochs of training, the test examples show the ability of the network to predict many cases with good accuracy, with several cases that are excessively blurry and a few outliers.

5: Improved Per Scene Normalization

In this model we use a deeper network structure with a lower compression ratio. There is a much higher number of learnable parameters, and we use the *Adam* optimizer with a learning rate of 0.0001. The loss function is L1.

The network uses 8 fully-connected compression layers with a LeakyReLU activation function, and 3 fully-connected upscaling layers.

The training examples have been normalized using the estimated per-scene normalization values, using *log* transform for intensities and *square root* transform for distances, followed by range compression and gamma correction.

Figure 27 shows the training loss over a long training session. The network is able to achieve

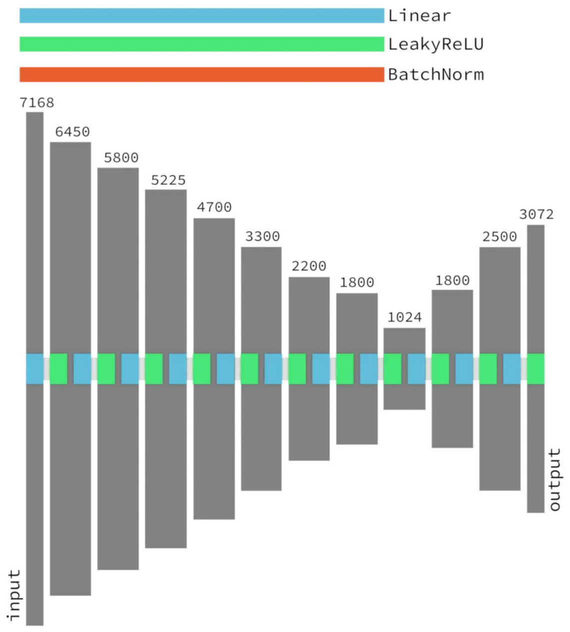


Figure 26: Network layout for Models 5 and 6

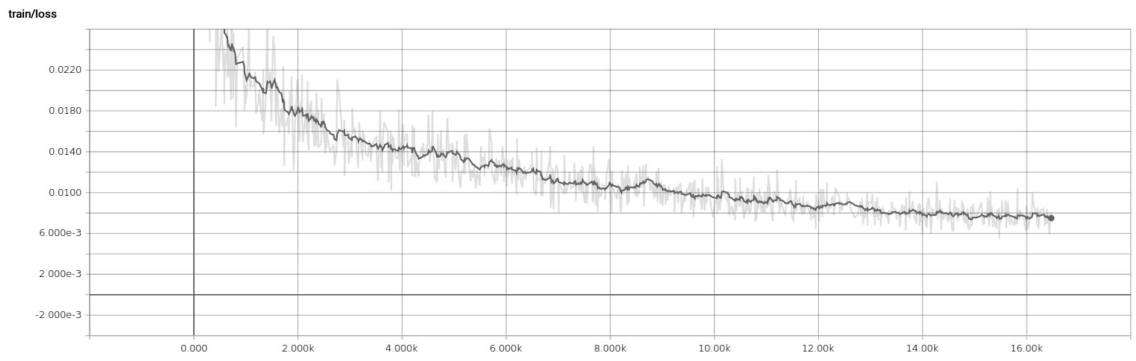


Figure 27: Model 5: New model with per-scene normalization

good progress over time. The test examples show good quality across the majority of scenes, with a few problematic outliers presenting highly noisy output or severe color shift.

6: Asymmetric Normalization

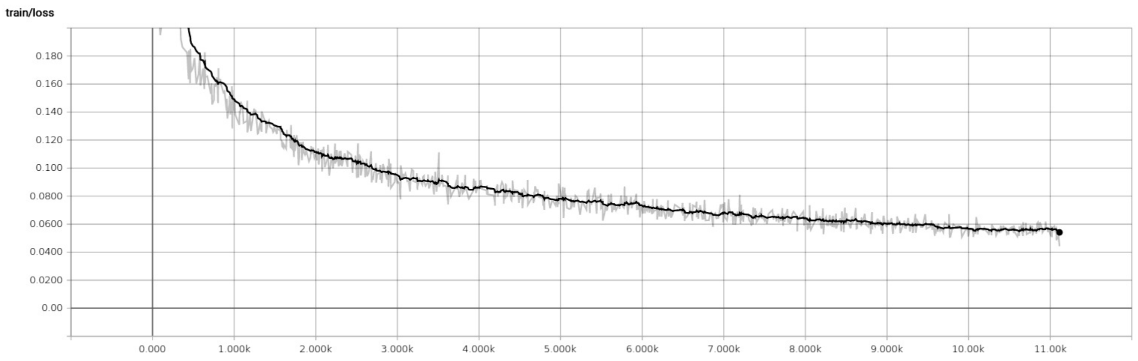


Figure 28: Model 6: Asymmetric transforms model training

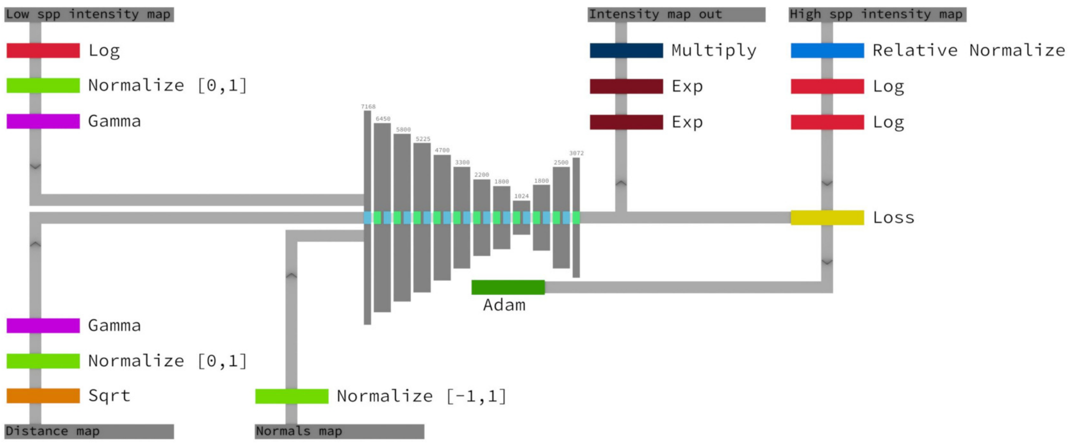


Figure 29: Detailed normalization and transformation steps for input and output data

In all previous attempts, the input intensity data and the ground truth maps were treated using the same normalization methods and transformations, although potentially with different parameters such as in Model 4.

In the asymmetric normalization model, we handle the two kinds of data in different ways, allowing the neural network to learn the non-linear relationship between the two. In Figure 29 two streams of data can be distinguished: on the left are the input intensity, distance and normal maps generated by the renderer, and on the right is the single ground truth intensity map part of the training suite.

The downstream transformation applies to the left side. The input data is normalized and transformed before it is used by the neural network.

The downstream transform for the low-quality intensity map uses a *log* transform, followed by a normalization pass that compresses the entire range to fit between 0 and 1, and a gamma correction. During the process, the mean value of the intensity is recorded.

The downstream transform for the distance map uses a similar strategy: a *square root* transform is used, followed by a range normalization and a gamma correction curve. The mean value is not recorded as it is not needed.

On the right side, at the output of the network, the prediction is compared with a processed version of the ground truth. The ground truth intensity map is divided by its own mean to convert the values from an absolute brightness scale to a relative one, and transformed twice using a *log* transform to reduce its dynamic range and boost the importance of the darker values, similarly to what is achieved with the gamma correction.

Due to the asymmetry in the transforms used on the two sides, a third pipeline is necessary for the upstream component on the right side. The upstream pipeline serves during normal usage to obtain the output image at its full dynamic range and scale from the output of the network. Each stage of the right downstream transform is reversed: two *exponentiation* transforms are used, followed by a multiplication by the mean intensity of the input intensity map that was recorded on the left downstream branch. The final pass restores the original absolute intensity scale and assumes that the input intensity is unbiased.

The network design is similar to Model 5: 8 and 3 fully connected layers with Leaky ReLU activation functions on downscaling and upscaling sections respectively. The optimizer is Adam, with a learning rate of 0.0001. The loss function is L1.

Figure 28 shows the training loss of the model during the first 11k iterations. The network has a smooth learning curve, and although presents higher loss values compared to Model 5 due to the different output data representation, it demonstrated better learning capability. The visual inspection of most scenes reveals good predictions, with occasional over blurriness but no heavily problematic examples in which the result was clearly wrong.

7: Refining the model

Model 7 is built on top of Model 6 by refining the data normalization procedure and by changing some of the activation functions.

The normalization scheme of Model 6 produced images with relative-scale intensities centered around an average value of 1. To move the average closer to 0 and make the distribution better focused within the unit range, we use a target mean value of 0.1.

The model has been updated by changing some of the Leaky ReLU activation functions with TanH functions.

While we previously used 10% of the training examples for testing, starting from Model 7 we used over 800 additional testing examples from 2 scenes that are not part of the training.

Figure 31 shows the loss on the training examples, and 32 the loss on the additional test examples extracted from the 2 new scenes.

The network progressively learns on both curves, although it's clearly noticeable that the loss values on the test examples are significantly higher, and that the curve remains mostly stationary while the loss on the training examples continues to fall. Even though the network is not yet

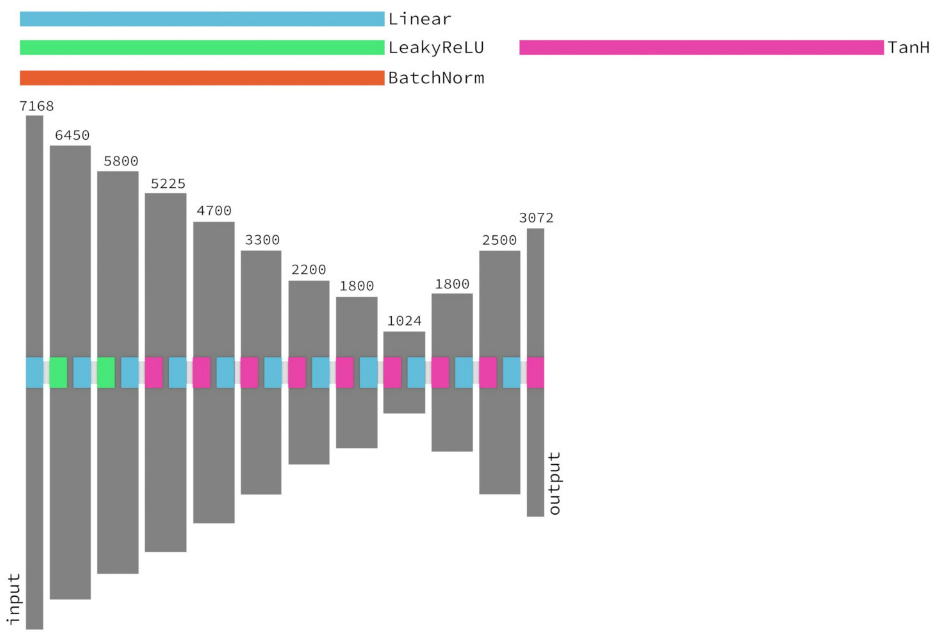


Figure 30: Network layout for Model 7

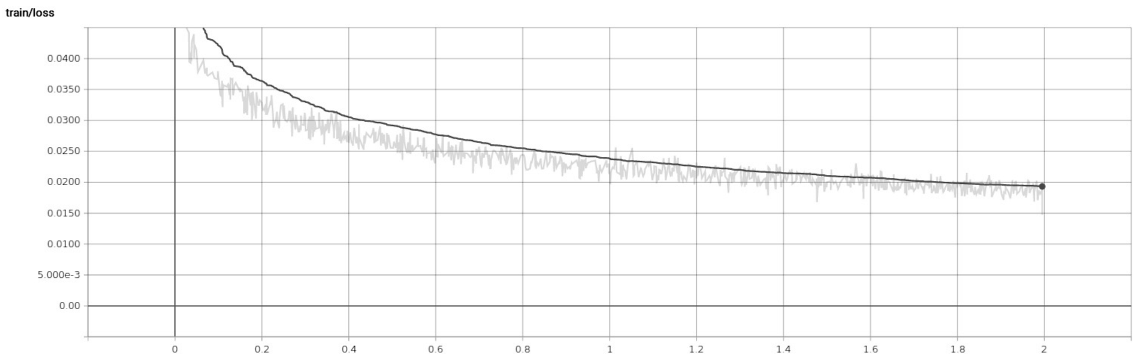


Figure 31: Training Loss in Model 7

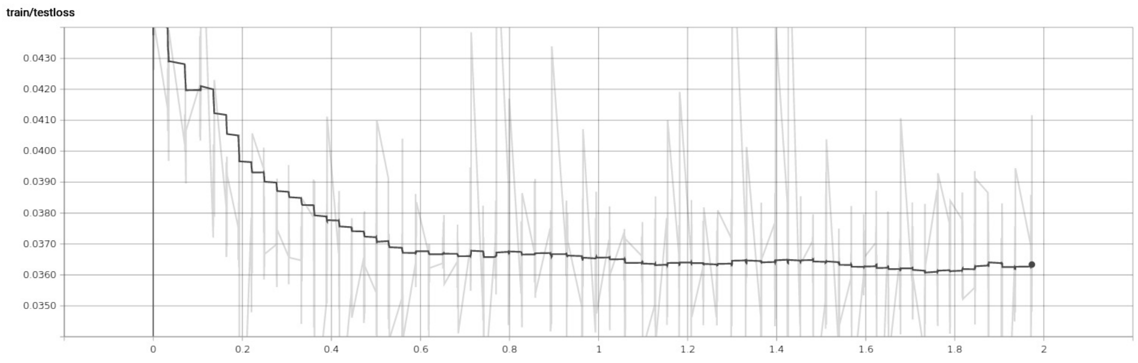


Figure 32: Test Loss in Model 7

overfitting, it is struggling to learn attributes that can be generalized to totally new scenes, although it can perform well on scenes that have been seen.

A visual analysis of the test examples reveals very blurry network output and strong color bias, with many features being completely missed.

8: Camera Space Normals

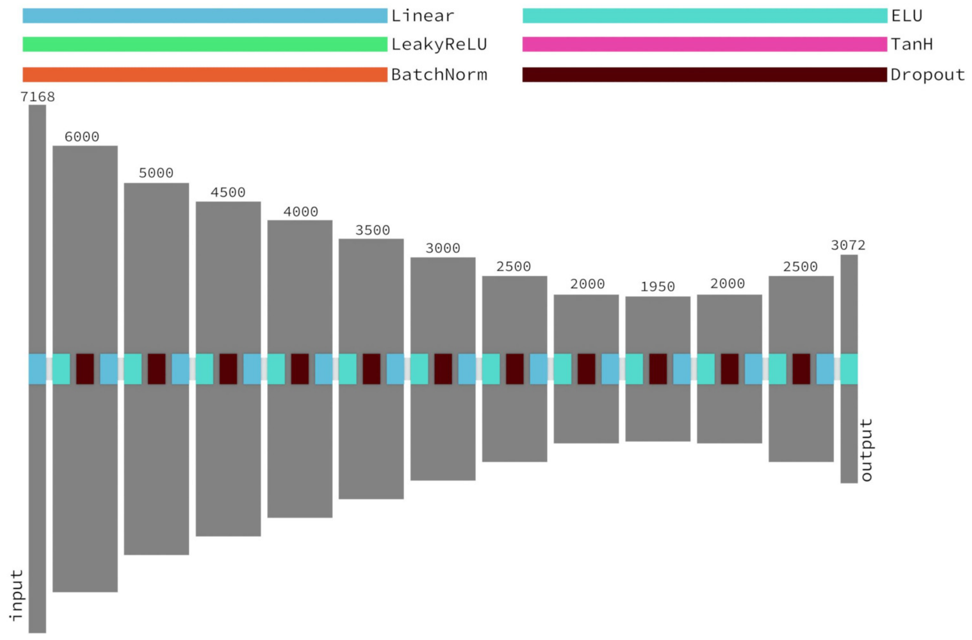


Figure 33: Network layout for Model 8

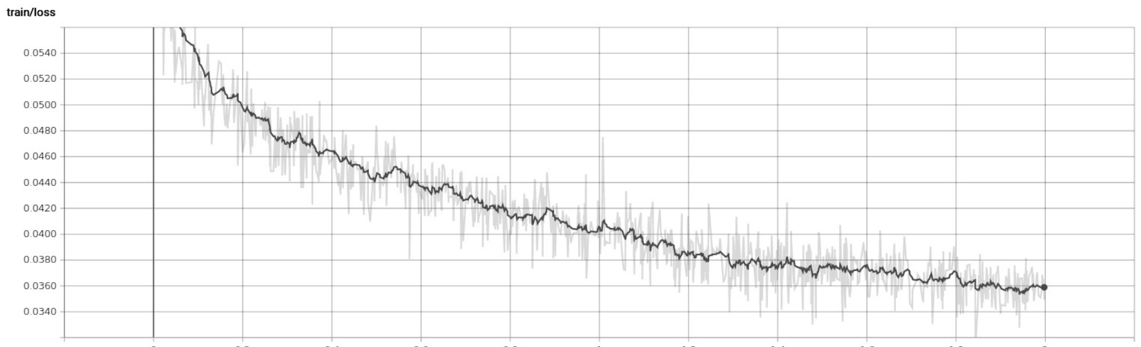


Figure 34: Training Loss in Model 8

To reduce the likelihood of the network to overfit to training scenes, we changed the normals map representation from world space to camera space (see Section 5.3.2), and introduce dropout in some layers.

Model 8 introduces the Exponential Linear Unit (ELU) [41] activation function. The shape is similar to a LeakyReLU, and it presents a smooth curve and a mean output value closer to 0.

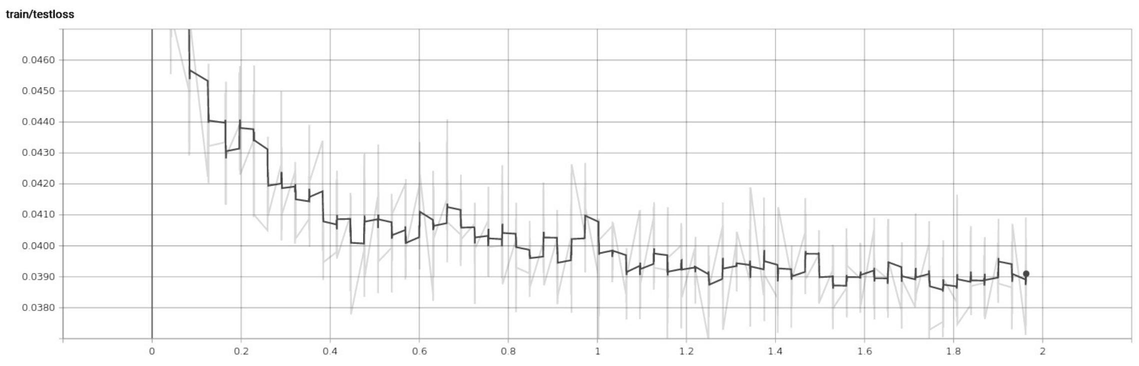


Figure 35: Test Loss in Model 8

A dropout layer with a factor of 0.2 has been applied to each hidden layer.

Figures 34 and 35 show the training and testing loss of Model 8 respectively. The introduction of the dropout layers reduces performance significantly, while it can arguably keep the loss of the test examples closer to the expected value.

The outputs however are still disappointing and would not be appropriate for use on a generic scene.

Model 8 is our last model based on fully connected layers.

9: Convolutional Autoencoder

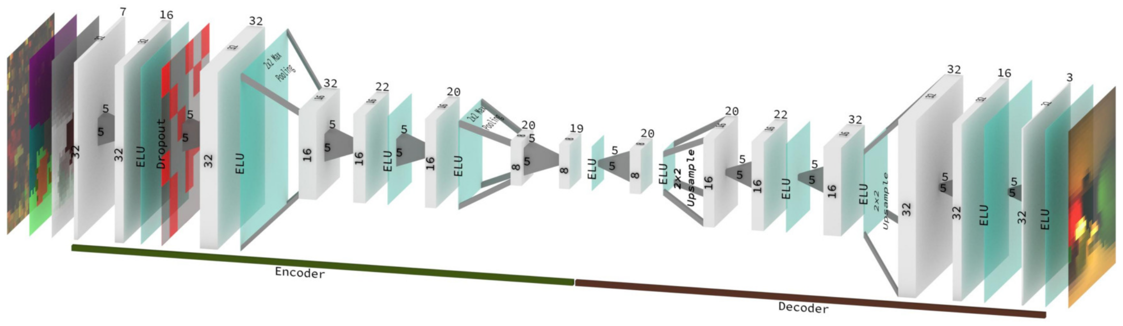


Figure 36: Network layout for Model 9

Model 9 moves away from fully connected layers towards convolutional autoencoders.

A Convolutional Neural Network [46] has the ability to process structured 2D data by learning

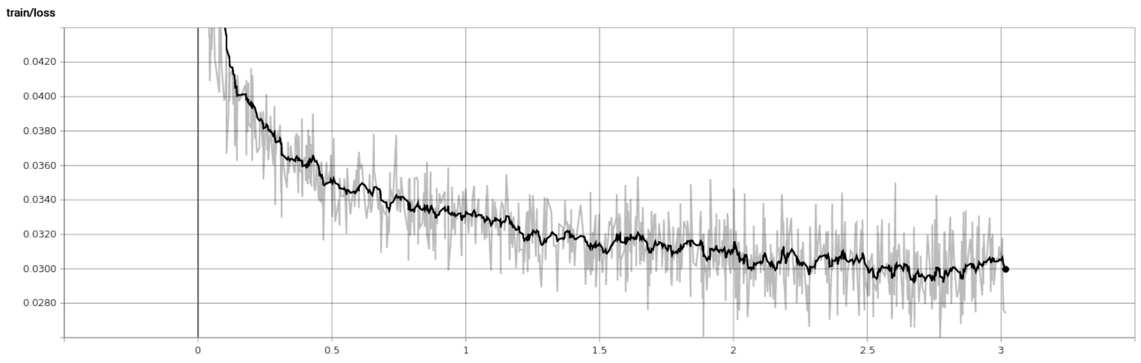


Figure 37: Training Loss in Model 9

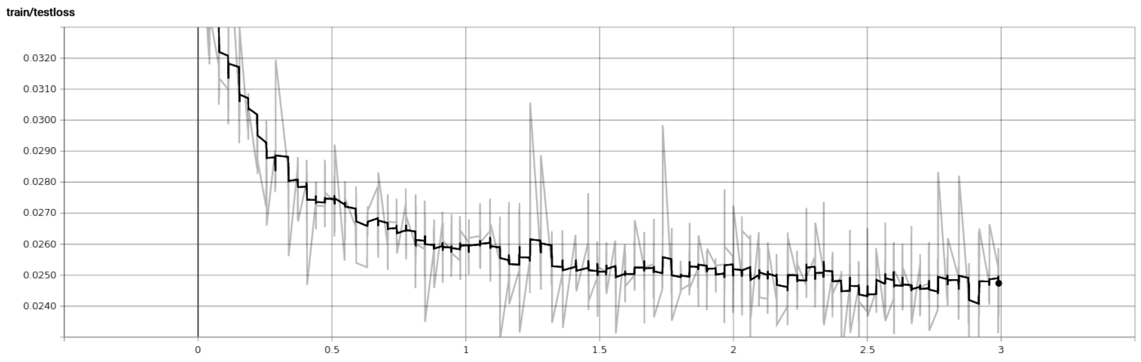


Figure 38: Test Loss in Model 9

optimal convolution functions in order to extract features from the inputs. Stacking layers of convolutions and non-linear activation functions is a powerful solution to process images with arbitrary functions.

This model is based on the convolutional autoencoder layout used in *Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder* [29]. The 32x32 input layers are compressed down to an 8x8 representation by the encoder, and the decoder upsamples the data in a symmetrical way to the output format. The activation functions used after each convolutional layer are Exponential Linear Units (ELU) [41], and there is a single dropout layer in the encoder with a factor of 0.1 to reduce the possibilities of overfitting.

The input consists of 7 layers: intensity RGB, normals XYZ, and distance. The output only has the 3 channels for intensity.

Figures 37 and 38 show that Model 9 has good learning progress, with both training and test loss values changing at the same rate.

A visual analysis of the predicted examples from the unseen test scenes shows much better results compared to the fully connected models, although a significant amount of blurriness remains in many cases.

10: Convolutional Autoencoder with Skip Connections
Further following the model presented in *Interactive reconstruction of Monte Carlo image sequences using a recurrent denoising autoencoder* [29], Model 10 extends the previous with small changes and the introduction of skip connections between some of the layers. The reasoning behind the use of the skip connections is to allow the network to reconstruct a smooth output from its

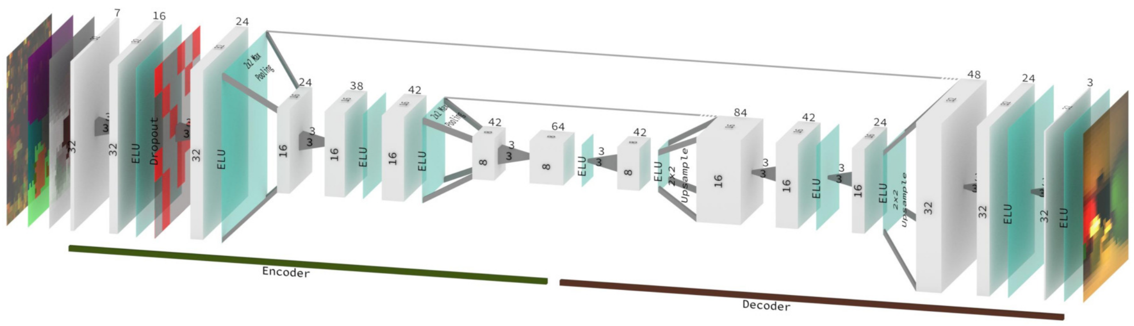


Figure 39: Network layout for Model 10

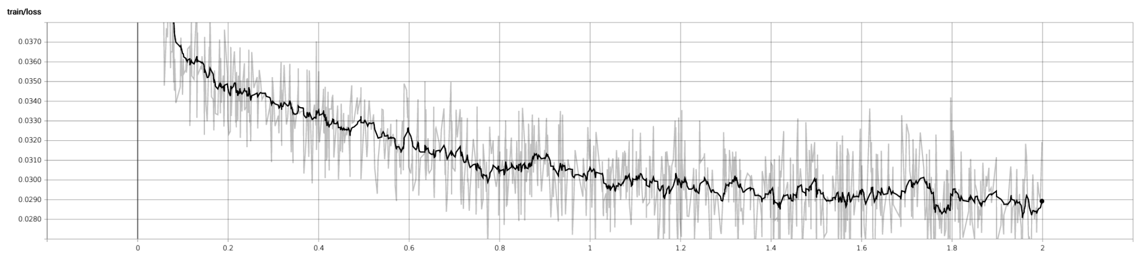


Figure 40: Training Loss in Model 10

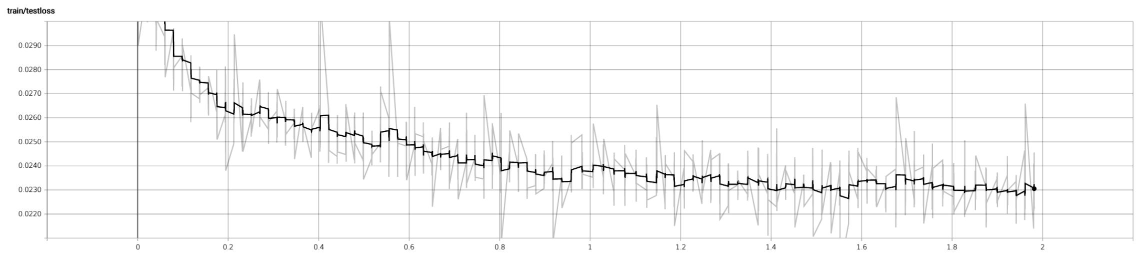


Figure 41: Test Loss in Model 10

internal representation while still retaining the finer details obtained from the corresponding encoder layer.

The 5x5 convolutions and deconvolutions present in Model 9 have been updated to use smaller 3x3 convolutions, more adequate for our application where the size of the input images is small.

In Figures 40 and 41 Model 10 shows similar learning progress compared to Model 9, with slightly lower overall loss values thanks to the ability of the network to better retain finer details.

The output of the network shows good results across all training and test examples.

11: Modified U-Net

The final model is similar to the U-Net [57], a convolutional autoencoder with skip connections and regular dimensionality progression. We discuss in more detail the structure of this model in Section 4.5.5, and its results in Section 6.1.

5.2 Model training

We implemented the neural network model and training process in Python using Pytorch (`http://pytorch.org/`). All machine learning processing has been conducted on a single machine with an Intel i7 4770 processor, 16GB of memory, and an Nvidia GTX 1060 6GB card for GPU acceleration.

We chose a base layer size of K=64, and a minimum hidden layer dimension of 4x4. These parameters result in 3 downsizing steps and 3 corresponding upsampling steps.

We used the Adam optimizer included in PyTorch, with a learning rate of $6e-5$. We achieved good network performance after 20 minutes of training with mini-batches containing 32 examples each. Our training covers 3 epochs over the augmented dataset: more than half a million individual examples.

We implemented data augmentation in our Pytorch data loader to include all combinations of rotation and flipping.

5.3 PBRT Extension

The OSR method and algorithm has been implemented by modifying PBRTv3, a C++ ray tracer implementing many popular algorithms, such as Path, Bi-directional path and Photon Mapping, written with a strong educational purpose.

In this section we discuss the implementation details of the PBRT-OSR extension. Starting from PBRTv3, we implement a new `integrator` which evaluates direct and indirect illumination separately, and communicates to the neural network to obtain the radiance map predictions.

5.3.1 Normalization Estimation

The Normalization Estimation process computes the normalization range values used by the neural network in the Per-scene normalization mechanism (Section 4.5.2). Although the normalization estimation mechanism is not used in the final version of PBRT-OSR, we discuss here its original implementation.

The Normalization Estimation pass is executed as a preprocessing step on the entire scene, and aims at obtaining a measure of the typical dynamic range present. Both intensity and distance values are obtained.

It would not be possible to compute a highly accurate normalization value in a reasonable amount of time. Being a preprocessing step that does not generate any additional useful samples or data, it needs to be cheap compared to the rest of the computation. Therefore OSR uses a limited number of samples, set by default to 10000, to obtain an estimate of the dynamic range of the scene.

For each of the samples, a random path is shot into the scene from the main camera, and the maximum intensity and distance obtained are recorded. The intensity value is computed using path tracing, following random bounces until the Russian Roulette (Section 2.4.5) terminates the path or no intersection is found. The distance is the world distance between the first found intersection and the camera's position, or -1 in case no intersection is found. The final normalization values are

$$log(maxIntensity)$$

and

$$sqrt(maxDistance)$$

5.3.2 Normal Map Rendering

Both during normal rendering and reference generation (Section 5.3.3) a normal map is required. The auxiliary camera's first intersection for each ray is used to determine a sample of the surface normal.

When the camera ray does not find any intersection, the null normal value of $(0, 0, 0)$ is recorded in the normal map.

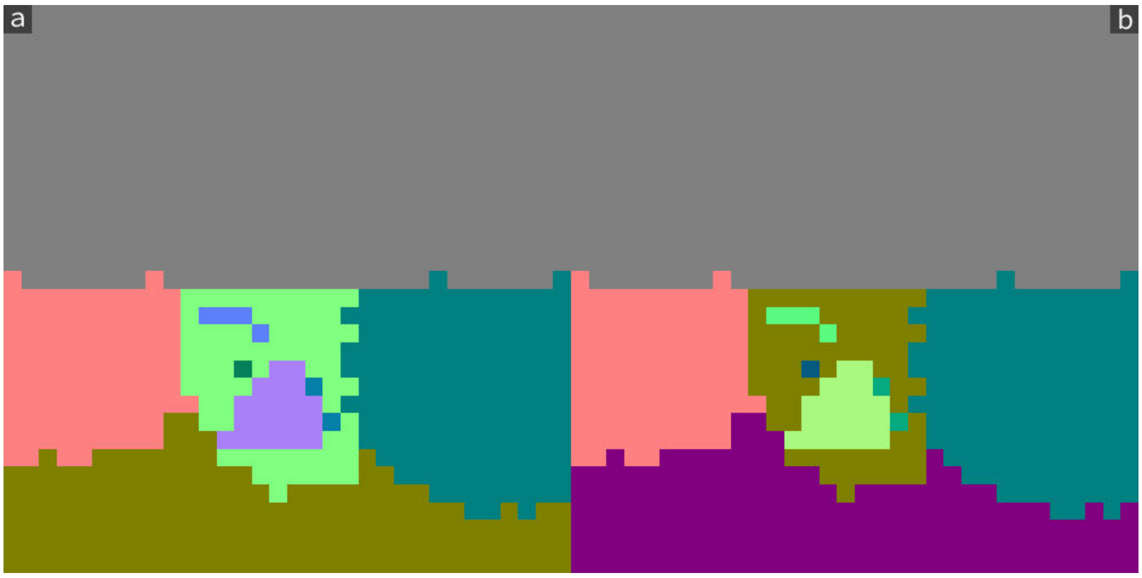


Figure 42: An example of normals rotation. a) World space normal map. b) Camera space normal map

When an intersection is found, the surface normal is recorded after it has been transformed into camera coordinates. The transformation is aimed at reducing the possibility of overfitting the network to specific scene geometry, as many interior scenes have regular and flat surfaces across large surfaces all oriented in the same world direction. Using a camera-relative normal increases the variation of the data, and helps the network to learn properties that are better represented in a local form, similarly to the way surface position points are recorded as distances from the auxiliary camera.

Figure 42 shows an example of a normal map before and after the rotation into camera coordinates.

5.3.3 Reference Generation

PBRT-OSR can operate in two different modes: *Reference* and *Rendering*.

While the *Rendering* mode can be used by the final user to render a scene, the *Reference* mode computes training examples given a scene.

Each training example consists of one-shot, distance, normals and high quality radiance maps. The Reference mode automatically generates a large number of examples suitable for both training and validation purposes.

To enable Reference mode, the user can provide the PBRT executable the flag `--referece=X` where X is the number of tiles per side to be used. The number of tiles determines the way the primary image film is subdivided into a regular grid. Each intersection in the grid is used to shoot a single camera ray into the scene, and the intersection found becomes the viewpoint of one set of examples.

The process loops for all tiles generated, and for each computes the four maps part of each training exampls.

The high quality path tracing render's number of samples can be specified from the command line by the user.

5.3.4 OSR Rendering

In the rendering process we use two independent steps to compute indirect illumination and direct illumination.

The direct illumination pass is handled using a standard *Direct Lighting* integrator implemented in the original PBRT software.

The indirect illumination uses the *Path* integrator to collect the neural network's input maps. The machine learning evaluation runs in a separate Python process running Pytorch. Each rendering thread has its own child process, and communication is handled through standard Unix pipes. Each Python evaluation process loads the saved model parameters, and reads flattened input data from `stdin`. The predicted output is written back to `stdout`.

During the rendering process the PBRT C++ process manages the full data transformation pipeline shown in Figure 29. The upstream tranformation component uses a slightly different strategy as it does not multiply its final intensity by the expected mean computed from the one-shot intensity map. To ensure that output map and input intensity have the same overall luminous intensity, the output of the network is multiplied by the target mean brightness and divided by its own brightness. This strategy eliminates any brightness bias caused by imprecision in the predictions.

6 Evaluation

In this section we evaluate the results of the project from both a quality and a performance point of view. We first evaluate the Neural Network in isolation in Section 6.1, perform an Ablation Study in Section 6.2, evaluate the rendering output quality in Section 6.3, the interpolation strategy in Section 6.4.

6.1 Neural Network Evaluation

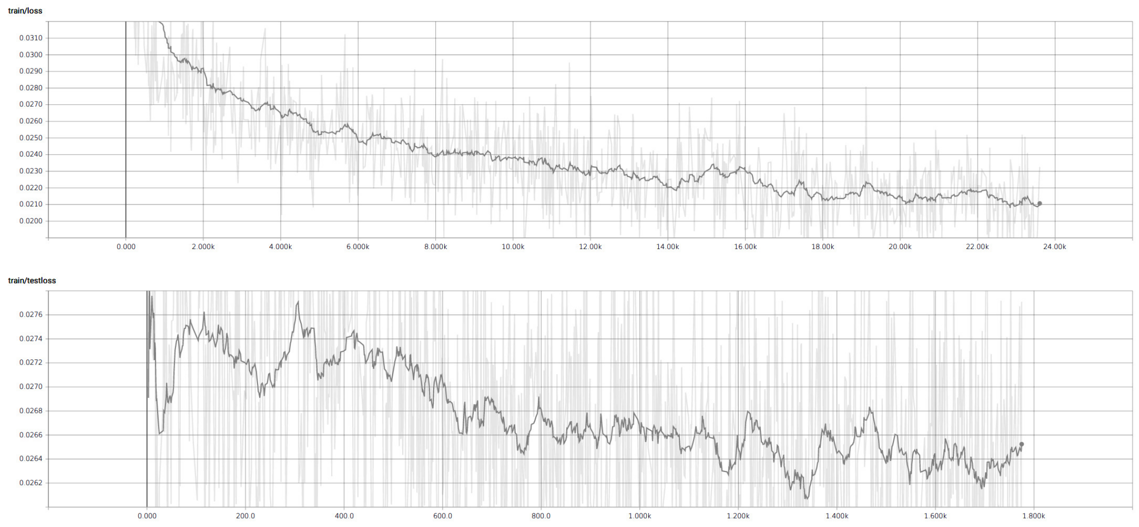


Figure 43: Training and Test L1 Loss in Model 11

Figure 43 shows training and test loss values during training. Our model achieves good learning progress over a 20 minutes learning period, without overfitting to the training dataset.

The network was trained on a collection of 3D scenes, and the evaluation is conducted on three selected scenes that were never part of the training dataset.

The scene *White Room Daytime* is part of the original PBRTv3 example scenes [20]. It is an interior illuminated by three large windows. This scene is a good representation of a typical architectural visualization, with a good balance between directly illuminated surfaces and global illumination contribution, and a good level of overall detail. A direct lighting only pass leaves large regions black where they face directions that have no light sources, and produces an innatural contrast.

The second scene, *Veach Ajar* by Veach and adapted by B. Bitterli [19], is well known for its extreme rendering difficulty. The only light source is located behind the almost closed door, and most of the visible light is indirect. Path tracing and even bi-directional path tracing have significant difficulty at obtaining a noise-free result, which makes it hard for us to obtain a ground truth with an acceptable level of quality. While this is clearly an edge case scenario in global illumination, we show how our neural network performs under worst-case conditions.

The last scene, *Mbed1*, was custom created for the purpose of this evaluation, and uses simpler, more geometrical models. There is a single light source that illuminates the room from the left side, and some strong color contrasts. This scene is not particularly difficult for traditional algorithms, and is a good evaluation to show any undesired overhead in OSR when dealing with simpler cases.

The difference in path tracing sampling difficulty across the test scenes allows us to verify that our network is able to deal with different input data quality seamlessly, and that it is able to extrapolate and adapt to geometry not encountered during training.

As with the training set generation, we obtain input maps and ground truth path traced radiance maps for several viewpoints located at primary surface intersection points. While the PBRT-OSR implementation re-normalizes the output images to match the expected intensity level of the entire radiance map, in our tests we compare directly using the final intensities obtained after the upstream processing to provide an unbiased view of the network performance.

Figure 44 shows a random selection of radiance maps predicted using our neural network model. The first three columns show the inputs of the network, and the result is compared to a simple gaussian blur and a reference path traced ground truth rendered at 1024 samples per pixel.

The gaussian blur uses a 1 standard deviation filter width, chosen for its good compromise between smoothing power and blurriness. A smaller radius would not be able to remove much noise, while a larger radius would yield considerably blurrier results compared to either ground truth or network predicted output. Due to the very sparse samples generated by the path tracer, a simple gaussian blur is a significant improvement over the 1spp (one-shot) intensity map. Sampling the BRDF of glossy surfaces using the blurred version would yield considerably lower noise due to the higher probability of hitting a non-black pixel on the hemisphere and due to the lower overall variance in intensity.

The examples show that the network predicted output can consistently outperform the gaussian blur by producing sharper images, preserving more detail, and removing more noise. We can observe that:

- The predicted result is generally sharper than the gaussian blur. This confirms the fact that we would not be able to use a larger radius for the gaussian blur for difficult cases without losing even more details.
- The network is able to reconstruct some details that are missing from the one-shot intensity map, although it can cause artifacts.
- The network has effectively learned to use adaptive blurring filters. In simple cases it behaves very similarly to a planar gaussian blur, but in the general case it is able to adapt to different sampling densities more effectively.
- The network eliminates all visible noise, and can produce images that look smoother and with less noise even when compared to the high quality path traced reference images. This shows the ability of the network to extrapolate from the training dataset, which still contains a small amount of noise in its ground truth examples.

To verify our claims, we run metrics on a large number of radiance evaluation points collected from the testing scenes, totalling over 1000 distinct testing examples that were not seen during training.

For each example set of images, we compute the L1 absolute difference from the ground truth, shown in Figure 45 and the Structural Similarity metric [69], show in Figure 46.

The Gaussian blur does considerably increase the quality of the radiance maps compared to using raw 1spp renders. The structural similarity increases, as the blur fills many of the black gaps present in the path traced map.

The network output considerably outperforms the Gaussian blur in both L1 and structural similarity metrics. We confirm that for both metrics the samples of Gaussian and predicted belong to different distributions. We compute Kruskal-Wallis null hypothesis testing [44] p values between the Gaussian and predicted metric values for L1 and Structural Similarity to

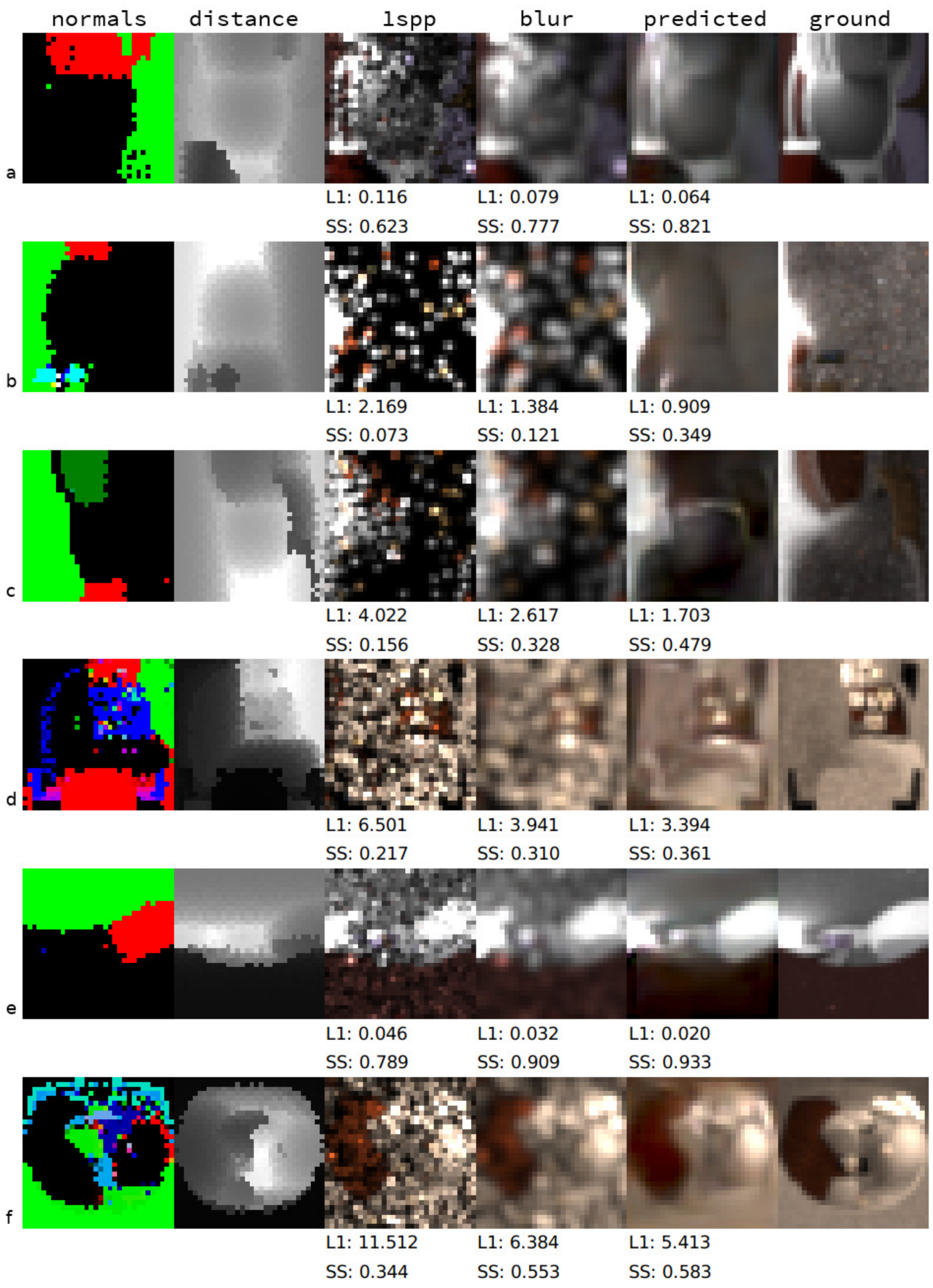


Figure 44: A random selection from the test scenes *Veach Ajar*, *Custom Mbed*, *White Room Daytime*. All examples show excellent noise removal, occasionally generating less noise than the reference ground truth. In a) there is some color inaccuracy, with the predicted image's purple being less saturated than the ground truth. b) and c), part of *Veach Ajar*, are very difficult due to the very sparse sampling of the input intensity map. They show excellent noise removal, producing less noise than the ground truth, although c) shows some shape distortions. d) and f) are less sharp than the ground truth, but still considerably more usable than the gaussian blur versions.

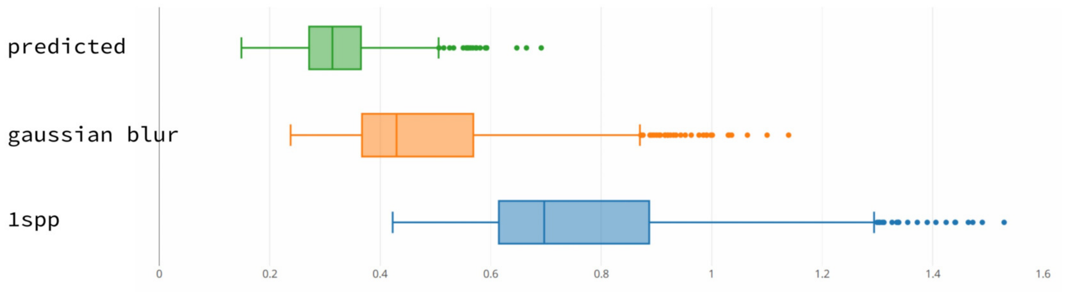


Figure 45: L1 absolute difference

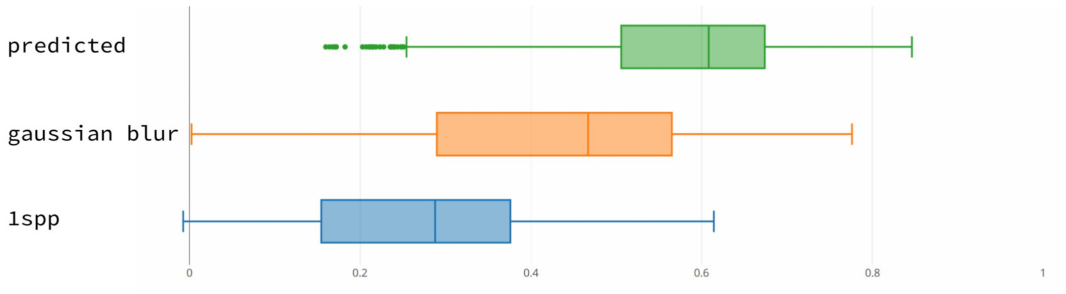


Figure 46: Structural Similarity

verify that they belong to different populations, and obtain values less than 0.00001.

6.2 Neural Network Ablation Testing

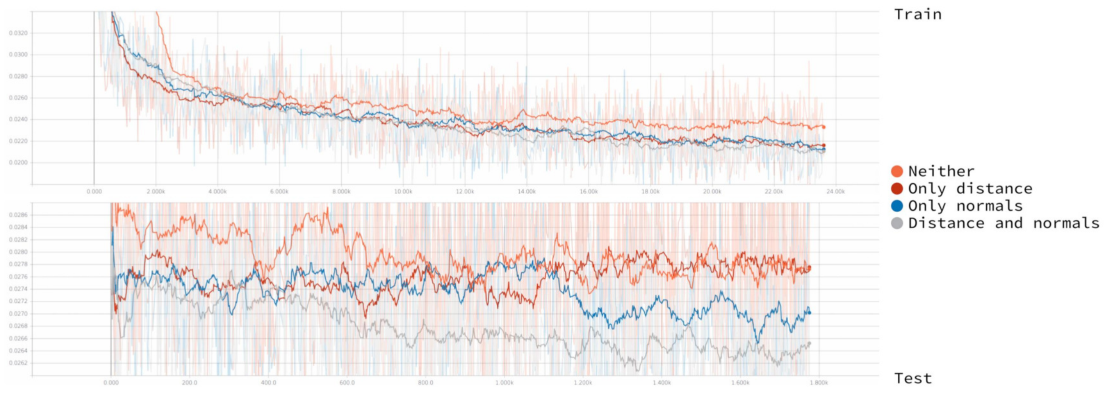


Figure 47

The ablation test aims at verifying some hypothesis about the data that the neural network uses to produce accurate predictions. Training is conducted using the entire dataset of intensity, normals and distance maps, and repeated with the normals, distance, or both components removed from the network's inputs. The structure of the network is left unchanged, and the ablation is implemented by setting the target components to be zero arrays.

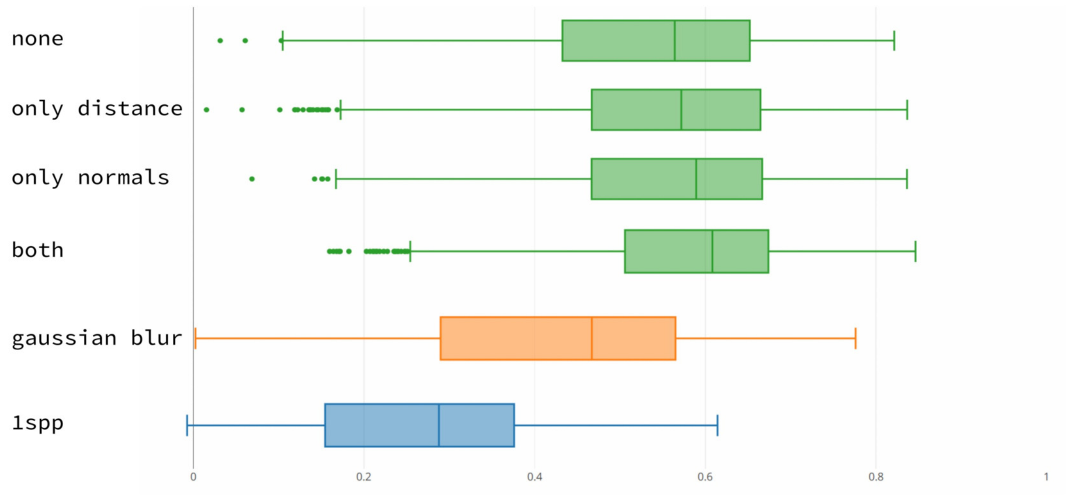


Figure 48

The evaluation mode is conducted in a similar way, by setting the ablated channels to zero arrays.

The expected result from this experiment is that the network performs better the more information it has, and that the addition of normals and distance maps is valuable for the quality of the predictions.

Figure 47 shows loss values during training and testing using the four different configurations.

Removing both normals and distance information and using only the one-shot intensity map as the input to the neural network yields the worst results. This is expected as the low quality path traced image contains very little information. While the network can still achieve better results than a gaussian blur by adapting to different levels of input sampling density, it remains an image-level blur filter.

Adding distance information significantly improves performance during training, although the testing set's loss values remain very similar to the configuration using only low quality intensity maps.

Using only normals in addition to the low quality intensity map achieves significantly better results in the test set. We observe that the addition of normals brings more useful information to the network than the addition of distances. Normals encode three-dimensional slopes, and can be useful to detect sharp edges and the direction in which surfaces are facing.

Interestingly, while the addition of distance maps alone did not help the test considerably, using all three input maps brings a significant boost in performance compared to having normals only. We conclude that the combination of normals and distances gives a more complete representation of the 3D structure of the scene.

The Structural Similarity Index test on the entire test suite in Figure 48 confirms that the addition of normal or distance maps help the network predicting more accurate results, with the normals being slightly more effective than distances alone. The combination of both additional layers further increases accuracy, and in particular improves the lower quartile significantly.

6.3 Final rendering evaluation

The evaluation of the final rendered images is based on both qualitative and quantitative metrics. We render each of our test scenes at different levels of quality, and present observations, statistics and comparisons with other methods and rendering algorithms.

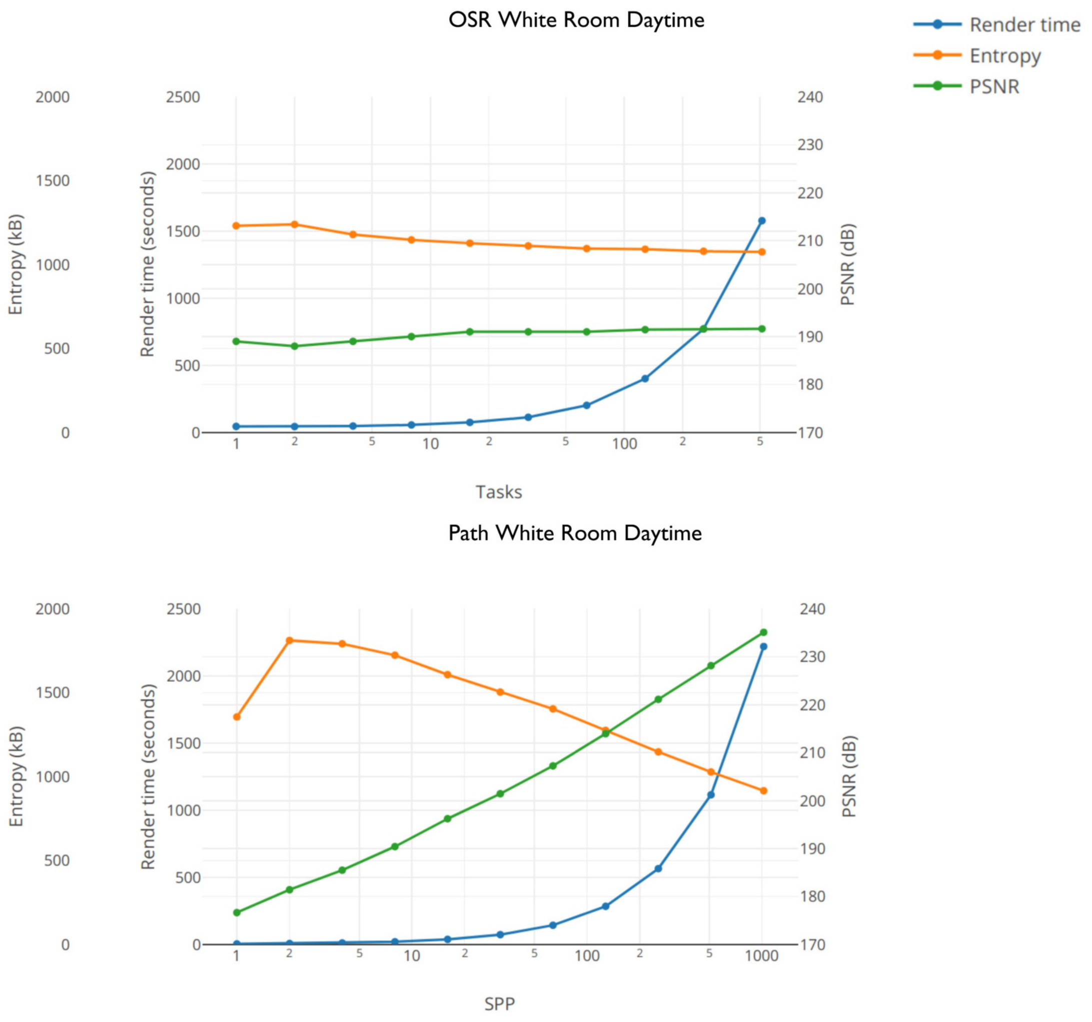


Figure 49: Quality and noise metrics on White Room Daytime. Above: OSR, Below: Path tracing. *Render time*: total rendering time in seconds. *Entropy*: indication of the amount of noise, measured as the final image size after PNG compression. *PSNR*: Peak Signal to Noise Ratio.

We first analyze images of the scene *White Room Daytime*. We use OSR and Path tracer integrators at increasing levels of quality in order to pick reasonable sampling levels for both algorithms for the comparison.

Figure 49 shows OSR and Path tracer plots of the scene rendered at different levels of quality. The Path tracer uses power of two sampling rates, from 1 to 1024 samples per pixel. A reference image is also generated using Path tracing at 2048 samples per pixel. OSR is not based on a per-pixel quality setting, but on two distinct settings: the number of samples per pixels for the direct

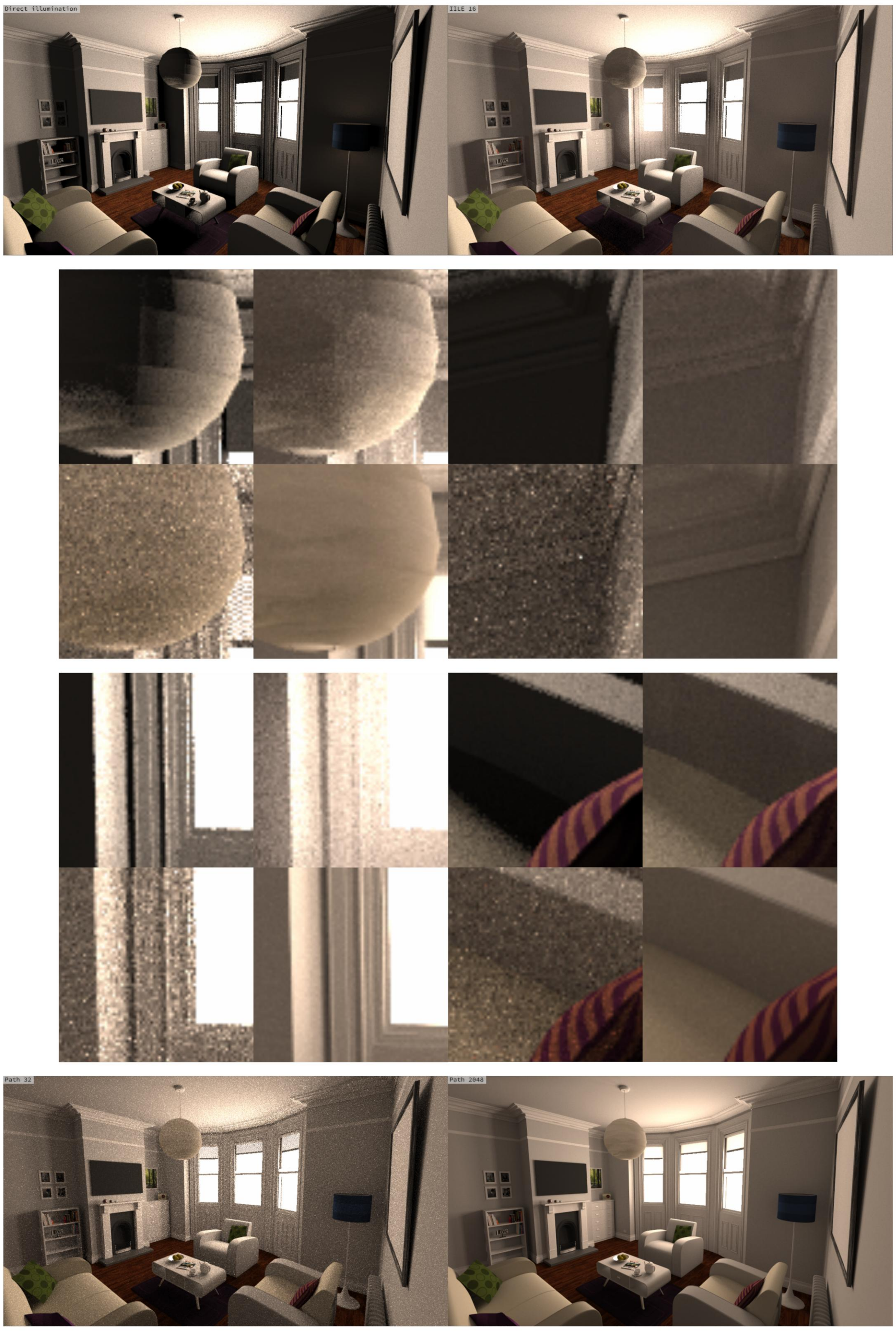


Figure 50: Comparison between Direct (top-left), OSR (top-right), Path 32 (bottom-left) and Path 2048 (bottom-right)

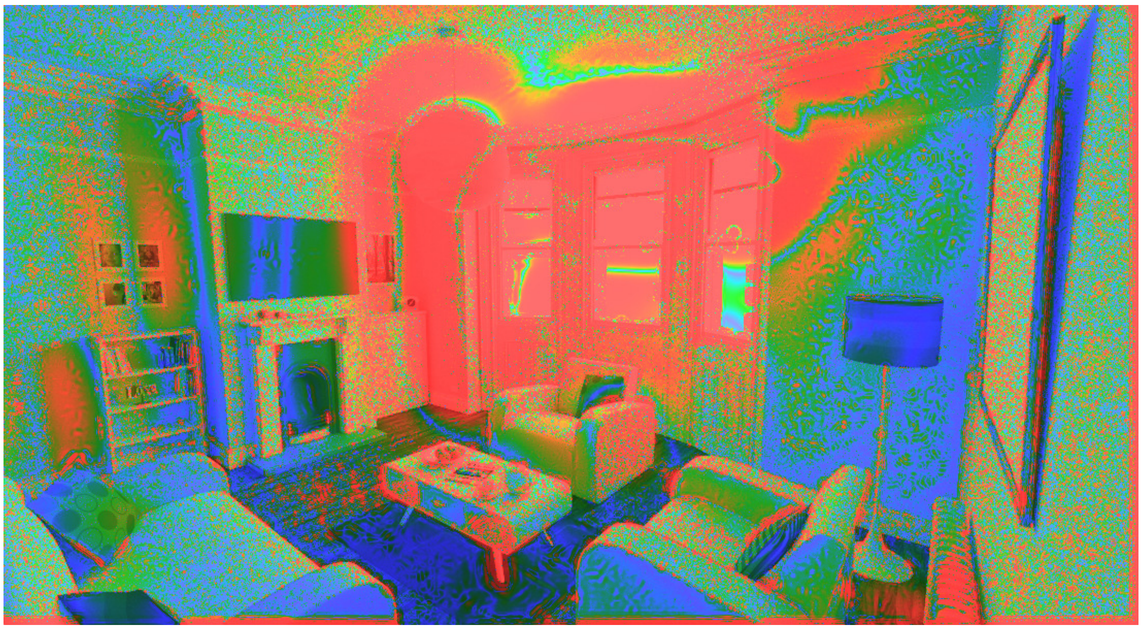


Figure 51: The HDR-VPD-2 shows that perceived difference concentrates near the window areas, and is low in the rest of the scene.

illumination integrator, and the number of *tasks* to be processed in the indirect illumination pass. We set the number of direct illumination samples to a fixed amount, enough to produce an almost noise-free first-bounce component. We then process the scene at different indirect tasks number, from 0 (no indirect component at all) up to 512.

The plots record:

- Rendering time in seconds
- Entropy in Kb. This is a measure of the amount of noise present in the rendering, and is reported as the size of the output after tonemapping the visible range and compressing the image in PNG format
- Peak Signal to Noise Ratio [17], a measure of the similarity between the image and a reference one

The plots for the Path tracer deliver the expected results: similarity with respect to the reference increases every time the number of samples is doubled, and the amount of noise decreases at a steady rate.

The metrics for the OSR output are, however, very different. The similarity to the reference increases slowly until 16 tasks, after which it remains almost stationary. The similarity metric saturates at this level due to the fact that OSR is not an unbiased algorithm even if it is capable of infinite progressive refinements. Some differences in the image cannot be cleared with longer rendering time, and any artifacts need to be judged subjectively.

The amount of noise decreases very slightly after each increase in number of tasks, although it starts at a significantly lower level than Path tracing. The OSR indirect passes do not produce much visible noise, with the small amount present being generated mainly by the direct illumination component.

As quality does not increase significantly in OSR after a certain number of tasks, we select the 16 tasks rendering to perform subjective comparisons. This image took 75 seconds using OSR, and we compare it with the Path traced render with the closest rendering time (73 seconds at 32S/px), and the reference rendered at 2048S/px (4445 seconds).

Figure 50 shows a comparison between Direct Illumination only, OSR and Path tracing. The OSR output looks much less noisy than the equal time path tracing. Lighting on the spherical lamp looks smooth despite a small amount of light bleeding from the windows area. The light bleeding is more severe in OSR near the windows, where the radiance maps were not sampled closely enough to achieve the contrast of the reference. Global illumination details in darker areas look very convincing and close to the reference.

We show the perceived difference index computed by HDR-VPD-2 [38] in Figure 51. We see that most of the difference is near the window and central lamp, which are the regions with the highest amount of light bleeding caused by the sampling and interpolation of radiance maps.

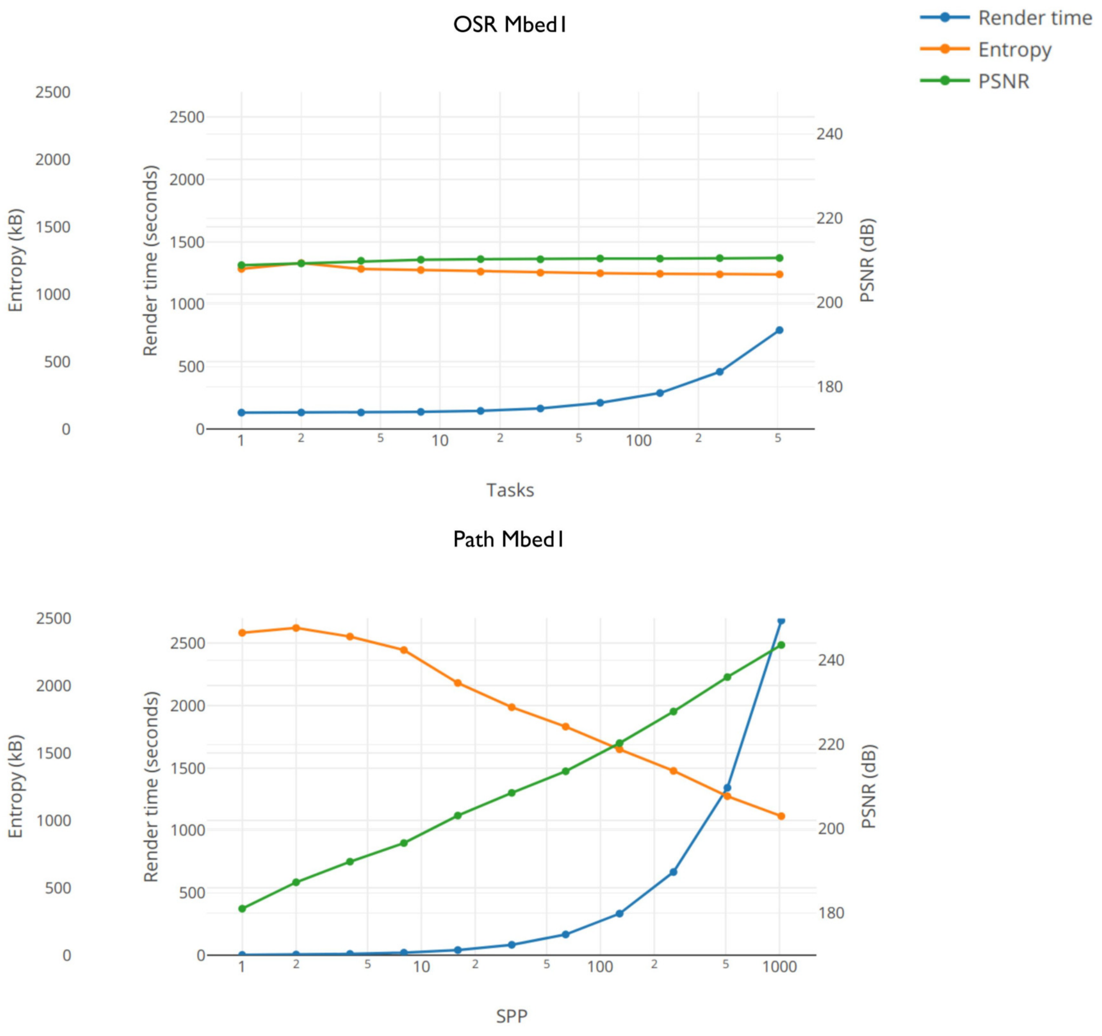


Figure 52: Quality and noise metrics on Mbed1. Above: OSR, Below: Path tracing. *Render time*: total rendering time in seconds. *Entropy*: indication of the amount of noise, measured as the final image size after PNG compression. *PSNR*: Peak Signal to Noise Ratio.

The scene *Mbed1* is significantly simpler than *White Room*. The amount of global illumination

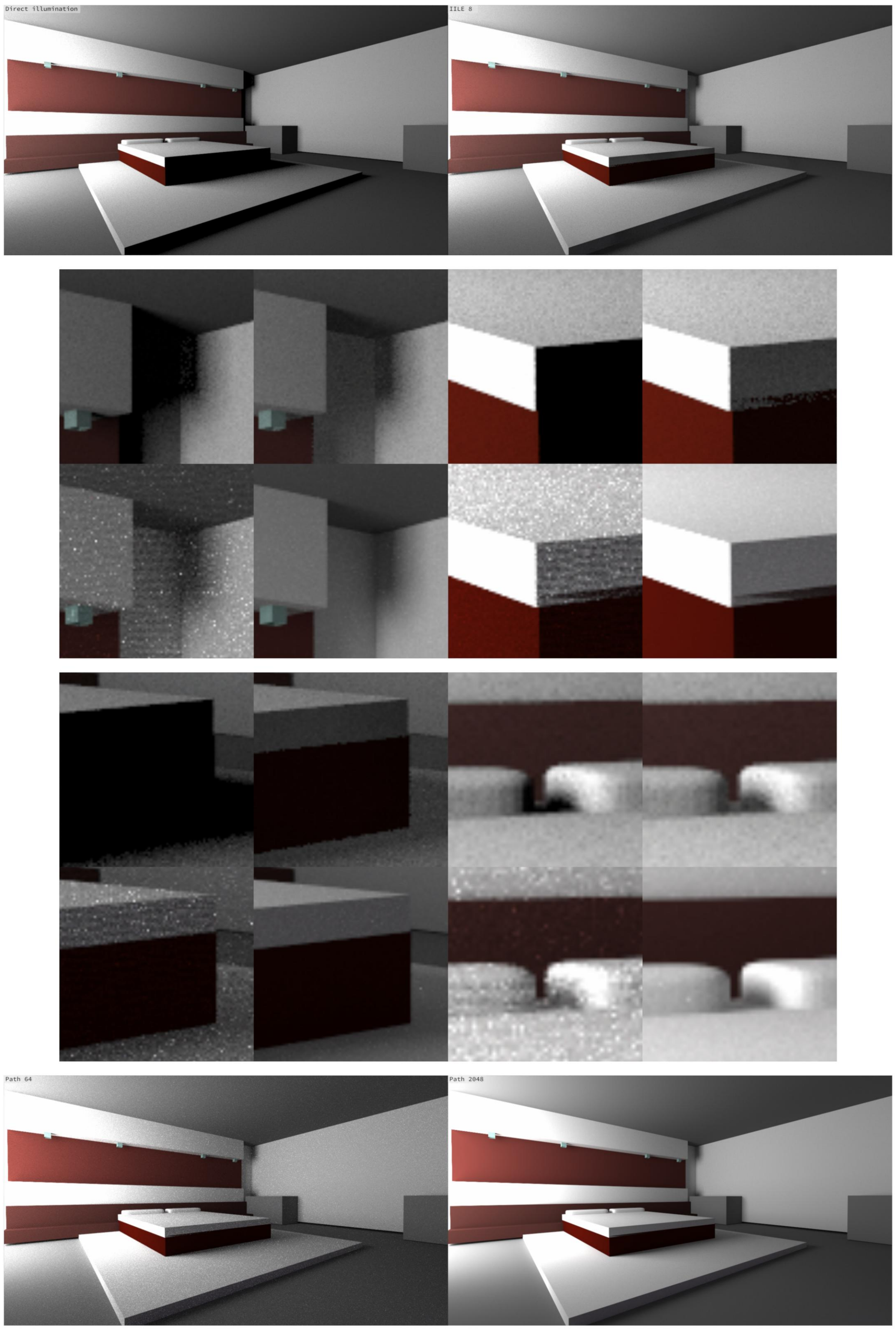


Figure 53: Comparison between Direct (top-left), OSR (top-right), Path 32 (bottom-left) and Path 2048 (bottom-right)

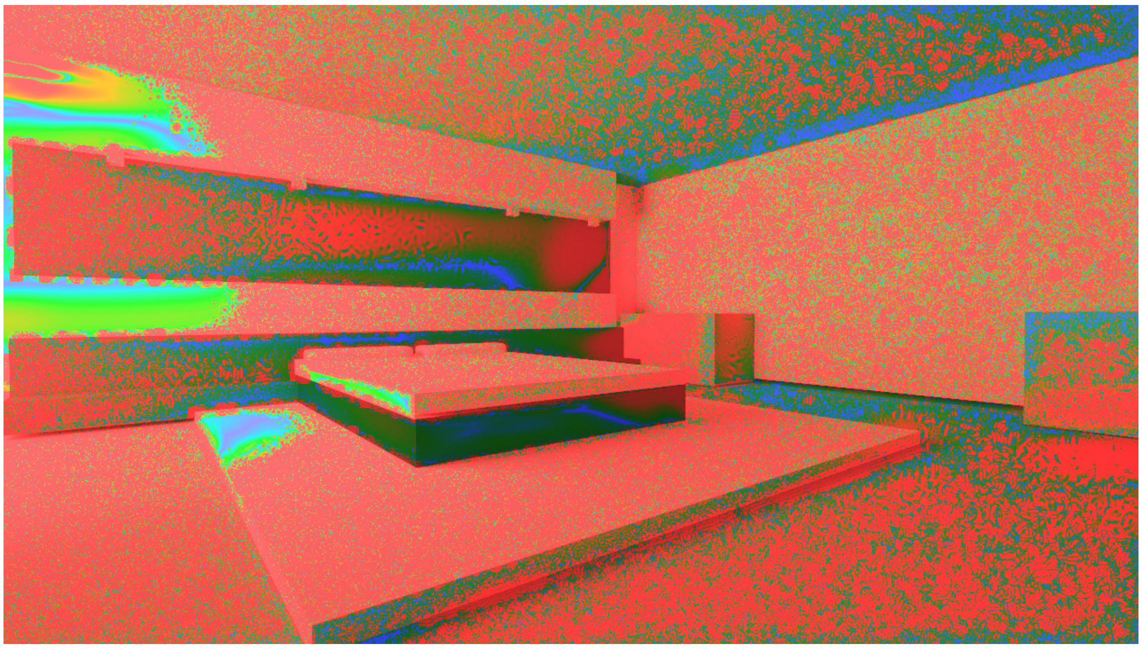


Figure 54: HDR-VPD-2 metric on Mbed1. The perceived difference is low and uniform across the image.

is small, although still very visible, and light paths towards the large area light source on the left side of the scene are mostly easy. There are a few corners and occluded areas that require the addition of indirect illumination, and we evaluate the effectiveness of OSR in the scenario in which Monte Carlo algorithms already perform well.

Figure 52 shows that the quality of the final output quickly saturates as the number of tasks increases. The level of noise is also stable and mostly produced by the direct illumination pass rendered at 64 samples per pixel.

The Path tracer, however, takes much longer to reach a similar level of similarity, and achieves a lower amount of noise only after 1024 samples per pixel.

For the comparison we choose the OSR rendering at 8 tasks, as increasing the number of indirect passes brings almost no visible change. OSR completed the rendering in 138 seconds, which includes the direct illumination pass. We select the similar-time path traced version at 64 samples per pixel, completed in 167 seconds. Figure 53 shows the detailed comparison, including direct illumination only and 2048 samples per pixel path traced reference.

Despite the simplicity of the scene, 64 Path tracing samples leave a considerable amount of noise, while OSR produces smooth and accurate indirect illumination effects in the dark corners. The magnifications show that OSR is capable of solving global illuminated details without producing visible artifacts in this scene, and that the overall appearance is very close to the reference. A comparison with the direct illumination only pass confirms that the residual noise present in the OSR image is not caused by the indirect illumination pass.

Figure 54 shows the HDR-VPD-2 perceived difference metric on Mbed1. The difference is uniform across the image and is generally low.

6.4 Interpolation

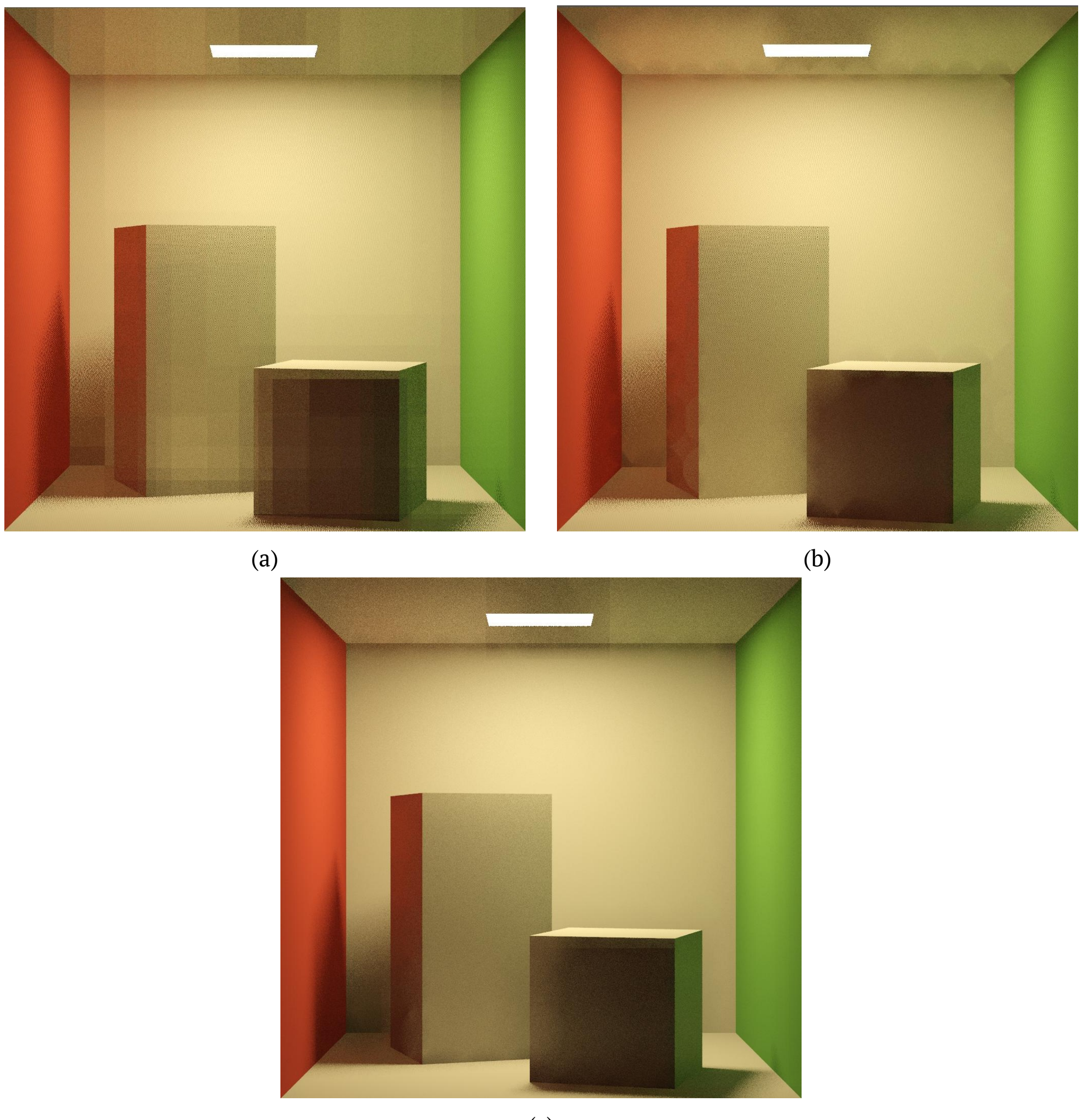


Figure 55: Comparison between simple distance based interpolation (a), interpolation based on both distance and surface normals (b), and using distance, surface normals and 3D position (c). Using surface normals improves smoothness across surfaces, and removes large artifacts and light bleeding on the edges of the cubes.

Figure 55 shows two images rendered at a limited number of samples. The Simple interpolation strategy only linearly interpolates using pixel distances on the image plane. Light bleeding and hard artifacts are clearly visible on the objects. The simple strategy is not able to take into account surfaces facing different directions, and spreads light values across the image film.

We resolve some major artifacts by including surface normals in the weighting system (Section 4.3). Some artifacts remain near the edges due to the very low number of samples, while the overall image looks smooth. Where two edges meet each other and sampling is sparse, artifacts are clearly visible.

The introduction of the third weighting element, the distances from the main camera, further improve interpolation quality in the corners by creating a smoother transition while maintaining sharp contrast.

6.5 Use Cases and Benefits

The typical use scenario is rendering an interior 3D scene where indirect light is significant. Most interior scenes have very little direct lighting, with most of the paths bouncing on several surfaces before reaching a light source. Windows and light sources are small, making path tracing inefficient. From our evaluation we can conclude that OSR is capable of significantly outperforming Path tracing in a typical interior scene by producing high quality output in a smaller time frame. The images have convincing GI effects despite some bias being present. The slowly changing indirect illumination easily masks away minor artifacts and interpolation imprecisions, while most of the details are preserved by the direct illumination pass.

OSR is appropriate for all use cases that do not require physically accurate rendering algorithms, and only need to produce high performance and high quality GI that looks convincing and realistic. Thanks to its particular quality over time curve, OSR is capable of yielding highly acceptable results after a very short amount of time, making it suitable for high performance global illumination previews for complex lighting scenarios.

7 Blender Plugin

We have developed a Blender plugin that allows to export arbitrary scenes to be rendered with PBRT-OSR. The users can easily download and install the add-on and start using the OSR renderer through a familiar and easy to use graphical interface.

We show our Blender exporter in Sections 7.1 and 7.2, and its implementation in Section 7.3. We then show our GUI application in Section 7.4 and its implementation in Section 7.5.

7.1 blendPBRTv3

Blender [2] is a well known advanced 3D modelling and rendering tool, widely used by the community. It offers standard mesh editing, sculpting, rigging, animation and much more, and is one of the industry-standard packages for 3D graphics, with almost universal compatibility with external renderers and tools. Blender is an open source project, and has evolved over time into a complex package that offers an incredible number of features to obtain all kinds of graphical effects.

The best way to make our software easily accessible for the final user is to create an exporter compatible with a well known 3D software package. Blender is a natural choice as it is open source, and provides a powerful and flexible API for add-on creation.

At the time of writing, PBRTv3 does not have any official exporter for Blender. The PBRTv3 documentation recommends the use of Blender's OBJ exporter to obtain a rough mapping of meshes and materials. The process requires the manual addition of camera information, light sources and material properties, which is a long and tedious process. It is easy to make mistakes when handling the complex 3D coordinate conversions necessary to obtain a PBRT-compatible representation, as Blender and PBRT do not share the same coordinate system, and specify camera location and direction in different ways. Converting a scene to PBRT requires therefore a large amount of time and a lot of trial and error. The development of our Blender for PBRTv3 exporter is therefore a major contribution not only to the OSR project, but for all PBRTv3 users as the plugin is compatible with standard unmodified PBRTv3 binaries as well.

We created `blendPBRTv3` as a standard Blender plugin, that can be installed and used entirely from the graphical interface. To ease the process of obtaining the necessary dependencies required to run the neural network, we provide easy-to-use installation scripts, while those users who are only interested in exporting scenes to PBRTv3 can obtain the exporter only without other components. The addition of the OSR GUI application makes the entire workflow of modelling, exporting and rendering of a scene from blender completely graphical, without any need of using the command line interface of PBRT, or leaving the workspace of Blender.

The plugin adds a rendering engine to the list of available engines in Blender, and offers a graphical interface to define scene properties, rendering settings, and materials for the objects. We support the most commonly used features, such as the Perspective Camera, Image-based lighting, and UV texture mapping. Materials can be defined in the Materials properties tab, where the plugin produces a mapping for the settings of some native PBRT materials: Matte, Plastic, Glass, Mirror, Mix. All available channels, such as diffuse colors or roughness values, can be mapped to image textures instead of constant RGB or Float values for a high degree of flexibility, and the Mix material allows to recursively combine an arbitrary number of materials by using texture maps as the mixture coefficient.

The PBRTv3 Blender Plugin has enough features to enable users to easily export new or existing scenes to be used with PBRTv3 or PBRTv3-OSR. Although we don't support the full extended feature set of PBRTv3, our tool makes it easy to render even complex scenarios, and to obtain a solid starting point for more scenes that would require further processing after exporting. As we

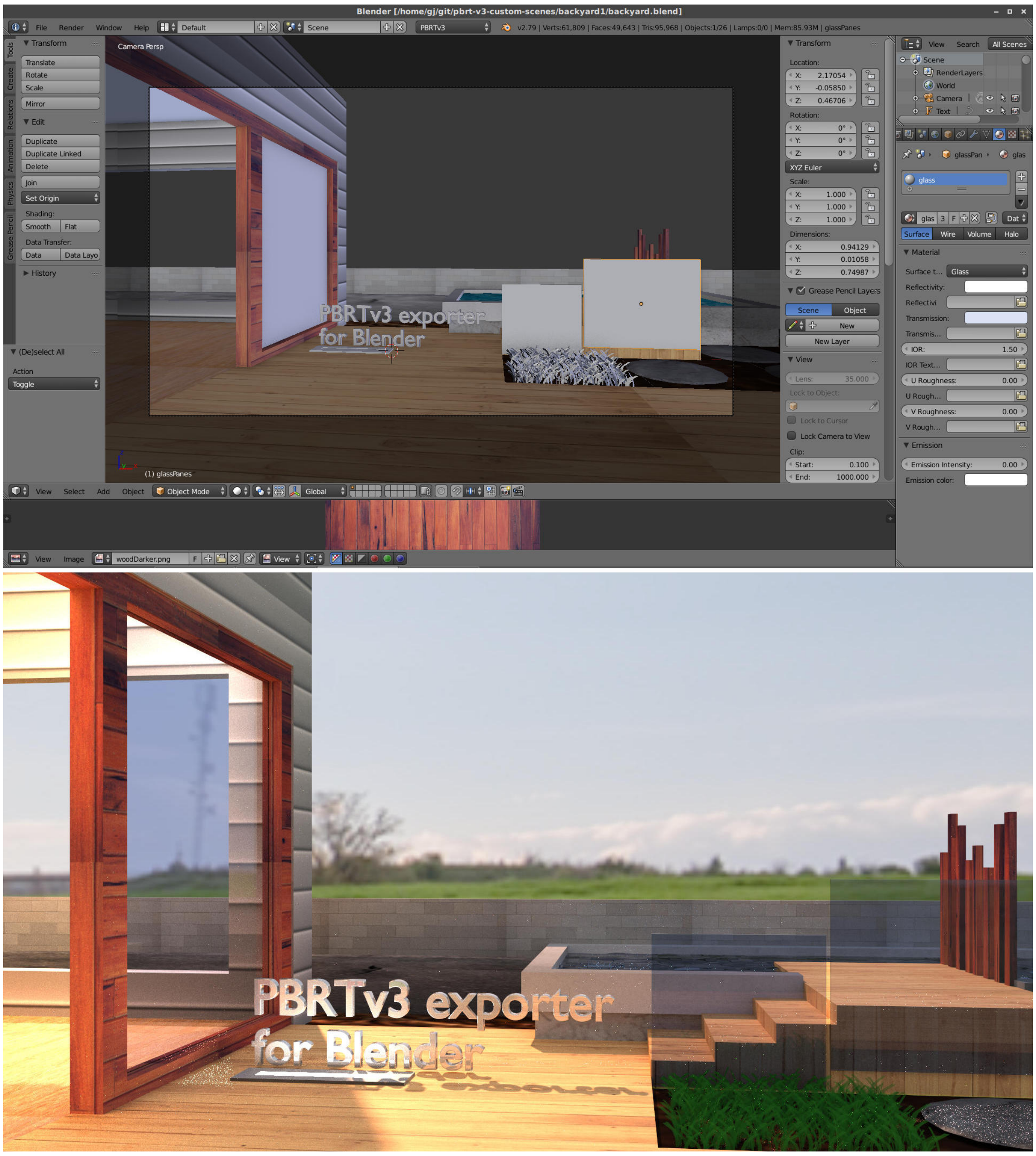


Figure 56: A simple scene in Blender, exported to PBRTv3 using the exporter. The scene contains many of the supported features, including environment lighting and a variety of materials.

preserved the overall look and feel of the Blender User Interface within the Plugin, users that are accustomed to Blender and other standard rendering engine exporters should be able to quickly become productive with our exporter.

The Plugin is fully backward compatible with unmodified PBRTv3, but it supports the extra feature when used with PBRTv3-OSR of lauching the OSR-GUI directly from Blender. Linking the GUI application allows the user to have a streamlined workflow in which pressing the Render button in Blender can automatically launch the renderer with immediate visual feedback on the final image. Further details on the GUI are in Section 7.4.

In Figure 56 is a scene created in Blender and exported to PBRTv3. We show that our exporter supports a wide range of features, and is applicable to real world workloads.

7.2 Blender Exporter Gallery

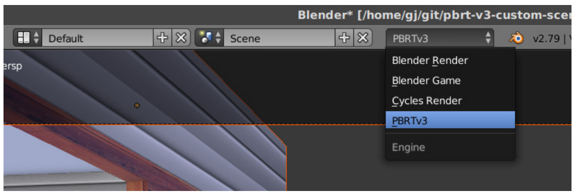
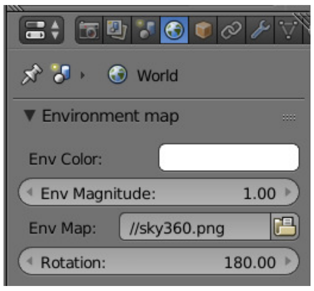


Figure 57: Once installed, PBRTv3 appears in the list of available renderers, and the user can select it from the dropdown list like any standard renderer.


World

Environment map

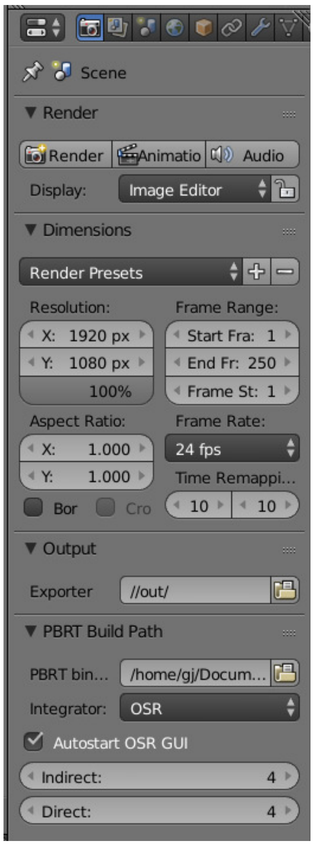
Env Color:

Env Magnitude: 1.00

Env Map: //sky360.png

Rotation: 180.00

Figure 59: The World panel includes options for the background lighting color, and the possibility to specify an environment map in PNG or EXR formats to be used as a light probe.


Scene

Render

Render | Animatio | Audio

Display: Image Editor

Dimensions

Render Presets

Resolution: X: 1920 px, Y: 1080 px, 100%

Frame Range: Start Fra: 1, End Fr: 250, Frame St: 1

Aspect Ratio: X: 1.000, Y: 1.000

Frame Rate: 24 fps

Time Remappi... 10 10

Bor Cro

Output

Exporter //out/

PBRT Build Path

PBRT bin... /home/gj/Docum...

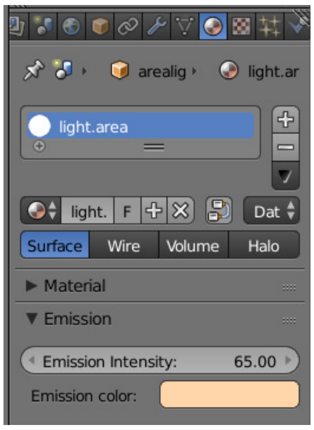
Integrator: OSR

☑ Autostart OSR GUI

Indirect: 4

Direct: 4

Figure 58: The Render Settings allow the user to specify the resolution of the film, the output directory and the Integrator settings. The OSR renderer is available among the Integrators, with settings to control the quality of the output and a checkbox to start the GUI after exporting.


arealig › light.ar

light.area

light. F Dat

Surface | Wire | Volume | Halo

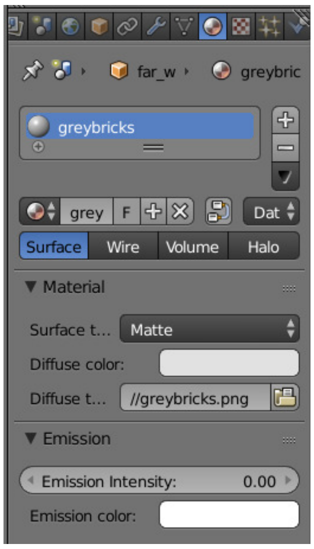
Material

Emission

Emission Intensity: 65.00

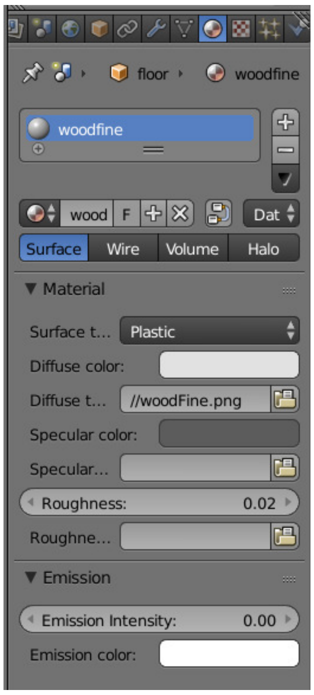
Emission color:

Figure 60: Light sources can be added as emission materials. Emission can be specified as an RGB color, and multiplied by a power factor.



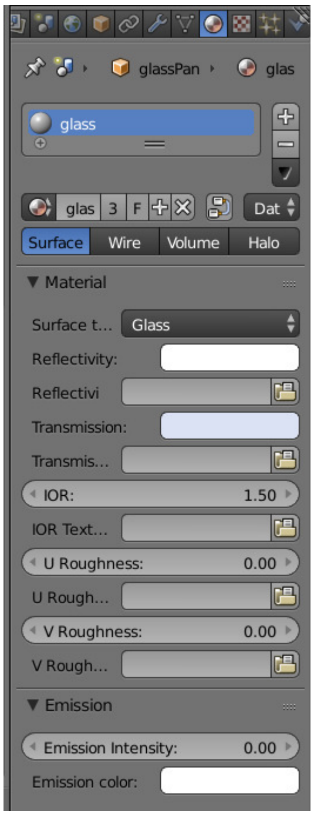
Surface t... Matte
Diffuse color:
Diffuse t... //greybricks.png
Emission Intensity: 0.00
Emission color:

Figure 61: *Matte* materials only have a diffuse component. The diffuse color can be specified as a fixed color, or as a texture.



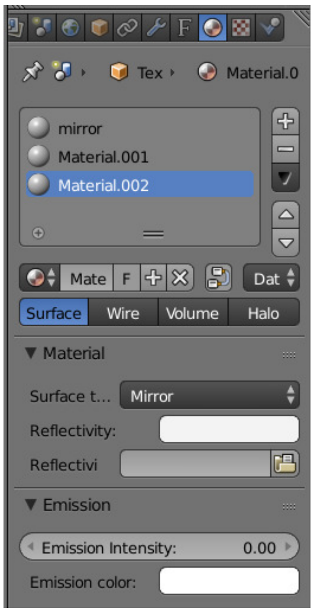
Surface t... Plastic
Diffuse color:
Diffuse t... //woodFine.png
Specular color:
Specular...
Roughness: 0.02
Roughne...
Emission Intensity: 0.00
Emission color:

Figure 62: *Plastic* has individual Diffuse and Specular components, each of them can use either an RGB color or an image texture. Roughness can be specified as a value or a float image texture, and controls the blurriness of the reflections.



Surface t... Glass
Reflectivity:
Reflectivi
Transmission:
Transmis...
IOR: 1.50
IOR Text...
U Roughness: 0.00
U Rough...
V Roughness: 0.00
V Rough...
Emission Intensity: 0.00
Emission color:

Figure 63: *Glass* defines transparent materials with an internal volume. All parameters, including Reflectivity, Transmission, Index of Refraction and anisotropic Roughness can be controlled using values or image textures.


Tex › Material.0

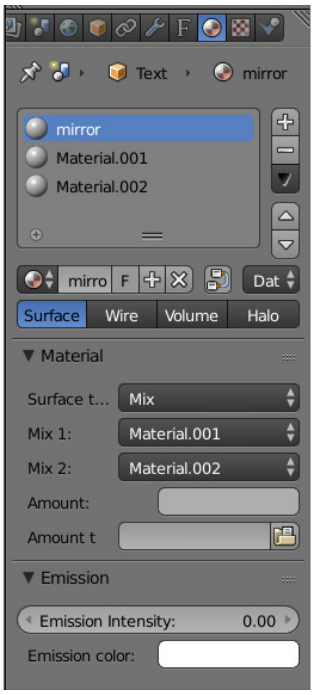
mirror
Material.001
Material.002

Mate F Dat
Surface Wire Volume Halo

▼ Material
Surface t... Mirror
Reflectivity:
Reflectivi

▼ Emission
Emission Intensity: 0.00
Emission color:

Figure 64: *Mirror* materials are perfectly reflective, and have a reflectivity channel.


Text › mirror

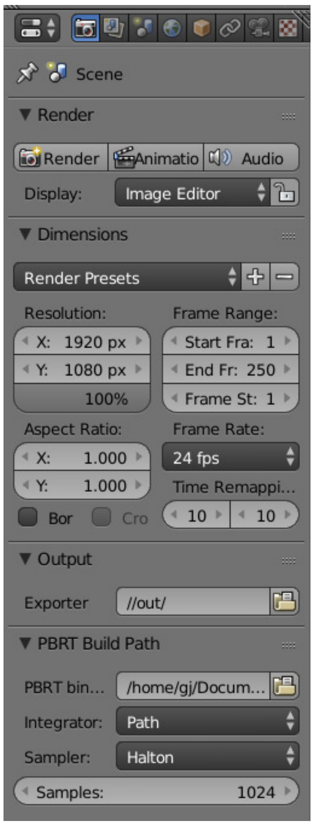
mirror
Material.001
Material.002

mirro F Dat
Surface Wire Volume Halo

▼ Material
Surface t... Mix
Mix 1: Material.001
Mix 2: Material.002
Amount:
Amount t

▼ Emission
Emission Intensity: 0.00
Emission color:

Figure 65: The *Mix* material allows to specify two other materials to be mixed together. The mixing factor can be a constant value, or be specified through an image map to create complex materials.


Scene

▼ Render
Render Animatio Audio
Display: Image Editor

▼ Dimensions
Render Presets
Resolution: Frame Range:
X: 1920 px Start Fra: 1
Y: 1080 px End Fr: 250
100% Frame St: 1
Aspect Ratio: Frame Rate:
X: 1.000 24 fps
Y: 1.000 Time Remappi...
Bor Cro 10 10

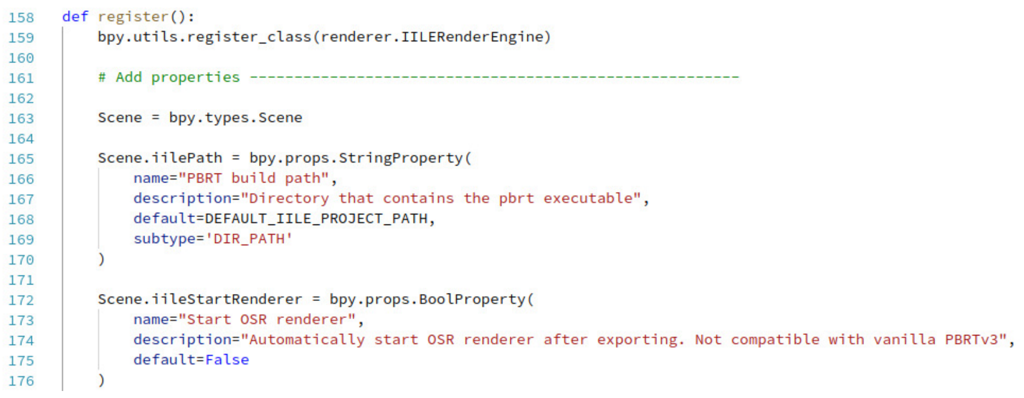
▼ Output
Exporter //out/

▼ PBRT Build Path
PBRT bin... /home/gj/Docum...
Integrator: Path
Sampler: Halton
Samples: 1024

Figure 66: Standard PBRT integrators such as Path are supported, with common options such as the sampler and number of samples per pixel to be rendered.

7.3 Blender Exporter Implementation

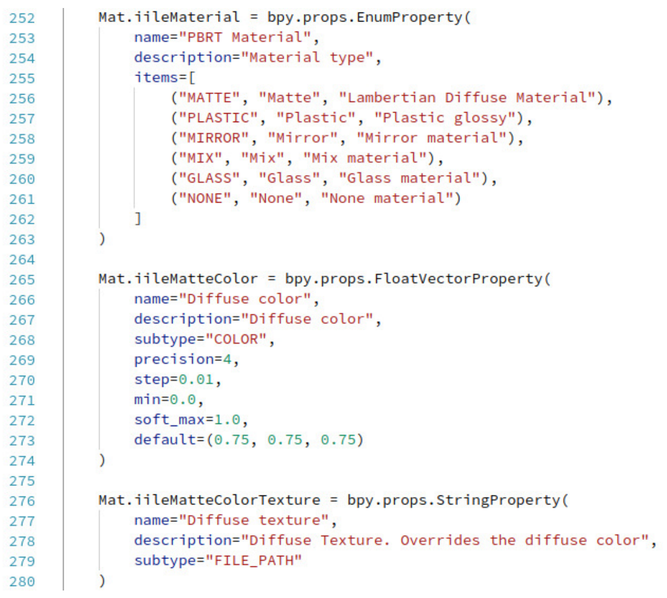
We implemented blendPBRTv3 as a standard Blender Add-On using the Blender 2.79 API [3]. The Blender API has full Python 3 access to every component in the application, including objects, materials, and UI panels.



```python
def register():
    bpy.utils.register_class(renderer.IILERenderEngine)

    # Add properties ------------------------------------------------------

    Scene = bpy.types.Scene

    Scene.iilePath = bpy.props.StringProperty(
        name="PBRT build path",
        description="Directory that contains the pbrt executable",
        default=DEFAULT_IILE_PROJECT_PATH,
        subtype='DIR_PATH'
    )

    Scene.iileStartRenderer = bpy.props.BoolProperty(
        name="Start OSR renderer",
        description="Automatically start OSR renderer after exporting. Not compatible with vanilla PBRTv3",
        default=False
    )
```

Figure 67: `register` is the entry point of the plugin.

Any Blender Add-on is a Python3 script, or collection of scripts, that implement a very simple interface composed of 2 functions: `register` and `unregister`. The `register` function is called by Blender when the plugin is loaded according to the user or scene configuration, and is responsible for loading and setting up classes, UI elements, data properties and actions. The `unregister` function is called when the plugin is disabled from the User Preferences, and performs all the actions necessary to clean up the environment and deregistering all custom classes. blendPBRTv3's `register` registers a new `RenderEngine`, adds a number of new properties used to store data specific to PBRT materials and settings, and adds some UI elements. Figure 67 shows the first few lines of blendPBRTv3.

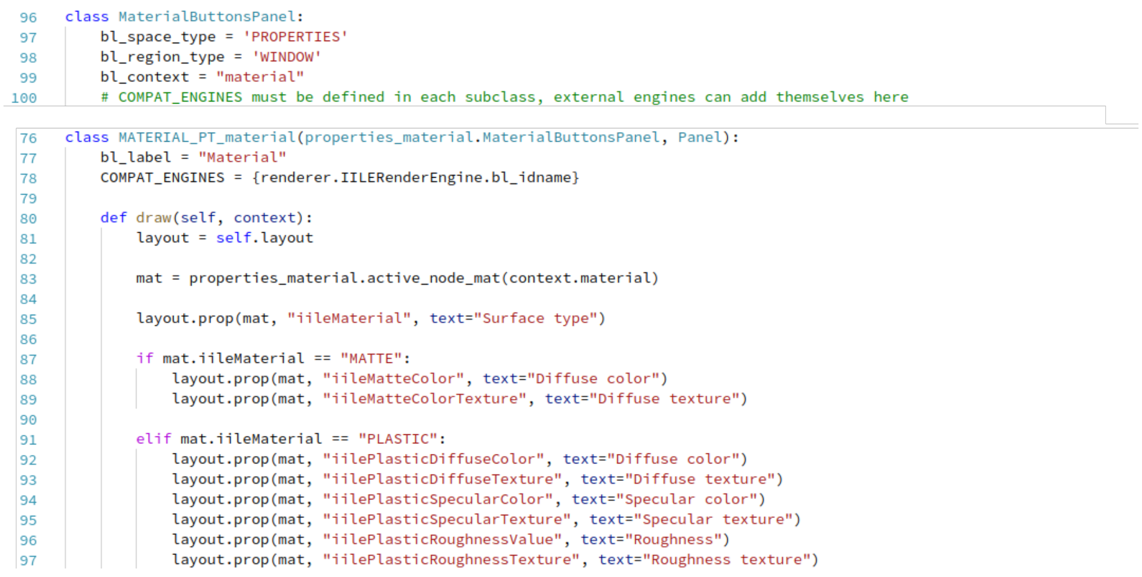


```python
    Mat.iileMaterial = bpy.props.EnumProperty(
        name="PBRT Material",
        description="Material type",
        items=[
            ("MATTE", "Matte", "Lambertian Diffuse Material"),
            ("PLASTIC", "Plastic", "Plastic glossy"),
            ("MIRROR", "Mirror", "Mirror material"),
            ("MIX", "Mix", "Mix material"),
            ("GLASS", "Glass", "Glass material"),
            ("NONE", "None", "None material")
        ]
    )

    Mat.iileMatteColor = bpy.props.FloatVectorProperty(
        name="Diffuse color",
        description="Diffuse color",
        subtype="COLOR",
        precision=4,
        step=0.01,
        min=0.0,
        soft_max=1.0,
        default=(0.75, 0.75, 0.75)
    )

    Mat.iileMatteColorTexture = bpy.props.StringProperty(
        name="Diffuse texture",
        description="Diffuse Texture. Overrides the diffuse color",
        subtype="FILE_PATH"
    )
```

Figure 68: Configuration of the Materials dropdown and of the properties of Matte materials.

A `Property` in blender is a data slot that can be used to store information and values entered by the user. The values are stored automatically in the `.blend` project files. A Property can be associated to a built-in class, such as an Object, Material, or Scene, and all instances of the type inherit the property. Properties are designed to be easily configurable and accessible, as each type of Property has its associated UI element configuration to be easily added to a graphical panel, and can be read and written like a regular Python3 variable in a completely transparent way. We extensively use properties to define data fields that store PBRT settings, material color values, texture image paths and more.

In Figure 68 we show some lines where we define the list of materials and the properties specific for the Matte material. The `EnumProperty` can be used to define a dropdown list. The Material type list is fixed, but the API allows to pass a function instead of a list to obtain dynamic lists. The definition of two properties for Matte materials follows. The Diffuse color is a color `FloatVector` property, which is associated to a color picker in the user interface. The Diffuse color texture is a `StringProperty` that stores the path to an image file. Defining the subtype `FILE_PATH` makes the UI open a file browser when the user clicks on the property.

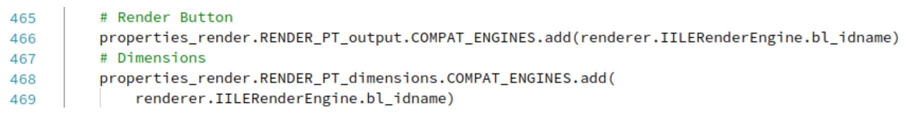


```python
class MaterialButtonsPanel:
    bl_space_type = 'PROPERTIES'
    bl_region_type = 'WINDOW'
    bl_context = "material"
    # COMPAT_ENGINES must be defined in each subclass, external engines can add themselves here
```

```python
class MATERIAL_PT_material(properties_material.MaterialButtonsPanel, Panel):
    bl_label = "Material"
    COMPAT_ENGINES = {renderer.IILERenderEngine.bl_idname}

    def draw(self, context):
        layout = self.layout

        mat = properties_material.active_node_mat(context.material)

        layout.prop(mat, "iileMaterial", text="Surface type")

        if mat.iileMaterial == "MATTE":
            layout.prop(mat, "iileMatteColor", text="Diffuse color")
            layout.prop(mat, "iileMatteColorTexture", text="Diffuse texture")

        elif mat.iileMaterial == "PLASTIC":
            layout.prop(mat, "iilePlasticDiffuseColor", text="Diffuse color")
            layout.prop(mat, "iilePlasticDiffuseTexture", text="Diffuse texture")
            layout.prop(mat, "iilePlasticSpecularColor", text="Specular color")
            layout.prop(mat, "iilePlasticSpecularTexture", text="Specular texture")
            layout.prop(mat, "iilePlasticRoughnessValue", text="Roughness")
            layout.prop(mat, "iilePlasticRoughnessTexture", text="Roughness texture")
```

Figure 69: Definition of the Materials UI panel.

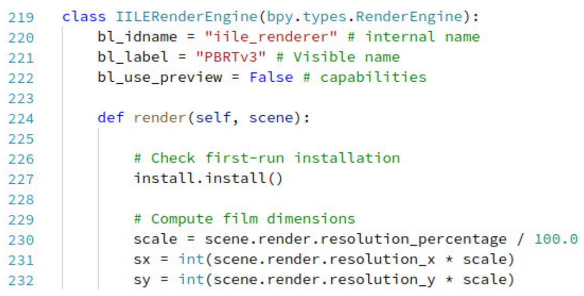
Custom User Interface panels can be defined in a plugin as classes. In Figure 69 we show a few lines from our custom Materials panel, which inherits from Blender's built-in `MaterialButtonsPanel`, which specifies a location within Blender. The `COMPAT_ENGINES` specifies the list of `RenderEngines` that are compatible with this graphical panel, and we only add `blendPBRTv3`. The main function in a panel is the `draw` method. We simply add in the UI the properties that we have previously defined in `register`. The flexibility of the UI allows to have control flow in the `draw` method. In our case we use `if` statements to show or hide properties that are specific to particular materials.



```python
    # Render Button
    properties_render.RENDER_PT_output.COMPAT_ENGINES.add(renderer.IILERenderEngine.bl_idname)
    # Dimensions
    properties_render.RENDER_PT_dimensions.COMPAT_ENGINES.add(
        renderer.IILERenderEngine.bl_idname)
```

Figure 70: Existing panels can be re-used by adding `blendPBRTv3` to the list of compatible engines.

Some of the panels in Blender can be reused. This saves programming time, and leaves the user interface identical to the default. Reusing of an existing panel can be achieved by adding our RenderEngine to the list of supported engines, as shown in Figure 70.



```python
class IILERenderEngine(bpy.types.RenderEngine):
    bl_idname = "iile_renderer" # internal name
    bl_label = "PBRTv3" # Visible name
    bl_use_preview = False # capabilities

    def render(self, scene):

        # Check first-run installation
        install.install()

        # Compute film dimensions
        scale = scene.render.resolution_percentage / 100.0
        sx = int(scene.render.resolution_x * scale)
        sy = int(scene.render.resolution_y * scale)
```

Figure 71: The `RenderEngine` defines the single `render` method.

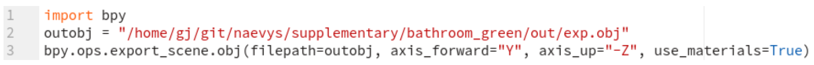
The main component of the plugin is the implementation of the `bpy.types.RenderEngine` class, which defines a new rendering engine. Here we define our custom rendering function that exports the scene to disk. Figure 71 shows that the class defines the `render` method, which is called by Blender when the user initiates a Render action.

The rendering process of `blendPBRTv3` can be subdivided as several tasks:

1. Export the scene to OBJ and MTL files
2. Run `obj2pbrt`
3. Compute scene transformation and write camera and film settings
4. Resolve materials dependencies and properties
5. Parse and update light sources
6. Launch OSR GUI

1. Export to OBJ and MTL

The plugin first initiates the meshes by exporting the entire scene into the OBJ format [24] using the Blender built-in OBJ exporter. This step outputs a `.obj` and a `.mtl` files, containing the primitives vertices, UV coordinates, and basic material properties.



```python
import bpy
outobj = "/home/gj/git/naevys/supplementary/bathroom_green/out/exp.obj"
bpy.ops.export_scene.obj(filepath=outobj, axis_forward="Y", axis_up="-Z", use_materials=True)
```

Figure 72: A generated Python3 script for the auxiliary Blender instance.

A plugin action in Blender is un synchronously and is required to return upon finishing. Additionally, in order for Blender to be able to run multiple actions concurrently, any action is not permitted to perform UI refreshes or draw operations as they can cause conflicts and crashes. The OBJ exporter in Blender unfortunately does draw updates to the mouse cursor to show its progress. This behaviour is not permitted in a plugin, and we have indeed experienced random crashes. To solve this issue, we perform the task using a separate Blender instance started in CLI mode, running a Python script generated by `blendPBRTv3` that exports the scene into the output directory. Figure 72 shows a script generated by the exporter.

The approach of using an independent instance to avoid graphical update conflicts has worked very well in all scenarios, but brings a few minor disadvantages: memory usage is increased due to the necessity of loading the scene an additional time, and the user is required to save the project to disk before rendering, as any unsaved changes might not be reflected in the exported files.

The OBJ format does not encode which coordinate system is used. To account for the different coordinate systems between Blender and PBRT, we configure the OBJ exporter to assume Y axis forward and -Z axis up.

2. obj2pbrt

The OBJ and MLT files contain all the objects in the scene, and a rough approximation of their materials. PBRTv3 includes an executable, `obj2pbrt`, which takes an OBJ file and outputs a corresponding PBRT scene file containing geometry and some material information.

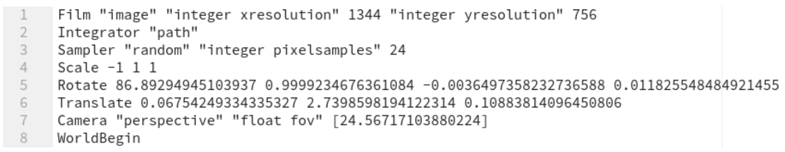
`blendPBRTv3` needs to find the PBRT executables in order to perform this step. For compatibility with the original PBRTv3 project and possibly its forks, we distribute the plugin with PBRT-OSR included, or excluded. If the PBRT project is found in the plugins directory, the exporter automatically extracts the files. Alternatively, the exporter attempts to use the `pbrt` and `obj2pbrt` executables present in the `PATH`, and, as a third options, allows the user to specify the path to the directory where the executables are located.

After obtaining the PBRT scene files, we use PBRT's `--toply` function to extract large objects into PLY files, and make the scenefile smaller and easier to work with.

At this stage the PBRT scene only contains basic geometrical information, and lacks specifications for the camera setup, lighting and materials.

3. Transformations, Camera and Film

Global settings and properties, such as the World to Camera transformation, camera properties, film, integrator and sampling settings can be defined at the beginning of a PBRT scene file. The rest of the information that was generated by `obj2pbrt` can be appended within the tags `WorldBegin` and `WorldEnd`.



```
Film "image" "integer xresolution" 1344 "integer yresolution" 756
Integrator "path"
Sampler "random" "integer pixelsamples" 24
Scale -1 1 1
Rotate 86.89294945103937 0.9999234676361084 -0.0036497358232736588 0.011825548484921455
Translate 0.06754249334335327 2.7398598194122314 0.10883814096450806
Camera "perspective" "float fov" [24.56717103880224]
WorldBegin
```

Figure 73: Camera and global settings generated by `blendPBRTv3`.

Figure 73 shows an example scene file header generated by `blendPBRTv3`. Image film settings are read from the Blender API, and Integrator and Sampler are obtained from the custom properties we defined in the exporter.

World to Camera tranformation directives in the scene file are applied to the global transformation matrix, which is always initialized as the Identity matrix. When the Camera object is initialized, the current transformation matrix is used. The first transformation is a flip on the X axis, necessary to switch from a right-handed coordinate system (Blender) to a left-handed one (PBRT). The Rotation defines the rotation magnitude in degrees, and the 3D vector that is used as rotation axis. Blender by default uses XYZ Euler rotation coordinates, but can be configured to use Axis Angle values that can be easily mapped to PBRT rotations. The translation simply encodes the world location of the camera.

We currently only support the `perspective` camera, which takes a Field of View argument, also read from the default camera properties.

At the end of the header, we write `WorldBegin` to signal the beginning of materials and objects definitions.

4. Resolve materials

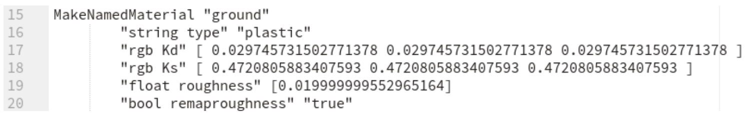
Material types and parameters are encoded for each object, in properties that are defined during plugin registration. `blendPBRTv3` currently supports Matte, Plastic, Mirror, Mix, Glass and None material types.

All materials have straightforward properties such as diffuse and specular colors, with the exception of the Mix material. A Mix material is a weighted blend of two other materials. The inputs of Mix are therefore two material names, and a weighting factor that can be a constant or a texture. The definition of the Mix material therefore allows to create complex materials starting from more basic ones, and blending them over the surface depending on the values of a texture. Mix materials can be recursively chained with other Mix materials in an arbitrarily deep configuration.

In a PBRT scene file, materials can be pre-defined before being usage by assigning names. Material names are a useful feature because it allows the plugin to determine the corresonding materials assigned to each object by finding the correct material in the Blender API by its name. Additionally, names are used to specify Mix material components.

When a Mix material is defined, its two component materials need to have already been defined. This requirement poses the need of processing the materials in a particular order. `blendPBRTv3` starts materials processing by recursively building a tree of dependencies. The exporter visits each material defined in Blender, and adds its component dependencies before the material itself if the type is Mix. As Mix components can also have type Mix, the process is recursive.

Once the dependencies order have been resolved, we translate each material property in its corresponding PBRT scene syntax.



```
MakeNamedMaterial "ground"
        "string type" "plastic"
        "rgb Kd" [ 0.029745731502771378 0.029745731502771378 0.029745731502771378 ]
        "rgb Ks" [ 0.4720805883407593 0.4720805883407593 0.4720805883407593 ]
        "float roughness" [0.019999999552965164]
        "bool remaproughness" "true"
```

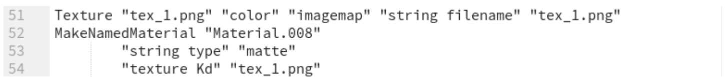
Figure 74: Named materials define a name and a list of attributes.

Figure 74 shows a Plastic material exported by the plugin. The material has a name, and a list of attributes.

Most of the attributes can use an image texture instead of a constant value. The UI allows the user to select the path to an image file. `blendPBRTv3` copies the selected texture in the output directory using a sequentially generated file name, and links the texture to the appropriate channel. An example of a texture map assigned to a material channel is shown in Figure 75.

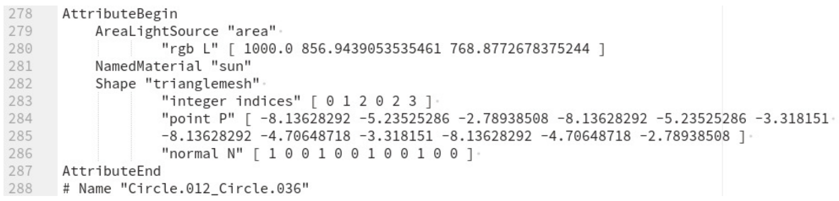
5. Parse and update light sources

`blendPBRTv3` supports two kinds of light sources: Infinite and Emission objects. A single Infinite light source can be defined in the World properties panel in Blender, and it can be a single environment color, or an environment map in PNG or EXR format. The Infinite light is added to



```
Texture "tex_1.png" "color" "imagemap" "string filename" "tex_1.png"
MakeNamedMaterial "Material.008"
        "string type" "matte"
        "texture Kd" "tex_1.png"
```

Figure 75: Image textures are supported in many material channels.



```
AttributeBegin
    AreaLightSource "area"
            "rgb L" [ 1000.0 856.9439053535461 768.8772678375244 ]
    NamedMaterial "sun"
    Shape "trianglemesh"
            "integer indices" [ 0 1 2 0 2 3 ]
            "point P" [ -8.13628292 -5.23525286 -2.78938508 -8.13628292 -5.23525286 -3.318151
            -8.13628292 -4.70648718 -3.318151 -8.13628292 -4.70648718 -2.78938508 ]
            "normal N" [ 1 0 0 1 0 0 1 0 0 1 0 0 ]
AttributeEnd
# Name "Circle.012_Circle.036"
```

Figure 76: An example of `AreaLightSource` attribute.

the scene file in a way that is similar to that used to generate materials.

Objects can be configured to emit light, and generalize all other kinds of light sources. The PBRT `AreaLightSource` is in fact defined over an object, and can assume therefore any shape despite its name referring to Area lights. The plugin allows the user to specify an Emission value for a material. We did not define a custom Emission property, but re-used the one defined for the Blender Internal Render Engine, as the OBJ exporter is capable of reading emission values and generate corresponding emission values that are converted to `AreaLightSource` by `obj2pbrt`. We use an additional property to allow the user to define an emission color.

The exporter parses the PBRT scene file. The parser reads the input line by line, and groups up structures into blocks based on their indentation level. The exporter then analyzes each block, and performs the following operations:

- Delete the body of all `MakeNamedMaterial` blocks because the exporter generates more detailed materials in step 4.
- Add emission color to all `AreaLightSource` attributes by matching their assigned material names.

Figure 76 shows an example of an `AreaLightSource` attribute processed by `blendPBRTv3`.

The scene information generated in steps 3 and 4 are written at the beginning of the processed PBRT scene file, and the new output is written to the output directory of the project.

6. Launch OSR GUI

When the OSR integrator is selected, the user can choose whether to run the export job only, or launch the GUI application automatically. The GUI executable resides in a parent directory of the PBRT binaries, and is automatically found by the plugin assuming that the directory structures was not modified. The process is started as a Python3 `subprocess`, and launched synchronously. The GUI starts rendering automatically.

7.4 OSR GUI

We implemented a simple GUI for PBRTv3-OSR that can be started automatically from the Blender Plugin. This makes the workflow simple and graphically based. The scene starts to render after the user presses the Render button in Blender, and the GUI produces visual feedback on the current state of the final image with very little delay.

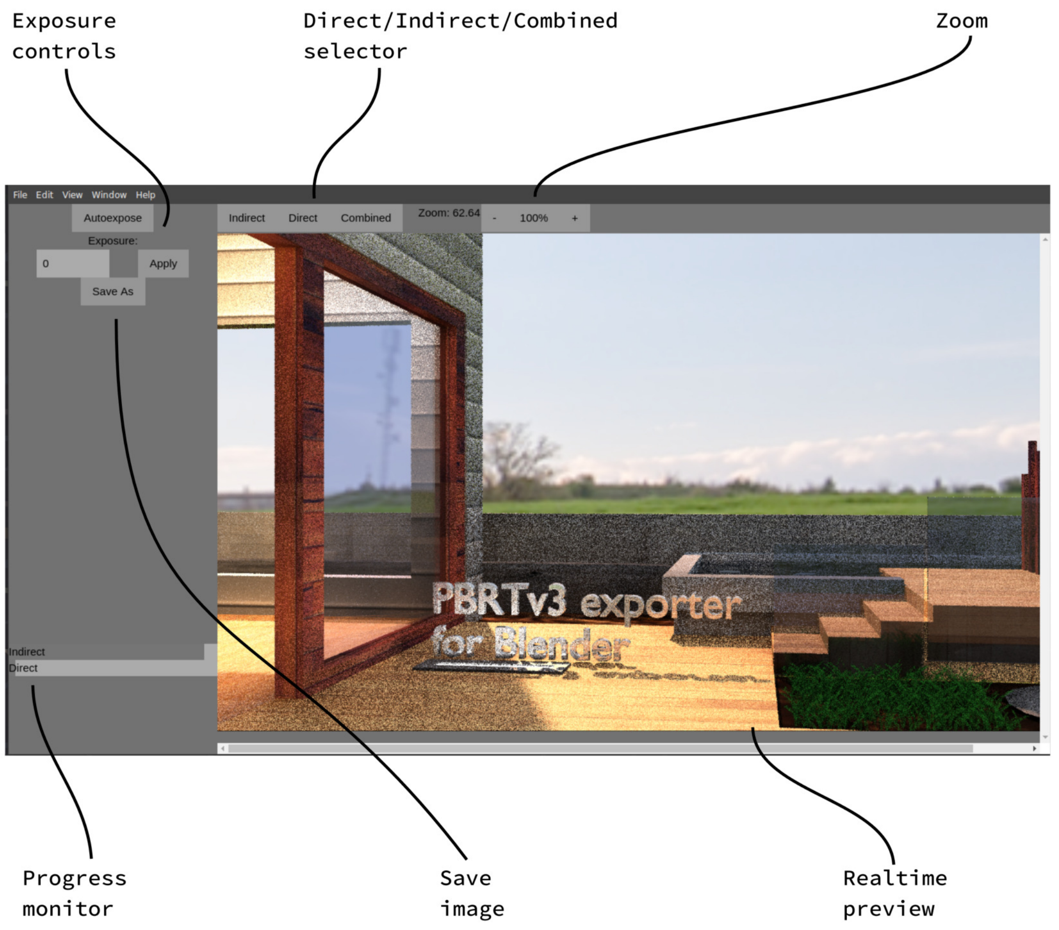


Figure 77: A screenshot of the GUI when rendering a scene. The GUI has tonemapping settings, and shows the progress of both direct and indirect illumination passes.

The OSR GUI shows the realtime rendering output of PBRT in its main frame. The image can be resized by the user and navigated using standard controls. By default, the GUI applies an automatic linear tonemapping to the high dynamic range image produced by PBRT, and the user can override this by setting manually the gain factor.

At the top of the preview, it is possible to switch between the combined output, direct illumination only and indirect illumination only. This can be a useful feature to visually determine the balance of computational resources to be allocated to the direct and indirect illumination components.

The currently displayed image can be saved by the user as an LDR image using the *Save As* button, and two progress bars display the progress on direct and indirect passes.

7.5 OSR GUI Implementation

The GUI is written in Javascript, HTML and CSS using the Electron framework [6]. Electron is based on the Chromium Browser [5], Node.js [16] and the Chrome V8 Javascript engine [4]. These tools make it possible to write a graphical desktop application using javascript, HTML and CSS. The full compatibility with both browser javascript libraries and Node.js javascript components make the system extremely flexible thanks to a large amount of code and tools already available.

In the frontend we used AngularJS [1] to handle data distribution and events, and in the backend a few standard Node.js libraries to manage processes and files.

The application runs in a single browser window based on W3CSS styling [23]. The colors have been customized to match those in Blender to make the appearance more uniform.

After all native components are loaded, the GUI starts PBRT in a Node.js *child process*. Bidirectional communication with PBRT is based on both standard in and out streams, and files exchanged in a Control Directory. The Control Directory is a temporary location on disk where PBRT can output intermediate results, such as partially rendered high dynamic range images, while the standard streams are mainly used to signal events and updates, such as the progress on the rendering or the availability of new files in the Control Directory.

Tonemapping from the high dynamic range PFM files produced by PBRT to a low dynamic range format is performed using an additional tool, *cpfm*, written in C++ for performance reasons. While our Python3 *pfm* module, used extensively for our machine learning model, has the necessary features, it takes a relatively long time to load the libraries and process the data. Executing a single tonemap task of a 2 Megapixel image using *pfm* through a subprocess call from the GUI takes 0.5 seconds, while *cpfm* requires less than 0.1 seconds. Although this difference might not generally lead to rewriting software in a different language, the difference in usability in our case is important, because the tonemapper is called every time a new update is produced by PBRT and needs to be displayed by the GUI, as well as when the user updates tonemapping settings and expects an immediate change in the preview.

The two parameters for HDR to LDR linear tonemapping are *gain* and *gamma*, and are applied as:

$$l_{x,y,c} = 255 * clamp(h_{x,y,c} * 2^{gain})^{1/gamma}$$

Where $l_{x,y,c}$ is the output LDR value for a pixel (x, y) with 8 bits per channel, $h_{x,y,c}$ is the input HDR value of a pixel, and the *clamp* function clips values to fall within the 0 to 1 range. Gain is determined by the automatic tonemapping mode or provided by the user, and Gamma is set at a fixed 1.8 in the GUI code.

cpfm can perform both automatic and manual tonemapping. In automatic mode, there is no additional command line argument, and the software computes the mean brightness of the input image, and first attempts to set a gain level that would push the mean to be 1.0. *cpfm* proceeds by progressively reducing the gain factor by 1 until less than 5% of the image is clipped at its top. After determining the gain level, gamma correction is applied, and the final image is saved in BMP format, chosen for its performance due to the lack of compression. In manual mode, the gain parameter is passed on the command line, and used directory for tonemapping.

The automatic tonemapping strategy in *cpfm* is robust and yields a balanced output in all tested scenarios. Furthermore, its output is stable with different levels of noise.

8 Discussion

The OSR project offers a new biased rendering method for global illumination capable of producing high quality results in a small amount of time, while supporting a wide range of material types and scene compositions.

Compared to other methods, OSR does not require any form of precomputation on the target scene, and accelerates the computation of global illumination at a virtually unbounded number of indirect bounces at the cost of a small amount of bias introduced.

8.1 Limitations

Even though the progressive refinement algorithm permits the user to set an arbitrary number of passes, OSR remains a biased approach to global illumination. The neural network, although very accurate in typical scenarios, can produce artifacts, inaccurate results, and loss of detail, causing hard to correct errors.

A second important limiting factor is the fixed resolution of the radiance maps, set for OSR at a default of 32x32 pixels. Small details are inevitably lost.

These two limitations become particularly visible when computing globally illuminated glossy reflections, which appear less defined, although less noisy, than the path traced reference.

Temporal stability is another limitation of OSR. The rendering tiles can have large variations from one sample to another, and while being perceived as smooth transitions in a single frame, animations would display them as low frequency flickering of large areas in the scene. The lack of temporal stability indicates that OSR would not be adequate for animation rendering. To solve the temporal stability issue, a persistent caching system for radiance maps can be implemented, allowing animations to be rendered at faster rates and with less temporal noise caused by flickering of radiance maps.

Due to the low resolution of the indirect radiance maps, OSR is not capable of producing accurate caustics, which require a much denser sampling of indirect light. As light that travels from glass windows can also be considered as a kind of caustics, OSR is not adequate to render interior scenes illuminated through windows, similarly to how Path tracers have extreme difficulty at finding valid paths in the same scenario. In such a case, the Direct illumination component would be completely black, and the OSR integrator would be attempting to resolve all light paths, even those that would appear to be directly illuminating the scene. The overall scene would look highly splotchy and unrealistic: the underlying one-shot maps are not accurate enough as those are also based on a Path tracer, but OSR still attempts to generate shadows and ambient occlusion. Under these scenarios, Bidirectional Metropolis Light Transport and Photon Mapping remain primary choices for both performance and quality.

8.2 Future Work

In the current implementation, OSR uses separate Python processes to evaluate the neural network at runtime. The evaluation processes use a significant fraction of the computing power during rendering, and a re-implementation of the network structure in the PBRT C++ process would significantly boost performance.

The progressive refinement algorithm used in OSR requires no preprocessing and very little additional memory to run. However in some scenarios predetermined and optimized radiance interpolation points can be a more effective solution, and implementing the possibility to choose between the two strategies would be a useful future extension.

To solve the temporal stability issue, a persistent caching system for radiance maps can be implemented, allowing animations to be rendered at faster rates and with less temporal noise caused by flickering of radiance maps.

An interesting extension of OSR would be the use of Metropolis Light Transport (MLT) as underlying rendering algorithm instead of Path Tracing. Although each Metropolis sample is more expensive than a Path sample, the ability of MLT in solving difficult light paths could boost OSR's ability to resolve indirect illumination, especially when light travels through refractive media before entering the scene. The complementing direct illumination pass can also be achieved using MLT by setting an appropriate maximum depth level.

9 Conclusion

We present *One Shot Radiance*, a novel ray tracing method that uses Convolutional Autoencoders to obtain high performance and accurate global illumination effects. All paths beyond the first bounce are estimated using a Neural Network that uses low quality One-shot, distance and normal maps. We show that our method achieves competitive performance while being able to produce higher quality images than same-time Path Tracing, with a significantly smaller amount of noise. Our method supports a wide range of material types and does not need offline precomputation or per-scene training. The re-integration of directly illuminated component using a separate integrator maintains fine details and high resolution shadows in the scene without introducing a significant amount of noise. OSR produces high quality images with low noise in a very short amount of time, and is suitable for all applications that do not require physical correctness.

The Blender Add-on for PBRT-OSR and the Graphical user interface make the software accessible and ready for use, by providing a workflow that is well integrated in Blender and that does not require any programming knowledge.

9.1 Challenges and Lessons Learned

The implementation of OSR has encountered a number of challenges. The choice of PBRTv3 as the base rendering engine was appropriate. PBRT was written with an educational intention, as it is the implementation of the omonimous book. Despite this, working with the C++ source code of the engine required a long period of time to start understanding how it works and how it can be extended with a new integrator. The overall code of the project is well written, but there is a tendency of leaving large parts of the source code uncommented because most of the explanation is in the book. While this is fine for readers that explore the inner workings of a physically based rendering engine, it slows down development considerably, as often variable names are short and mysterious, and geometrical conventions are not always clear.

It could have been a more interesting choice to write OSR based not on an educational and research oriented rendering engine, but on a production quality one that is effectively used by creators and 3D artists. Production rendering engines such as LuxCoreRender [12] have a much stronger focus on real world performance and usability, and their development is guided by what users need from the software.

Interestingly, LuxCoreRender derived from LuxRender, which in turn was originally based on PBRT, but over the years the project has diverged very significantly from its parent codebase, and can be safely considered as a completely new rendering engine now. Developing OSR on LuxCoreRender would have been a different experience, posing a new set of challenges and difficulties that could have been paid off by the excellent integration with Blender and by he possibility of having our extension accepted in the main project branch.

One of the primary reasons that drove us to develop the Blender Plugin was in fact the very poor real world usability of PBRTv3. With the exception of the set of example scenes from the official website, there is a strong lack of PBRT-compatible projects on the web. Converting a Blender scene for PBRTv3 required a large amount of manual adjustments and trial and error, and the small number of available scenes was becoming a limiting factor for our neural network's dataset. Developing our own Blender Plugin and exporter was an interesting challenge, and the compatibility of the plugin with the original PBRTv3 can be a great addition to the community of researchers that use PBRT for their projects. The Blender API is vast and powerful, and learning how to develop and integrate an Add-on is valuable experience.

The work on the Neural Network has taken a significant amount of time in the project. The lack of previous experience in Machine Learning has required a period of reading and documentation, while the simplicity and accessibility of PyTorch made the whole process reasonably smooth and enjoyable. As illustrated in Section 5.1, training the model has involved a number of failed attempts, but a lot was learned in the process and we are happy about our final results.

The development effort spent on the OSR project has been significant, with over 20000 lines of code written in C, C++, Python, Javascript, HTML and CSS. The final result is a working application that demonstrates a new application of Machine Learning to Computer Graphics that is usable in real-world scenarios to accelerate rendering of Global Illumination.

References

[1] Angularjs. https://angularjs.org/. Accessed: 03/06/2018.

[2] Blender. https://www.blender.org/. Accessed: 31/05/2018.

[3] Blender api. https://docs.blender.org/api/2.79/. Accessed: 03/06/2018.

[4] Chrome v8. https://developers.google.com/v8/. Accessed: 03/06/2018.

[5] Chromium. https://www.chromium.org/Home. Accessed: 03/06/2018.

[6] Electron. https://electronjs.org/. Accessed: 03/06/2018.

[7] Equirectangular projection. https://en.wikipedia.org/wiki/Equirectangular_projection. Accessed: 25/04/2018.

[8] Helmholtz reciprocity. https://en.wikipedia.org/wiki/Helmholtz_reciprocity. Accessed: 27/05/2018.

[9] High-resolution light probe image gallery. http://gl.ict.usc.edu/Data/HighResProbes/. Accessed: 25/04/2018.

[10] Iray performance tips. http://www.designimage.co.uk/iray-performance-tips-to-reduce-fireflies/. Accessed: 13/10/2017.

[11] Linear regression. https://en.wikipedia.org/wiki/Linear_regression. Accessed: 24/05/2018.

[12] Luxcorerender. https://luxcorerender.org/. Accessed: 08/06/2018.

[13] Luxrender portals. http://www.luxrender.net/wiki/Portals. Accessed: 29/12/2017.

[14] Luxrender volumes. http://www.luxrender.net/wiki/LuxRender_Volumes. Accessed: 13/10/2017.

[15] The mnist dataset of handwritten digits. http://yann.lecun.com/exdb/mnist/. Accessed: 30/05/2018.

[16] Node.js. https://nodejs.org/en/. Accessed: 03/06/2018.

[17] Peak signal-to-noise ratio. https://en.wikipedia.org/wiki/Peak_signal-to-noise_ratio. Accessed: 19/05/2018.

[18] Reducing noise. https://docs.blender.org/manual/en/dev/render/cycles/optimizations/reducing_noise.html. Accessed: 04/11/20107.

[19] Rendering resources. https://benedikt-bitterli.me/resources/. Accessed: 24/05/2018.

[20] Scenes for pbrt-v3. http://pbrt.org/scenes-v3.html. Accessed: 25/04/2018.

[21] Spectral rendering. https://en.wikipedia.org/wiki/Spectral_rendering. Accessed: 18/12/2017.

[22] Variance clamping. http://www.luxrender.net/forum/viewtopic.php?f=8&t=12358&p=119137. Accessed: 04/11/2017.

[23] W3css. https://www.w3schools.com/w3css/. Accessed: 03/06/2018.

[24] Wavefront obj format. http://www.martinreddy.net/gfx/3d/OBJ.spec. Accessed: 03/06/2018.

[25] What is machine learning? https://www.quora.com/What-is-machine-learning-4/answer/Crist%C3%B3bal-Esteban. Accessed: 24/05/2018.

[26] Steve Bako, Thijs Vogels, Brian McWilliams, Mark Meyer, Jan Novák, Alex Harvill, Pradeep Sen, Tony DeRose, and Fabrice Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. In *Proceedings of ACM SIGGRAPH 2017*. ACM, 2017.

[27] Kevin Beason. Global illumination in 99 lines of c++. http://www.kevinbeason.com/smallpt/. Accessed: 13/10/2017.

[28] David Burke, Abhijeet Ghosh, and Wolfgang Heidrich. Bidirectional importance sampling for direct illumination. *Rendering Techniques*, 5:147–156, 2005.

[29] Chakravarty R. Alla Chaitanya, Anton S. Kaplanyan, Christoph Schied, Marco Salvi, Aaron Lefohn, Derek Nowrouzezahrai, and Timo Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. In *ACM SIGGRAPH 2017*, 2017.

[30] Ken Dahm and Alexander Keller. Learning light transport the reinforced way. *arXiv preprint arXiv:1701.07403*, 2017.

[31] Paul Debevec. Image-based lighting. In *ACM SIGGRAPH 2005 Courses*, page 3. ACM, 2005.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *arXiv:1512.03385*, 2015.

[33] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[34] Henrik Wann Jensen. Global illumination using photon maps. In *Rendering Techniques' 96*, pages 21–30. Springer, 1996.

[35] James T Kajiya. The rendering equation. In *ACM Siggraph Computer Graphics*, volume 20, pages 143–150. ACM, 1986.

[36] Nima Khademi Kalantari, Steve Bako, and Pradeep Sen. A machine learning approach for filtering monte carlo noise. In *ACM Transactions on Graphics 34*, 2015.

[37] Simon Kallweit, Thomas Müller, Brian McWilliams, Markus H. Gross, and Jan Novák. Deep scattering: Rendering atmospheric clouds with radiance-predicting neural networks. *CoRR*, abs/1709.05418, 2017.

[38] Kanita Karađuzović-Hadžiabdić, Jasminka Hasić Telalović, and Rafał K Mantiuk. Assessment of multi-exposure hdr image deghosting methods. In *Computers & Graphics 63*, 2017.

[39] Alexander Keller. Instant radiosity. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, 1997.

[40] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *arXiv preprint arXiv:1412.6980*, 2014.

[41] Günter Klambauer, Thomas Unterthiner, Andreas Mayr, and Sepp Hochreiter. Self-normalizing neural networks. In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 2017.

[42] Claude Knaus and Matthias Zwicker. Progressive photon mapping: A probabilistic approach. *ACM Transactions on Graphics (TOG)*, 30(3):25, 2011.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[44] William H Kruskal and W Allen Wallis. Use of ranks in one-criterion variance analysis. In *Journal of the American statistical Association 47*, 1952.

[45] Eric P Lafortune and Yves D Willems. Bi-directional path tracing. 1993.

[46] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE vol 86*, 1998.

[47] Yann LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, 1998.

[48] Po-Ming Lee and Hung-Yi Chen. Adjustable gamma correction circuit for tft lcd. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 780–783. IEEE, 2005.

[49] Zichen Liu and Su-Lin Lee. Machine learning for filtering monte carlo noise in ray traced images. Imperial College London, 2017.

[50] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning*, 2013.

[51] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. In *arXiv preprint arXiv:1411.1784*, 2014.

[52] Oliver Nalbach, Elena Arabadzhiyska, Dushyant Mehta, H-P Seidel, and Tobias Ritschel. Deep shading: Convolutional neural networks for screen-space shading. In *Computer Graphics Forum*, 2017.

[53] Matt Pharr and Greg Humphreys. *Physically Based Rendering: From Theory To Implementation*. Morgan Kaufmann, 2nd edition, 2010.

[54] Bui Tuong Phong. Illumination for computer generated pictures. In *Communications of ACM 18*, 1975.

[55] Lee Prangnell. Visible light-based human visual system conceptual model. In *CoRR abs/1609.04830*, 2016.

[56] Martin Riedmiller and Heinrich Braun. Rprop - a fast adaptive learning algorithm. In *Proceedings of ISCIS VII*, 1992.

[57] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, 2015.

[58] Fabrice Rousselle, Claude Knaus, and Matthias Zwicker. Adaptive sampling and reconstruction using greedy error minimization. In *ACM Transactions on Graphics 30*, 2011.

[59] Fabien Sanglard. Decyphering the business card raytracer. `http://fabiensanglard.net/rayTracing_back_of_business_card`. Accessed: 13/10/2017.

[60] Pradeep Sen, Billy Chen, Gaurav Garg, Stephen R Marschner, Mark Horowitz, Marc Levoy, and Hendrik Lensch. Dual photography. In *ACM Transactions on Graphics 24*, 2005.

[61] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. In *The Journal of Machine Learning Research vol 15*, 2014.

[62] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, et al. Going deeper with convolutions. In *CVPR*, 2015.

[63] Manu Mathew Thomas and Angus G Forbes. Deep illumination: Approximating dynamic global illumination with generative adversarial network. In *arXiv preprint arXiv:1710.09834*, 2017.

[64] Eric Veach. Russian roulette and splitting. In *Robust Monte Carlo Methods for Light Transport Simulation*, page 67, 1997.

[65] Eric Veach and Leonidas Guibas. Bidirectional estimators for light transport. In *Photorealistic Rendering Techniques*, pages 145–167. Springer, 1995.

[66] Eric Veach and Leonidas J Guibas. Optimally combining sampling techniques for monte carlo rendering. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 419–428. ACM, 1995.

[67] Eric Veach and Leonidas J Guibas. Metropolis light transport. In *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, pages 65–76. ACM Press/Addison-Wesley Publishing Co., 1997.

[68] Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th international conference on Machine learning*, 2008.

[69] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. In *IEEE transactions on image processing 13*, 2004.

[70] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, 2014.

Appendices

A Additional Comparisons

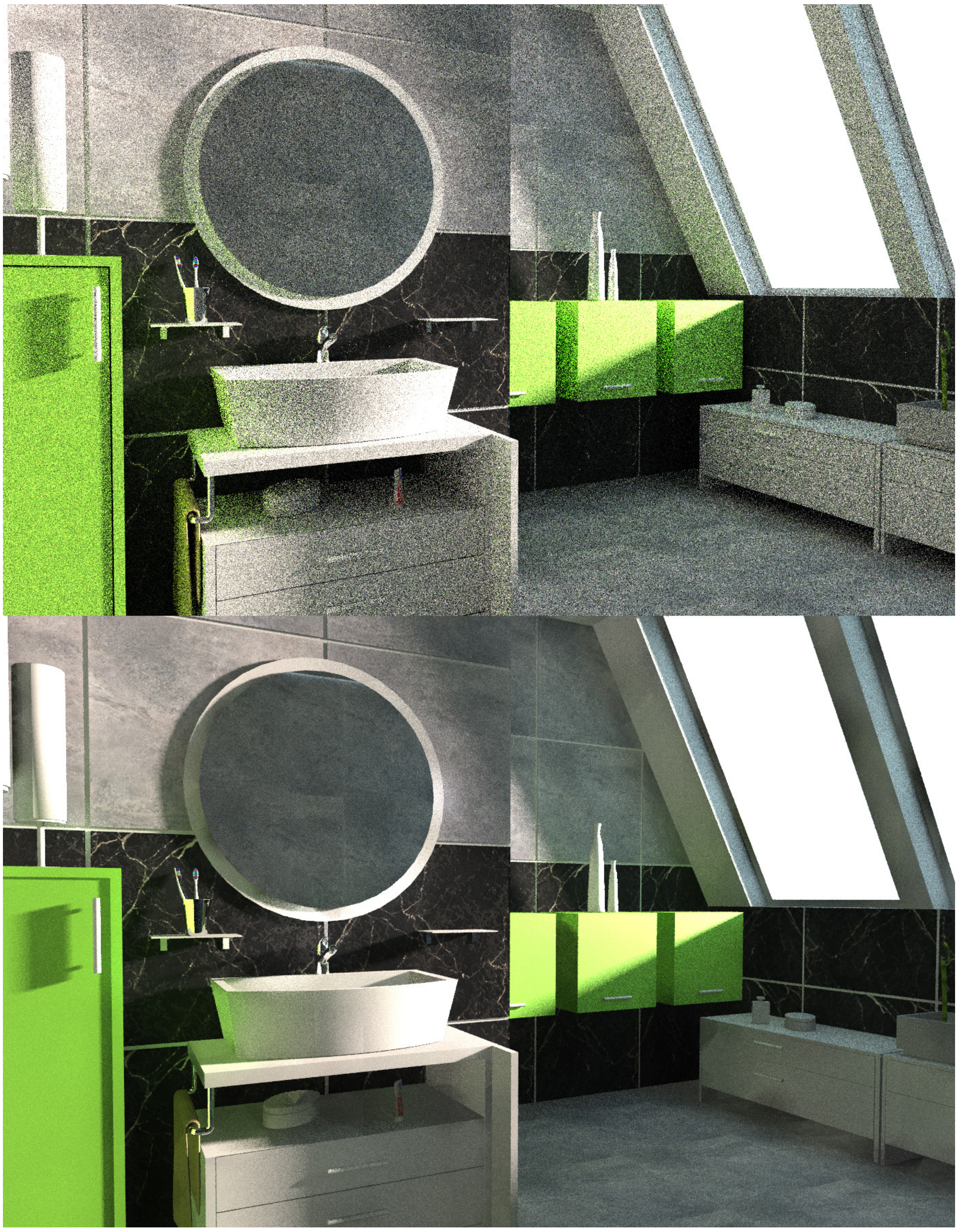


Figure 78: Bathroom green. Original scene by *cenobi* (`http://www.blendswap.com/blends/view/52486`). Top: Path 56 seconds, Bottom: OSR 57 seconds.

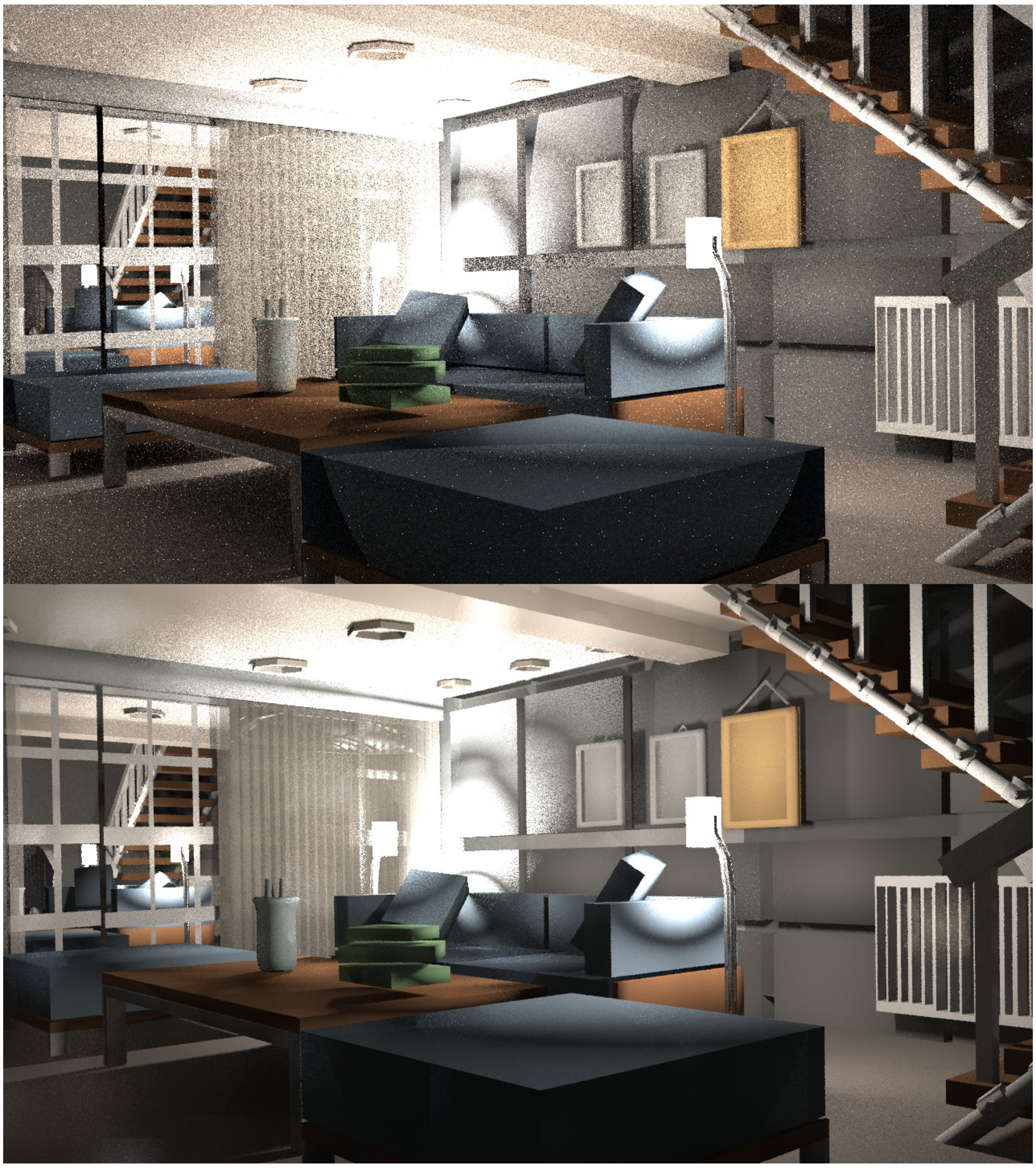


Figure 79: Full house. Original scene by *alecfara* (https://www.blendswap.com/blends/view/15938). Top: Path 52 seconds, Bottom: OSR 53 seconds.

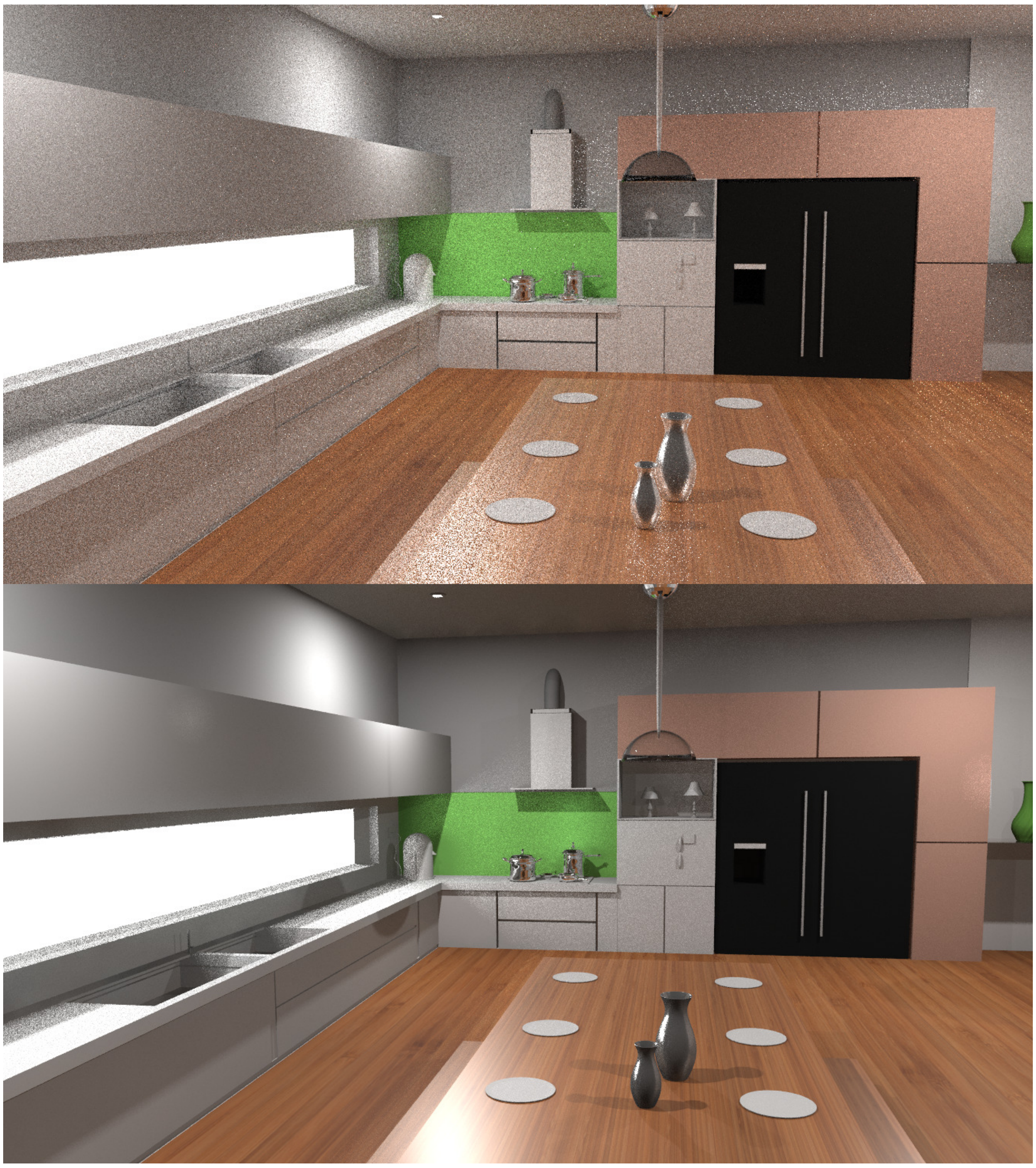


Figure 80: Kitchen. Original scene by *lukethsdhafdf* (http://www.blendswap.com/blends/view/48418). Top: Path 85 seconds, Bottom: OSR 90 seconds.

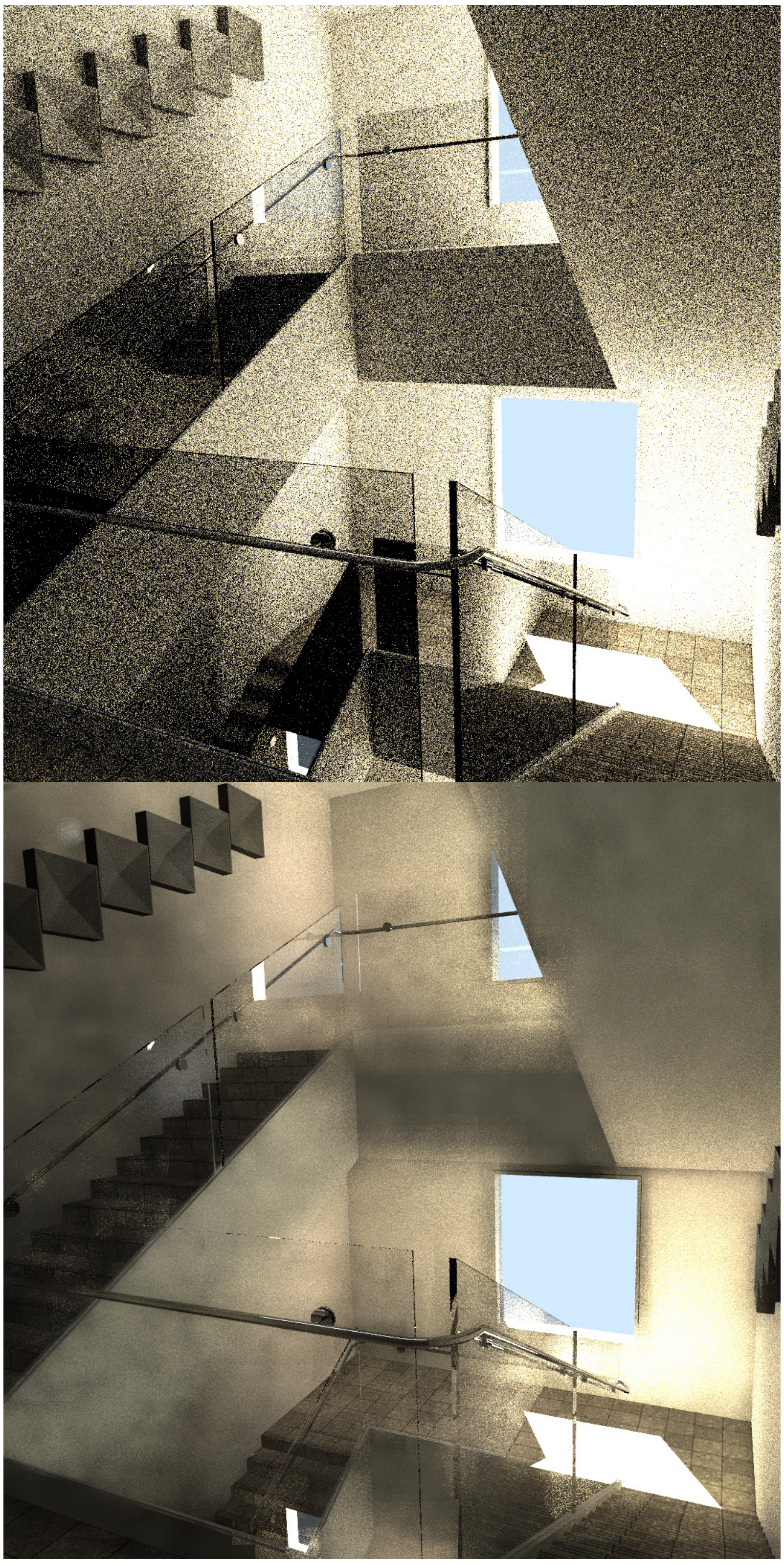


Figure 81: Stairway. Original scene by *Anticreep* (http://www.blendswap.com/blends/view/21326). Top: Path 211 seconds, Bottom: OSR 200 seconds. This scene has extremely difficult lighting. Path struggles at obtaining a clear image, while OSR produces far less noise at the cost of bias and softness.

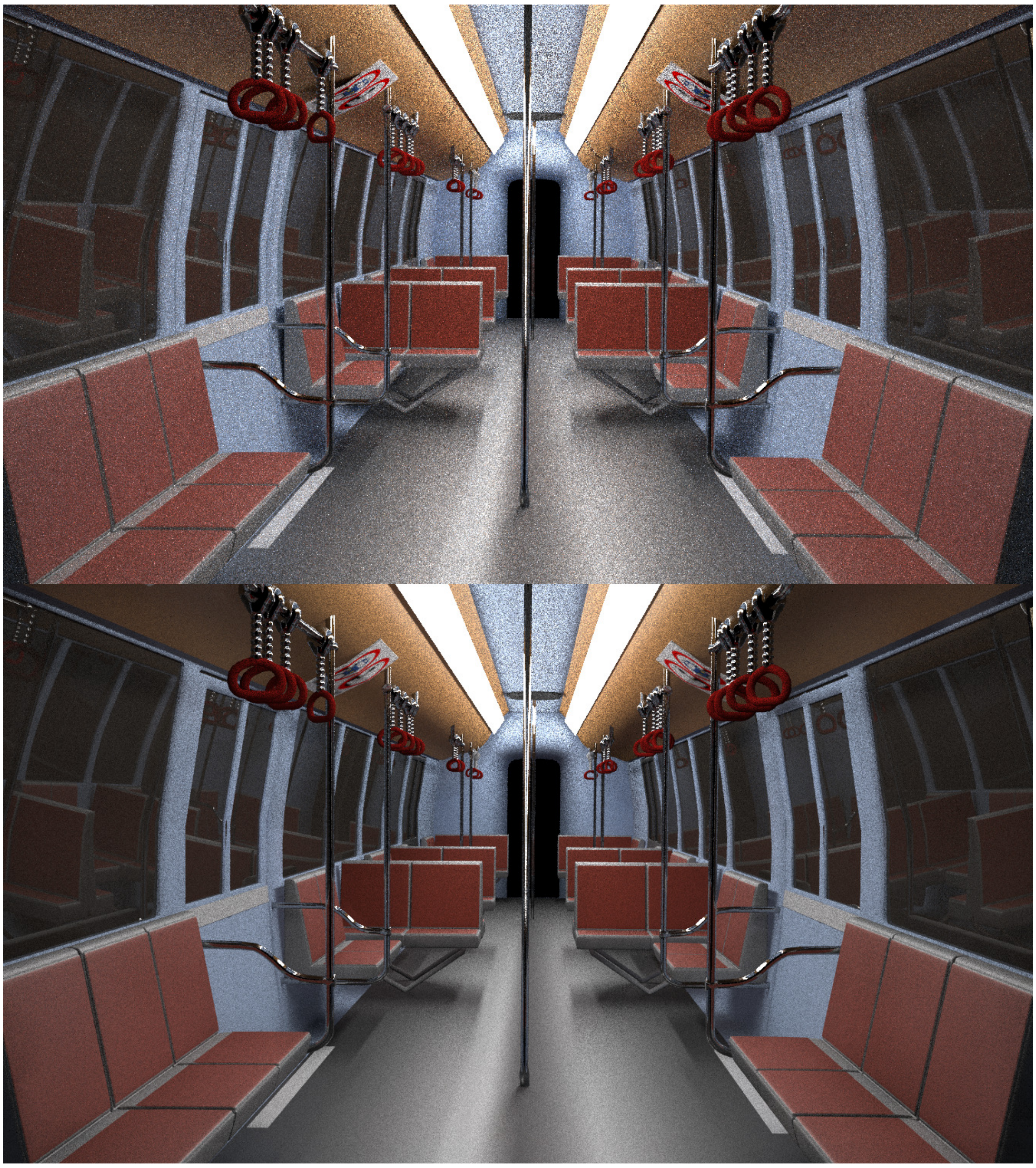


Figure 82: Stairway. Original scene by *muhtesemozcinar* (`http://www.blendswap.com/blends/view/72190`). Top: Path 90 seconds, Bottom: OSR 60 seconds.

B Important Code

We present some of the most important pieces of code part of the project. Please note that references in the code to *IISPT* or *IILE* are due to name changes of the project that have not been updated in the code.

The core of the PBRTv3 extension is the new Integrator that runs the OSR code. The integrator is based on the same interface as all other integrators in PBRT, and implements the following methods:

```
void Preprocess(const Scene &scene);

Spectrum Li(const RayDifferential &r,
             const Scene &scene,
             Sampler &sampler,
             MemoryArena &arena,
             int depth
             ) const;

void Li_reference(const RayDifferential &ray,
                           const Scene &scene,
                           Point2i pixel
                           ) const;

void Render(const Scene &scene);

void render_normal_2(const Scene &scene);

void render_reference(const Scene &scene);
```

`Preprocess` has an empty body, as OSR does not require any preprocessing. `Li` is not used in the code because we implement a custom render loop. The main `Render` function differentiates the two separate cases of Reference generation (Section 5.3.3) and normal rendering.

In the Reference generation main loop a regular grid on the image film is used to determine the viewpoints for the example sets:

```
for (int px_y = 0; px_y < sampleExtent.y; px_y += reference_tile_interval_y) {
   for (int px_x = 0; px_x < sampleExtent.x; px_x += reference_tile_interval_x)
       ↪ {

      ref_idx++;
      if ((ref_idx % reference_control_mod) != reference_control_match) {
         // This pixel is not a job of the current process
         continue;
      }

      CameraSample current_sample;
      current_sample.pFilm = Point2f(px_x, px_y);
      current_sample.time = 0;

      RayDifferential ray;
      Float rayWeight = camera->GenerateRayDifferential(current_sample, &ray);
      // It's a single pass per pixel, so we don't scale the differential
      ray.ScaleDifferentials(1);
      // The Li method, in reference mode, will automatically save the reference
          ↪  images
```

```
        // to the out/ directory
        Li_reference(ray, scene, Point2i(px_x, px_y));
    }
}
```

In `Li_reference`, the first intersection point is computed and used to place the auxiliary camera for the new viewpoint:

```
// Find closest ray intersection or return background radiance
SurfaceInteraction isect;
if (!scene.Intersect(ray, &isect)) {
    return;
}

// Invert normal if the surface's normal was pointing inwards
Normal3f surfNormal = isect.n;
if (Dot(Vector3f(isect.n.x, isect.n.y, isect.n.z), Vector3f(ray.d.x, ray.d.y, ray
    ↪ .d.z)) > 0.0) {
    surfNormal = Normal3f(-isect.n.x, -isect.n.y, -isect.n.z);
}

// auxRay is centered at the intersection point, and points towards the
    ↪ intersection
// surface normal
Ray auxRay = isect.SpawnRay(Vector3f(surfNormal));

// testCamera is used for the hemispheric rendering
std::string reference_d_name = generate_reference_name("d", pixel, ".pfm");
std::shared_ptr<HemisphericCamera> auxCamera (
            CreateHemisphericCamera(
                PbrtOptions.iisptHemiSize,
                PbrtOptions.iisptHemiSize,
                dcamera->medium,
                auxRay.o,
                auxRay.d,
                reference_d_name
                )
            );
```

The rendering using the auxiliary camera proceeds as a standard render job. The Auxiliary integrator is based on a standard Path tracer, with modifications to exclude direct radiance from light sources.

The normal rendering loop is fully optimized for multithreading, and relies on a Thread Pool and an additional class named the Render Runner:

```
// Create thread pool
unsigned noCpus = iile::cpusCountFull();
// noCpus = 1;
ThreadPool threadPool (noCpus);
std::vector<std::future<void>> futures;

// Start threads
for (int i = 0; i < noCpus; i++) {
    std::shared_ptr<IisptNnConnector> nnConnector =
            iile::NnConnectorManager::getInstance().getInstance().get(i);
    futures.push_back(threadPool.enqueue([i, schedule_monitor,
        ↪ film_monitor_indirect, film_monitor_direct, this, &scene, nnConnector
        ↪ ]() {
        std::shared_ptr<IisptRenderRunner> runner (
                    new IisptRenderRunner(
                        schedule_monitor,
                        film_monitor_indirect,
                        film_monitor_direct,
                        camera,
                        dcamera,
                        sampler,
                        i,
                        camera->film->GetSampleBounds(),
                        nnConnector
                        )
                    );
        if (i % 2 == 0) {
            runner->run_direct(scene);
            runner->run(scene);
        } else {
            runner->run(scene);
            runner->run_direct(scene);
        }
    }));
}

// Wait for threads to finish
for (int i = 0; i < noCpus; i++) {
    futures[i].get();
}
```

The indirect illumination render loop in each Runner obtains a task from the Task Manager and computes One Shot radiance maps, Normals and Distances on the local grid. The films are sent to the Neural Network process, and the results are normalized back to match the average intensity of the One Shot map.

```
        int tile_x = sm_task.x0;
        int tile_y = sm_task.y0;
        while (1) {
            IisptPoint2i hemi_key;
            hemi_key.x = tile_x;
            hemi_key.y = tile_y;
            Point2i pixel (tile_x, tile_y);
            sampler_next_pixel();
            CameraSample camera_sample =
                    sampler->GetCameraSample(pixel);
            RayDifferential r;
            main_camera->GenerateRayDifferential(
                        camera_sample,
                        &r
                        );
            r.ScaleDifferentials(1.0);
            SurfaceInteraction isect;
            Spectrum beta;
            Spectrum background;
            RayDifferential ray;
            Spectrum area_out;
```

```
bool intersection_found = find_intersection(
        r,
        scene,
        arena,
        &isect,
        &ray,
        &beta,
        &background,
        &area_out
        );
if (!intersection_found || beta.y() <= 0.0) {
    hemi_points[hemi_key] = nullptr;
} else {
    Normal3f surface_normal = isect.n;
    Vector3f sf_norm_vec = Vector3f(isect.n.x, isect.n.y, isect.n.z);
    Vector3f ray_vec = Vector3f(ray.d.x, ray.d.y, ray.d.z);
    if (Dot(sf_norm_vec, ray_vec) > 0.0) {
        surface_normal = Normal3f(
                -isect.n.x,
                -isect.n.y,
                -isect.n.z
                );
    }
    Ray aux_ray = isect.SpawnRay(Vector3f(surface_normal));
    std::unique_ptr<HemisphericCamera> aux_camera (
            CreateHemisphericCamera(
                PbrtOptions.iisptHemiSize,
                PbrtOptions.iisptHemiSize,
                dcamera->medium,
                aux_ray.o,
                aux_ray.d,
                std::string("/tmp/null")
                )
            );
    d_integrator->RenderView(
            scene,
            aux_camera.get()
            );
    std::unique_ptr<IntensityFilm> aux_intensity =
            d_integrator->get_intensity_film(aux_camera.get());
    NormalFilm* aux_normals =
            d_integrator->get_normal_film();
    DistanceFilm* aux_distance =
            d_integrator->get_distance_film();
    float intensityMean = normalizeMapsDownstream(
            aux_intensity.get(),
            aux_normals,
            aux_distance
            );
    int communicate_status = -1;
    std::shared_ptr<IntensityFilm> nn_film =
            nn_connector->communicate(
                aux_intensity.get(),
                aux_distance,
                aux_normals,
                communicate_status
```

```
                    );
            transformMapsUpstream(nn_film.get(), intensityMean);
            if (communicate_status) {
                std::cerr << "iisptrenderrunner.cpp: Thread " << thread_no << "
                    ↪ " << "NN communication issue" << std::endl;
                raise(SIGKILL);
            }
            aux_camera->set_nn_film(nn_film);
            hemi_points[hemi_key] = std::move(aux_camera);
        }
        bool advance_tile_y = false;
        if (tile_x == sm_task.x1 - 1) {
            tile_x = sm_task.x0;
            advance_tile_y = true;
        } else if (tile_x >= sm_task.x1) {
            std::raise(SIGKILL);
        } else {
            tile_x = std::min(
                    tile_x + sm_task.tilesize,
                    sm_task.x1 - 1
                    );
        }
        if (advance_tile_y) {
            if (tile_y == sm_task.y1 - 1) {
                break;
            } else {
                tile_y = std::min(
                        tile_y + sm_task.tilesize,
                        sm_task.y1 - 1
                        );
            }
        }
    }
```

The evaluation code renders pixel values using the cached radiance maps in the previous step.

```
for (int fy = sm_task.y0; fy < sm_task.y1; fy++) {
    for (int fx = sm_task.x0; fx < sm_task.x1; fx++) {
        Point2i f_pixel (fx, fy);
        Point2i neigh_s (
                fx - (iispt::positiveModulo(fx - sm_task.x0, sm_task.tilesize)),
                fy - (iispt::positiveModulo(fy - sm_task.y0, sm_task.tilesize))
                );
        Point2i neigh_e (
                std::min(
                    neigh_s.x + sm_task.tilesize,
                    sm_task.x1 - 1
                    ),
                std::min(
                    neigh_s.y + sm_task.tilesize,
                    sm_task.y1 - 1
                    )
                );
        Point2i neigh_r (
                neigh_e.x,
                neigh_s.y
                );
```

```
      Point2i neigh_b (
               neigh_s.x,
               neigh_e.y
               );
   }
}
```

Weighting of each of the four maps is computed using the distance and normals rules (Section 4.3):

```
static float weighting_distance_positions(
      Point2i a,
      Point2i b,
      float tile_distance
      )
{
   float pdist = points_distance(a, b);
   float res;
   if (tile_distance != 0.0) {
      res = pdist / tile_distance;
   } else {
      res = pdist;
   }
   if (res < 0.0) {
      return 0.0;
   }
   if (res > 1.0) {
      return 1.0;
   }
   return res;
}

static float weighting_distance_normals(
      Vector3f a,
      Vector3f b
      )
{
   // Check for invalid vectors
   float al = a.Length();
   float bl = b.Length();
   if (al <= 0.0 || bl <= 0.0) {
      return 1.0;
   }

   a = a / al;
   b = b / bl;

   float dt = Dot(a, b);
   if (dt < 0.0) {
      return 1.0;
   } else {
      return 1.0 - dt;
   }
}
```

The weighted hemispheres are sampled using weighted probabilities.

```
Spectrum IisptRenderRunner::sample_hemisphere(
        const Interaction &it,
        int len,
        float* weights,
        HemisphericCamera** cameras
        )
{
    Spectrum L(0.f);

    int samples_taken = 0;

    for (int i = 0; i < len; i++) {
        HemisphericCamera* a_camera = cameras[i];
        float a_weight = weights[i];

        // Attempt HEMISPHERIC_IMPORTANCE_SAMPLES to sample this camera
        // The expected number of samples across all the cameras will be
        // HEMISPHERIC_IMPORTANCE_SAMPLES
        for (int j = 0; j < HEMISPHERIC_IMPORTANCE_SAMPLES; j++) {
            float rr = rng->uniform_float();
            if (rr < a_weight) {
                samples_taken++;
                if (a_camera != NULL) {
                    int rx = rng->uniform_uint32(PbrtOptions.iisptHemiSize);
                    int ry = rng->uniform_uint32(PbrtOptions.iisptHemiSize);
                    L += estimate_direct(it, rx, ry, a_camera, rng.get());
                }
            }
        }
    }

    if (samples_taken > 0) {
        return L / samples_taken;
    } else {
        return Spectrum(0.0);
    }
}
```

The Task Monitor is the shared object that controls the structure and size of the indirect illumination tasks.

```
IisptScheduleMonitorTask IisptScheduleMonitor::next_task() {

    std::unique_lock<std::mutex> lock (mutex);

    int effective_radius = std::floor(current_radius);
    if (effective_radius < 1) {
        effective_radius = 1;
    }

    int task_size = effective_radius * NUMBER_TILES;

    // Form the result
    // The current nextx and nexty are valid starting coordinates
    IisptScheduleMonitorTask res;
    res.x0 = nextx;
    res.y0 = nexty;
    res.x1 = std::min(res.x0 + task_size, bounds.pMax.x);
    res.y1 = std::min(res.y0 + task_size, bounds.pMax.y);
    res.tilesize = effective_radius;
    res.pass = pass;
    res.taskNumber = taskNumber++;

    // Advance to the next tile

    nextx += task_size;
    if (nextx >= bounds.pMax.x) {
        // Reset x, advance y
        nextx = bounds.pMin.x;
        nexty += task_size;
    }

    if (nexty >= bounds.pMax.y) {
        // Reset y, advance radius
        nexty = bounds.pMin.y;
        current_radius *= update_multiplier;
        pass++;
    }

    return res;
}
```

The core of the Neural Network is the model, defined in PyTorch.

```
K = 64

class IISPTNet(torch.nn.Module):

    def __init__(self):
        super(IISPTNet, self).__init__()

        # Input depth:
        # Intensity RGB
        # Normals XYZ
        # Distance Z
        # 7 channels

        # Output depth:
        # Intensity RGB
        # 3 channels

        # In 32x32
        self.encoder0 = nn.Sequential(
            nn.Conv2d(7, K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(K, K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2)
        )
        # Out 32x32

        # In 32x32
        self.encoder1 = nn.Sequential(
            nn.MaxPool2d(2),
            nn.Conv2d(K, 2*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(2*K),
            nn.Conv2d(2*K, 2*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2)
        )
        # Out 16x16

        # In 16x16
        self.encoder2 = nn.Sequential(
            nn.MaxPool2d(2),
            nn.Conv2d(2*K, 4*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(4*K),
            nn.Conv2d(4*K, 4*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2)
        )
        # Out 8x8

        # In 8x8 -> 4x4
        self.encoder3 = nn.Sequential(
            nn.MaxPool2d(2),
            nn.Conv2d(4*K, 8*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(8*K),
            nn.Conv2d(8*K, 4*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Upsample(scale_factor=2, mode="bilinear")
        )
        # Out 8x8

        # In 8x8 + skip from encoder2
        self.decoder0 = nn.Sequential(
            nn.ConvTranspose2d(8*K, 4*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(4*K),
            nn.ConvTranspose2d(4*K, 2*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Upsample(scale_factor=2, mode="bilinear"),
        )
        # Out 16x16

        # In 16x16 + skip from encoder1
        self.decoder1 = nn.Sequential(
            nn.ConvTranspose2d(4*K, 2*K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.BatchNorm2d(2*K),
            nn.ConvTranspose2d(2*K, K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Upsample(scale_factor=2, mode="bilinear"),
        )
        # Out 32x32

        # In 32x32 + skip from encoder0
        self.decoder2 = nn.Sequential(
            nn.ConvTranspose2d(2*K, K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.ConvTranspose2d(K, K, 3, stride=1, padding=1),
            nn.LeakyReLU(0.2),
            nn.Conv2d(K, 3, 1, stride=1, padding=0),
            nn.ReLU()
        )
        # Out 32x32

    def forward(self, x):
        x0 = self.encoder0(x)
        x1 = self.encoder1(x0)
        x2 = self.encoder2(x1)
        x3 = self.encoder3(x2)

        x4 = self.decoder0(torch.cat((x3, x2), 1))
        x5 = self.decoder1(torch.cat((x4, x1), 1))
        x6 = self.decoder2(torch.cat((x5, x0), 1))

        return x6
```