Towards Realtime Monte Carlo Path Tracing on the Single CPU

for Accessible Photorealistic Virtual Reality

George Tang

Abstract

Specialized, high-end graphics processing units (GPU) can be used to achieve realtime Monte Carlo path tracing for virtual reality (VR). However, path tracing GPUs are very expensive and need to be individually purchased and installed to the computer, making them a burden for commercial and personal VR. To address this, we develop an AI-accelerated path tracer specifically optimized for devices with a single central processing unit (CPU) that achieves photorealistic output while functioning at interactive rates. Our architecture utilizes a KD-Tree acceleration structure with a new build and search algorithm that improves runtime by up to 40% and reduces memory usage by around 30%. We propose a two-component post-processing step that uses lightweight convolutional neural networks to first filter Monte Carlo noise from and then perform superresolution on renderings to reconstruct high-quality images from low-quality images rendered using extremely low ray budgets. To account for the reduction in trainable parameters and the great variety of scene geometry and lighting schemes, we propose Precision Graphics, where each scene has its own reconstruction network tailored to it. We apply Precision Graphics to our denoising network, dramatically increasing reconstruction quality. The superresolution network preserves image sharpness and visual effects better than interpolation methods even without Precision Graphics. The hybrid system functions at 1000x-5000x faster than standard path tracers while maintaining photorealistic output, setting a direction for realtime CPU path tracing for VR.

Keywords: virtual reality, Monte Carlo path tracing, CPU, KD-Tree, interactive rendering, image denoising, image superresolution, Precision Graphics, convolutional neural networks

1 Introduction

With the increase in computational power of computers, ray and path tracing have become the ideal rendering algorithms for accurate simulation of light transport and visual effects. In ray tracing, for every pixel in the image, primary rays are cast from the ray source to the pixel and into the scene. The ray bounces among the objects, eventually exiting the scene or reaching the set limit on the number of bounces. The information gathered by the ray is used to determine the value of the pixel (see Figure 1). Although this method produces very realistic shadows, reflections, and refractions, it does not take into account indirect lighting. The modern solution is the global illumination algorithm, implemented via Monte Carlo path tracing. The idea is at every ray-object intersection, we estimate indirect lighting using Monte Carlo integration. This can lead to thousands of sampling rays being cast per intersection, making the operation extremely expensive. A single movie frame rendered using path tracing can take hours to render on a CPU [23].

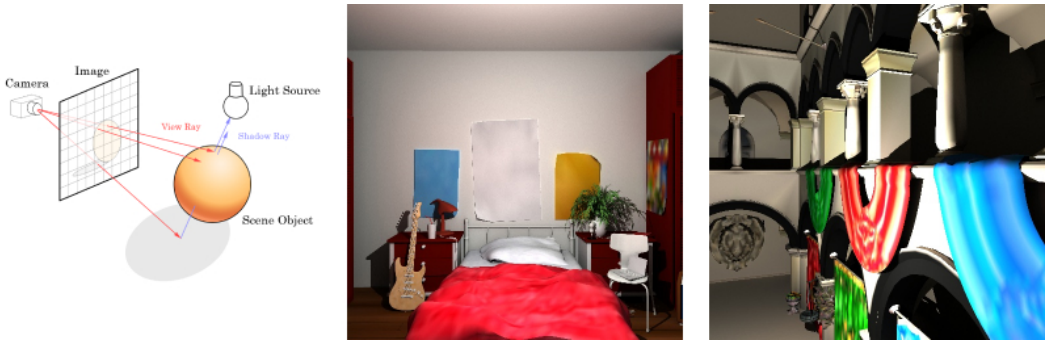

Figure 1: Left: Raytracing component of path tracing algorithm [19] Middle and Right: Path tracing can produce realistic lighting and high-quality images.

Many applications of virtual reality, from entertainment to medical rehabilitation and personnel training, require a high degree of photorealism to be immersive and effective [1, 7, 22]. Very recently, there has been a push towards path tracing on VR platforms. However, currently, even with the best path tracers and AI-accelerators, realtime path tracing is only possible on a high-end GPU or computing clusters [20, 4], both of which are inaccessible to the general public. Our goal is to work towards realtime path tracing for the single CPU, so anyone with a computer and cheap headset can experience movie-grade VR. We phrase the task as an optimization problem, with the following being minimized while the quality of rendered images remains comparable to that produced by standard path tracing:

- Cost of each ray cast
- Number of rays cast

The first problem has been actively studied since the inception of ray and path tracing. In naive ray and path tracing, to determine the nearest ray-object intersection, we have to iterate through every object in the scene and test if it intersects with the ray, resulting in a time complexity of O(N) per ray cast, where N is the number of objects. In scenes composed of hundreds of thousands or millions of triangles, this is extremely inefficient. However, notice we are testing intersections with triangles far away from the ray's path. Instead, we can spatially partition the triangles with an acceleration structure and traverse it to efficiently search ray-object intersections. The best

performing acceleration structure is the KD-Tree [10], a binary space partitioning tree with axis-aligned splitting planes. We implement a modification to the KD-Tree that increases traversal speed while reducing memory usage. In addition, we implement many other optimizations to make our path tracer comparable to industrial options.

The second problem is ill-posed. A reduction in the number of rays cast will lead to low-quality renderings. Specifically, if we reduce the number of sampling rays cast in the global illumination algorithm, the renderings will be noisy. We can also reduce the number of primary rays cast by rendering a smaller image and then increase its size (superresolution). The standard approach for superresolution in rendering is interpolation, but this results in a loss of image features such as fine edges and textures and decreases sharpness. Moreover, any post-processing method must be significantly faster than current post-processing methods as we are constrained to a single CPU.

Fortunately, there has been much work lately in the field of image denoising and superresolution using deep convolutional neural networks (CNN). Inspired by the concepts they laid out, we propose a set of much smaller, but faster networks with many novel features to address these challenges and satisfy the constraints unique to the single CPU. Specifically, we propose two novel concepts:

- Precision Graphics: autoencoder with skip connections specializes in filtering Monte Carlo noise from low-sample renderings on a scene-by-scene basis
- Interpolate denoised renderings then enhance image features and sharpness with superresolution CNN

Put together, these optimizations will speed up path tracing to where we can render high-quality high-poly scenes interactively (few seconds per frame) on the single CPU.

2 Related Work

There is extensive literature relevant to optimizing ray and path tracing. We begin our discussion with the implementation of and optimizations to the KD-Tree for path tracing. Next, we review research in deep-learning based image reconstruction and enhancement and analyze its potential to enhance low-quality renderings with low ray budgets. We divide the latter into two sections: Monte Carlo noise filtering and superresolution. Readers looking for more information on Monte Carlo path tracing can consult more detailed path tracing manuals [19, 21, 25].

KD-Tree Data Structure: A KD-Tree consists of a build and a traversal algorithm. We begin with the root node encompassing all objects. Naive KD-build recursively assigns each object in a node to the left child or right child or both based on the object's location relative to the node's axis-aligned splitting plane. As we move down the tree during construction, we cycle through the x, y, z-axes, and the axis-aligned splitting plane is determined to be located at the median of the axis components of the triangles' centroids. If the number of triangles is less than a certain threshold, then we stop splitting and denote the node as a leaf. Note only leaf nodes contain triangles. In the end, the axis-aligned splitting planes form bounding boxes that are used to determine if the ray intersects with any triangle in a group of triangles. Because a KD-Tree divides the search space in two each time, the maximum depth is $log(N)$. Because we sort the triangles by axis component to determine the splitting plane when we construct each node, the overall time complexity for KD-build is $O(Nlog^2(N))$ [26].

The traversal algorithm is also recursive. Starting with the root node, we determine if the ray intersects it. If so, then we determine if it intersects only the left child, only the right child, or both, and traverse in accordance (Figure 2). If we reach a leaf node, we test intersection with every

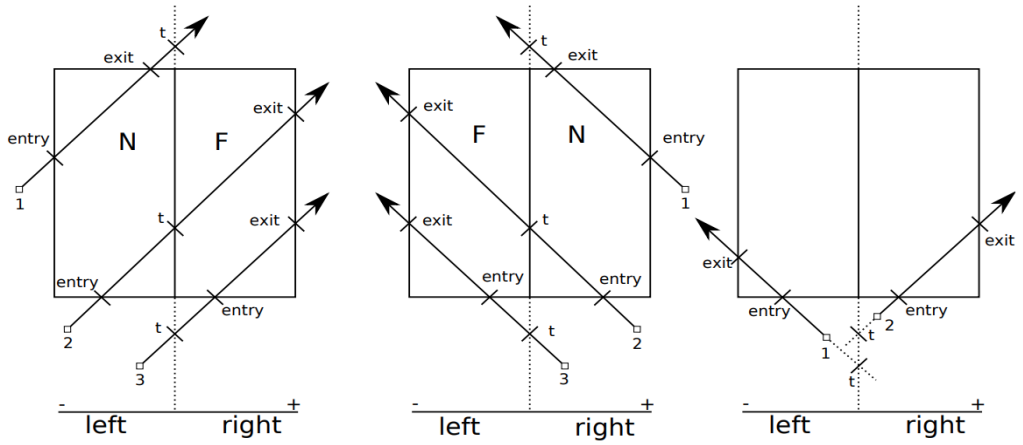

Figure 2: Casework for KD-traverse (see Algorithm 2). t refers the ray's distance to the splitting plane. If t is greater than the exit distance, then it only intersects the near node. If t is less, then it only intersects the far node. If t is in between, we have to search both nodes (from [9]).

object in the node. The time complexity for KD-traverse is $O(log(N))$. However, in practice it takes $O(Klog(N))$, where K is a constant (we found around 5-10), accounting for the duplication of triangles during KD-build when they overlap the splitting plane [9].

One optimization we can make is instead of determining the splitting value based on the median of the axis component of the centroid, we locally maximize the probability that the ray will have to search fewer triangles. Intuitively, this translates to maximizing empty space in the bounding boxes, and formally we define the Surface Area Heuristic, which has been proven to be robust in almost all conditions [26]. The cost to traverse a node can be estimated given a splitting configuration, and we want to select the min cost configuration:

$$Cost(V_L, V_R) \approx K_T + K_I\left(\frac{SA(V_L)}{SA(V)}T_L + \frac{SA(V_R)}{SA(V)}T_R\right)$$

where V_L, V_R, V are the left, right, and parent voxels (nodes), K_T is the cost of a bounding box intersection, K_I, the cost of a triangle intersection, $SA(V_L)$, $SA(V_R)$, $SA(V)$, the surface area of the left, right, and parent nodes, and T_L, T_R the number of triangles in the left and right nodes. We can also define an elegant splitting termination from SAH:

$$Terminate(V) = \begin{cases} true & \text{if } min(Cost(V_L, V_R)) > K_I T \\ false & \text{otherwise} \end{cases}$$

where T is the number of triangles in the parent node [26]. Other optimizations exist for the build and traversal to decrease their computation cost. However, for interactive rendering we do not need to improve the time complexity of our build algorithm as we only need to construct the tree once during pre-processing. For traversal, there are many algorithms such as neighbor links, ray coherence, and packet tracing yielding minor improvements [9], but all of them adhere to the constraint that only leaves contain triangles.

Algorithm 1 KD-Build with SAH

```
function FindPlaneSAH(V, axis)
    best = none
    minc = INF
    for cent[axis] in T do
        V_L, V_R ←Split(cent[axis])
        curr = Cost(V_L, V_R)
        if curr < minc then
            minc ← curr
            best ← cent[axis]
    return best

function KDBuild(V, depth)
    if Terminate(V) then
        return
    axis ← depth % 3
    p ←FindPlaneSAH(V, axis)
    V_L, V_R ←Split(p)
    T_L ← {t ∈ T | t ∩ V_L ≠ ∅}
    T_R ← {t ∈ T | t ∩ V_R ≠ ∅}
```

Algorithm 2 KD-Traverse

```
function KDTraverse(V_root, ray)
    stack.push(V_root)
    while stack not empty do
        (V, ent, ext) ← stack.pop
        while V not leaf do
            t ← DisToPlane(V.p)
            V_near, V_far ←Classify(V_L, V_R)
            if t ≥ exit or t < 0 then
                V ← V_near
                continue
            if t ≤ ent then
                V ← V_far
            else
                stack.push(far, t, ext)
                V ← V_near
                ext ← t
        if FindIntersection(V, ray) then
            return intersection
    return ∅
```

Deep Learning Based Image Reconstruction and Enhancement: CNNs are widely employed in images such as object detection, image classification, and image segmentation. CNN layers utilize learnable kernels that are convolved with the input to produce features. Operating on these features allow networks to specialize in tasks and increases performance [13]. Deep CNNs for image processing have recently gained much attention as it has been shown that they outperform virtually all state-of-the-art methods for denoising images corrupted with Gaussian noise, superresolution, and JPEG blocking [4]. The best performing architecture is a 30-layer encoder-decoder network with symmetrical skip connections trained on a very large dataset of photo crops. The encoder-decoder component allows the network to eliminate pixel corruption while skip connections enable the flow of information between the bottom and top layers, addressing vanishing gradient and increasing the effectiveness of end-to-end mapping [16].

Monte Carlo Noise Filtering: Removing noise in low-sample Monte Carlo renderings is related to Gaussian denoising, and the most prospective methods are machine-learning based. Kalantari et al. proposed a multi-layer perception that determines a non-linear mapping between the noisy and denoised renderings [11]. Bako et al. demonstrated that deep CNNs have tremendous potential in producing production-grade frames across different scene conditions [2]. Chaitanya et al. proposed a robust recurrent denoising autoencoder with skip connections based on Mao et al.'s work for Nvidia's RTX GPU path tracer (OptiX-AI). Their network modifies the encoder-decoder structure to include max pooling and upsampling, which results in about 10x increase in execution time compared with Mao et al.'s network and better results for spatially sparse noisy Monte Carlo renderings. The input consists of the low-sample rendering and auxiliary buffers (normals, depth, and roughness), which improves network performance [4]. However, these architectures are still too costly for interactive post-processing on the single CPU.

Image Superresolution: An approach to accelerate rendering in games is to render every other pixel and then interpolate (e.g. bicubic) the remaining pixels' values. However, this results in a

loss in sharpness, edges, textures, and minut features in the rendered image. Dong et al. showed CNNs have the ability to learn end-to-end mappings between low-resolution photo crops and their high-resolution counterparts. They proposed a lightweight architecture (SRCNN) that processes the bicubically interpolated image and achieves state-of-the-art results for peak signal to noise ratio (PSNR) [6]. Deeper architectures with residual blocks and skip connections achieve only marginally better results at the cost of computation time [14], making them infeasible for the single CPU. Thus, we propose a simple, lightweight architecture for our superresolution component as a post-processing to the bicubic interpolation for path tracing. To our knowledge, no research has applied machine learning methods to superresolution of computer renderings.

3 Path Tracing

To achieve interactive, photorealistic rendering, the path tracer must be able to efficiently handle complex scenes, complex lighting schemes, and visual effects in terms of speed and memory usage. For practical purposes, we limit the maximum depth to one for primary and secondary rays and two (one bounce) for visual effects.

3.1 Architecture

Objects are stored as polygon meshes, composed of polygons faces, which are broken up and stored as triangles. Each triangle is defined by vertices, and each vertex has a vertex normal and texture coordinates. Each mesh stores its texture map and triangles. Externally, these scenes are formatted as a Wavefront .obj file. To decrease memory usage, vertex, normal, and texture data are shared in every mesh, allowing a triangle to be defined by eight 32-bit ints (indices) as opposed to eight 64-bit doubles.

The rendering process begins with KD-Tree construction and initialization of camera and user movement settings. For every frame in interactive rendering, we first ray trace the diffuse map. When a ray intersects a triangle, shading, reflections, refractions, and texture are computed by interpolating the vertex normals and texture coordinates. Direct lighting at the intersection is computed by casting rays towards all light sources in the scene, and shadows result as a by-product. We implement an antialiasing option, accomplished by dividing each pixel into a lattice, casting a ray at each point, and then averaging the results. The center ray is used to determine the hit point coordinates, vertex normal, and color. In the second pass, we compute indirect lighting by casting rays from the hit point in the directions determined by Monte Carlo sampling and averaging the results. The overall lighting is a function of the surface properties, direct, and indirect lighting.

To speed up intersection tests, we employ the Möller-Trumbore Algorithm, which is presumed to be the fastest ray-triangle intersection algorithm [18], and a ray-box algorithm that is 30% faster than traditional methods for testing intersections with KD nodes [28]. In KD nodes, triangles are defined by two 32-bit ints: mesh id and triangle id. Finally, for every scene, we leave three parameters to be tuned, KT, KI, and the minimum number of nodes in a leaf. We include the last parameter choice to allow for more robust splitting termination because arithmetic on a modern computer is very fast, making further splitting under certain cases inefficient.

3.2 Proposed Local Branch Optimization

When a triangle overlaps with the splitting plane, we push it to both the left and right children as naive KD-build specifies only leaves can contain triangles. However, this leads to a case where

large triangles that span an entire node are duplicated to all the leaves in that branch, making traversal redundant as it always calculates the same ray-triangle intersection in every leaf node. We propose a new algorithm for this case: local branch optimization. When we encounter an overlap that spans *the entire subtree*, we do not duplicate, but instead, test for intersection in that node as we move down the tree during traversal. If an intersection is positive in a non-leaf node, we continue traversing the leaf nodes in the branch as non-leaf node solution may not be optimal. We implement local branch optimization by adding a pull-up component to KD-build. When we move up the tree, pull-up recursively removes the same triangle from two children nodes and adds it to itself. This modification works with the SAH because the pull-up component does not affect the SAH (all nodes: SAH and termination modified by constant, K_I; parent nodes: add K_I back, T remains constant regardless).

However, naive local branch optimization also leads to a speed decrease because if we test positive for an intersection in a non-leaf node, which also happens to be the answer, we still have to traverse until we find a nonoptimal solution in a leaf node or we search the entire subtree. We can modify KD-traverse so it keeps track of the minimum distance (closest object encountered so far), and if the enter distance is greater than the this minimum distance (see Figure 2), then we can avoid traversing the child, allowing us to not only avoid the detriment but also improve traversal time.

We test the algorithm's performance on four diverse scenes (also used in our denoising and superresolution analysis), Cornell Box with mirrors (lighting and reflection), Car in a red box (lighting and texture), Bedroom (complex geometry), and the Sponza Atrium (standard rendering model), which were obtained from an online source [17]. From Table 1, we see the algorithm reduced the number of triangles stored in the KD-Tree by around 30% and improves traversal speed by up to 40%. The free memory allows us to render more complex scenes without memory thrashing which can slow down rendering 10x [5], and the speed boost allots more than enough time for us to run our post-processing enhancement.

Table 1. Number of Triangles in KD-Tree and Time Before and After Local Branch Optimization

	Cornell Box	Car	Room	Sponza
Triangles (Original)	2188	10264	340265	262267
Triangles (Unoptimized)	12440	97248	2619894	1739676
Triangles (Optimized)	9073	67293	1738934	1206221
Avg Time (Unoptimized) (s)	6.52	10.51	94.10	128.37
Avg Time (Optimized) (s)	5.69	7.66	56.91	98.16

*4spp (samples per pixels) renderings for 40 300x300 comprehensive (spatially well-distributed) camera shots selected per scene on one 2.7 GHz core w/o antialiasing.

3.3 Parallelization

Raytracing is embarrassingly parallel. Each pixel's value is calculated independently of other pixels. We parallelize our path tracer with Open Message Passing Interface (MPI). A master node assigns

itself and the worker nodes the rows of the image and assembles the final rendering together. This design allows the path tracer to run on many cores. For consumers, our goal is highly interactive path tracing on 4 to 8 core CPUs (Intel i5, i7, dual). Because there is no guarantee all cores have shared memory, each core reads and stores its own data and builds the KD-Tree in pre-processing. Thus, we are constrained by the memory limit of one core (usually 8 GB), which is the reason we want to optimize memory usage.

4 Monte Carlo Noise Filtering

Our denoising network learns a mapping between noisy and noise-free renderings by training on noisy and noise-free image pairs. We optimize the reconstruction method proposed by Chaitanya et al. for the single CPU. Their OptiX-AI is a denoising autoencoder [4], an encoder-decoder network that converts the input to some internal representation and then reconstructs it from that representation [3]. A denoising autoencoder reconstructs noise-free outputs from noisy inputs. OptiX-AI consists of 10 convolution blocks, each consisting of three convolutional layers, 2x2 max pooling or 2x2 nearest neighbor upsampling, and a skip connection to the respective decoder block for every encoder block. Inner blocks contain 4/3 more filters, leading to a fixed compression ratio of 3 after every max pooling. The structure was intended for evaluation on a GPU [4], making it too costly for the single CPU, especially for highly interactive rendering of simple scenes.

4.1 Denoising Autoencoder

Our network consists of encoder-decoder phases with 10 convolution blocks. Each encoder block has 2x2 max pooling and a skip connection to the respective decoding block, and each decoder block has 2x2 upsampling. Every block contains only one convolutional layer. The number of filters per layer is also fixed at 32, so our network has a compression ratio of 4. The execution time for our network is minimally 3x faster than OptiX-AI, and 30x faster than the network proposed by Mao et al. Since our network is fully convolutional, we can operate on inputs of any dimension [15]. To accelerate training, our training data consists of noisy and noise-free 128x128 crops from the corresponding 512x512 noisy and noise-free renderings. Like OptiX-AI, our network can also use any number of auxiliary inputs. We choose to utilize the diffuse map to help the network distinguish between image regions while providing information about lighting and color.

4.2 Precision Graphics

Our lightweight network only contains 95,107 trainable parameters, much less than OptiX-AI's 3.2 million [4] and Mao et al.'s network. This presents two major issues. First, the lightweight architecture greatly reduces its ability to learn many complex relationships between scene geometry, lighting, and noise. Furthermore, our network must be able to generalize to any scene without overfitting on any particular data characteristic, which is very difficult without very deep CNN architecture [2]. Chaitanya et al. note OptiX-AI needs data with sufficient scene variety to perform well but states it can avoid this potential pitfall by 'specializing' in an individual scene by training only on data from that scene [4]. We can also use this approach to improve our network's performance since the network only has to learn mappings constrained to one scene.

We formalize these concepts into Precision Graphics, where every scene has a corresponding 'specialized' version of our denoising autoencoder. That means for every new scene, the designer must generate noisy and noise-free pairs of scene data on a computing cluster or GPU. But scene

design, especially for simulations and games is restricted to professionals, and the emergence of contracted cloud computing like Amazon's AWS makes computational resources readily available to professionals.

4.3 Training

We propose a loss function that is a weighted sum of the mean squared error (MSE) of the pixel-wise differences and the MSE of the noisy rendering (diffuse map shadows are completely black) convolved with the Laplacian of Gaussian (LG) filter (blur then edge detection), allowing our network to learn fine details. We set the LG filter's standard deviation to 1.4 to filter out Monte Carlo noise and normalize the edge map into the interval from 0 to 1.

$$L_{MSE} = \frac{1}{N}\sum_{i}^{N}(P_i - T_i)^2$$

$$L_{LG} = \frac{1}{N}\sum_{i}^{N}(P_i' - T_i')^2$$

$$L = 0.7L_{MSE} + 0.3L_{LG}$$

where P_i and T_i are the ith pixels of the predicted and target image and P_i' and T_i' are the ith pixels of the predicted and target image convolved with the LG filter and normalized.

The noisy and noise-free renderings needed for training and evaluation were generated on a 256 core computing cluster over several days per scene. Each dataset contains 20 comprehensive camera positions within the user movement area, and each position produces 16 512x512 noisy, noise-free (reference), and diffuse map triplets with views oriented at different angles in increment of 22.5 degrees. From these 320 images, we randomly obtain 32 128x128 crops, producing a total of 10240 triplets, which is randomly split into 7500 training, 1370 validation, and 1370 test triplets.

5 Image Superresolution

Consider a rendering where every other pixel is path traced, and we must determine the missing pixels from the filled pixels. Traditional methods for image superresolution involve some kind of interpolation, leading to blurring and loss of fine details. We enhance the interpolated image with a network that learns a mapping between it and its high-resolution counterpart.

Increasing image dimensions 2x increases generation time 4x, making data generation for Precision Graphics for superresolution is impractical. Unlike denoising where the entire noisy image has to be reconstructed, interpolation yields visually decent results for interiors of bounded regions without fine textures, so deeper networks only increase in PSNR by a few dBs [14]. Thus, Precision Graphics is also redundant as network size does not affect quality too much. Path tracing produces very realistic images, which motivates us to use photo crops as our training data. This makes the superresolution network generalizable to all scenes, reducing dependence on Precision Graphics.

5.1 Lightweight CNN

Our goal is not to reconstruct an entire high-resolution rendering from a low-resolution, but to enhance image features to make the interpolation appear more realistic. Thus, we operate exclusively on the Y channel (luminance) of the YCbCr colorspace since that preserves color and region

boundaries while allowing changes in the intensity gradient [12]. We modify the lightweight three-layer SRCNN architecture proposed by Dong et al. We choose to use 9-3-5 kernel sizes for the layers, which corresponds to in SRCNN patch extraction and representation, nonlinear mapping to a higher-dimensional vector, and reconstruction [6]. Each layer, however, contains 32 filters. Our model contains 37,793 parameters compared to SRCNN's 8,032 parameters. The superresolution network is also fully convolutional. We intend to input the 512x512 output from the denoising network and increase the image size to 1024x1024.

5.2 Training

We use the same loss function as the denoising network because of its ability to account for edges but set $\sigma = 1$ to preserve edge map sharpness. The data consists of 20000 256x256 random image crops from the 2014 MS-COCO validation dataset, randomly split into 15000 training, 2500 validation, and 2500 testing. The training input was created by downsampling each image to 128x128 and bicubically interpolating it back to 256x256. We also generated 60 (15 each scene) 1024x1024 renderings to verify the network's ability to generalize what it learned from photo crops to computer renderings.

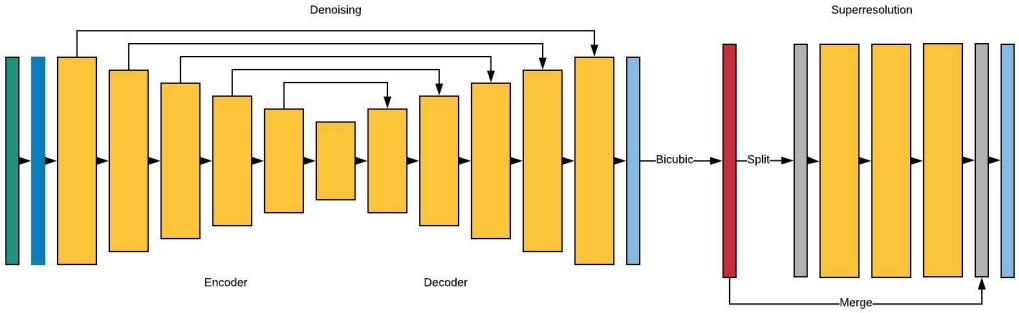

Figure 3: Our overall post-processing architecture. Yellow blocks represent convolutional blocks. Dark blue represents the noisy image, green diffuse map, light blue clear image, red intermediate buffer, and grey Y channel

6 Results

We implemented our algorithms in Keras with Tensorflow backend. The models were trained on a Nvidia GeForce GTX 1060 GPU. We use Cornell Box, Car, Room, and Sponza as before. The first two scene's reference renderings were path traced with 4096spp while the last two with 1024spp. We begin by analyzing our network training behavior. Then we investigate our denoising and superresolution networks' performance. Last, we discuss the application of our system to interactive and potential for realtime path tracing. For measuring of reconstruction quality, we utilize the *Root Mean Squared Error* (RMSE) and *Peak Signal to Noise Ratio* (PSNR). The RMSE is used in place of the MSE to reflect the actual pixel-wise differences. The PSNR is a popular metric that depends on I, the maximum pixel value (we use 0-1 normalized images) [24]. Acceptable values for PSNR are above 30 dB [27].

6.1 Training Behavior

We trained the networks for 100 epochs with a batch size of 32 using the Adam optimizer. Our denoising autoencoder is small, so it rapidly converges in the first few epochs of training shown by the stagnation of validation loss. After the 10th epoch, it only improves a few dBs in PSNR (see Figure 4). Thus, near-optimal results can be obtained without using a GPU for training, reducing Precision Graphics' dependence on computational resources. We observed a similar trend for the superresolution CNN. Our datasets are large enough to prevent overfitting on the training data, shown by the validation PSNR consistently matching or even exceeding the training PSNR (see Figure 4). For our analysis, we used the model with the lowest validation loss.

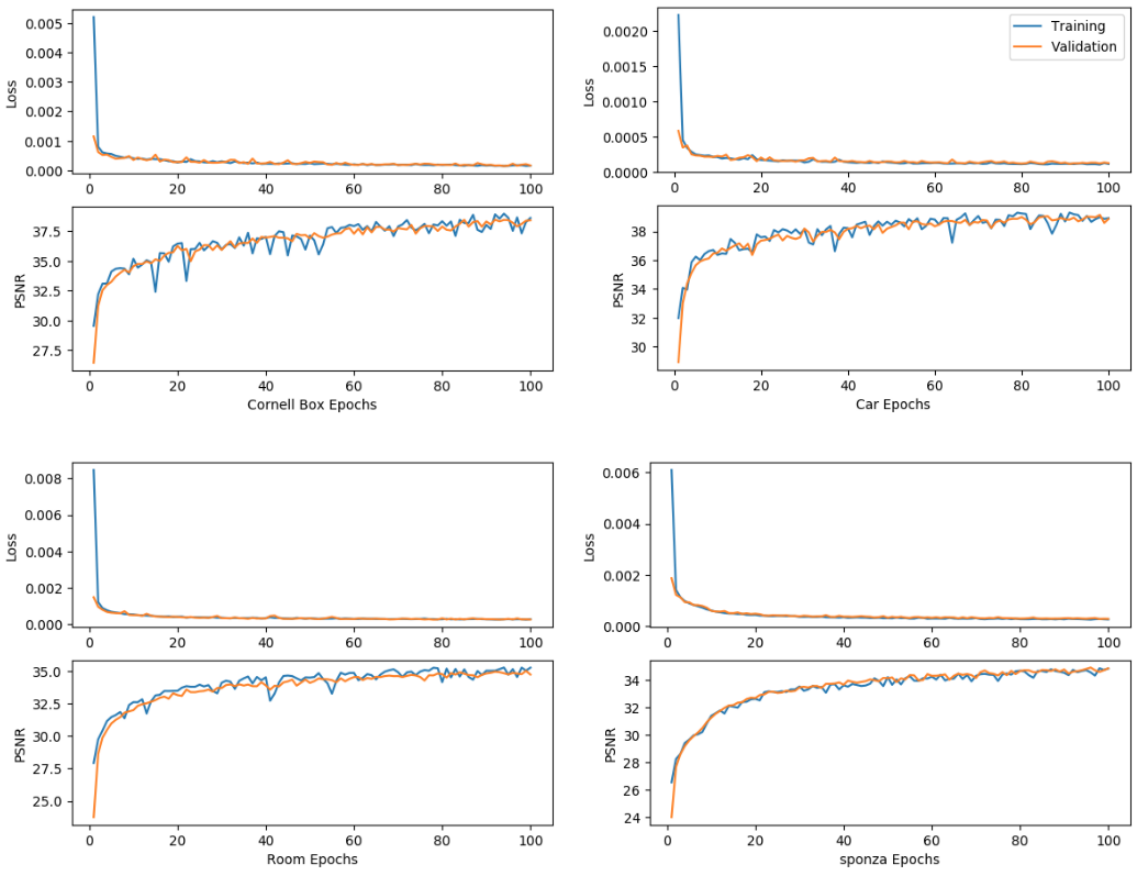

Figure 4: We show the loss and PSNR for the training and validation sets over the 100 epochs. Notice our networks begins converging around the 10th epoch, and training beyond that yields little improvement.

6.2 Denoising Quality and Precision Graphics

From Table 2, we see our autoencoder performs well on individual scenes, far exceeding the PSNR lower bound. For the standard model, Sponza, Precision Graphics's RMSE of 0.019 is comparable to OptiX-AI's 0.016 [4]. We compare Precision Graphics to a general purpose model (called mixed model) with the same denoising architecture but trained on 7500, validated on 1370, and tested on 1370 renderings randomly selected from the four datasets. The mixed model performs the worst on its own test set compared to the Precision Graphics models. From Table 3, we see it performs worse than the individual models for each scene's test data. This is attributed to less exposure individual scene characteristic in favor of learning a general relationship between noisy and noise-free renderings. We show this is ineffective by testing the mixed model on a fifth scene, Pine [17], a tree model contained in Car's box but with different wall colors, textures, and lighting (more light sources, stronger illumination). The model's PSNR drops 8 dB below the accepted limit, the RMSE is more than 3x greater than the worst performing Precision Graphics model, and artifacts evolve (see Figure 6).

Qualitatively, Precision Graphics models reconstruct shadows, edges, color, and lighting more accurately than the mixed models (see Figure 5), which sometimes results in alien artifacts (Cornell Box) mostly likely induced from learning from more than one scene. However, our model has certain limitations. It degrades reflections and certain output regions contain uneven coloring. Moreover, its performance is constrained by the thoroughness of the data sampling carried out by the scene designer.

Table 2. Denoising Algorithms Performance

Scene	**Training PSNR**	**Validation PSNR**	**Test PSNR**	**Test RMSE**
Reflective Cornell Box	38.45	38.97	38.62	0.012
Car	38.81	39.31	39.04	0.011
Room	34.78	35.19	34.92	0.018
Sponza	34.57	34.84	34.42	0.019
Mixed	34.47	34.89	34.39	0.019

Table 3. Mixed Scene Denoising Algorithm Performance

Scene	Cornell Box	Car	Room	Sponza	Pine
Test PSNR	36.33	37.53	33.69	32.37	22.79
Test RSME	0.015	0.013	0.021	0.024	0.074
PSNR Difference	2.29	1.51	1.23	2.05	

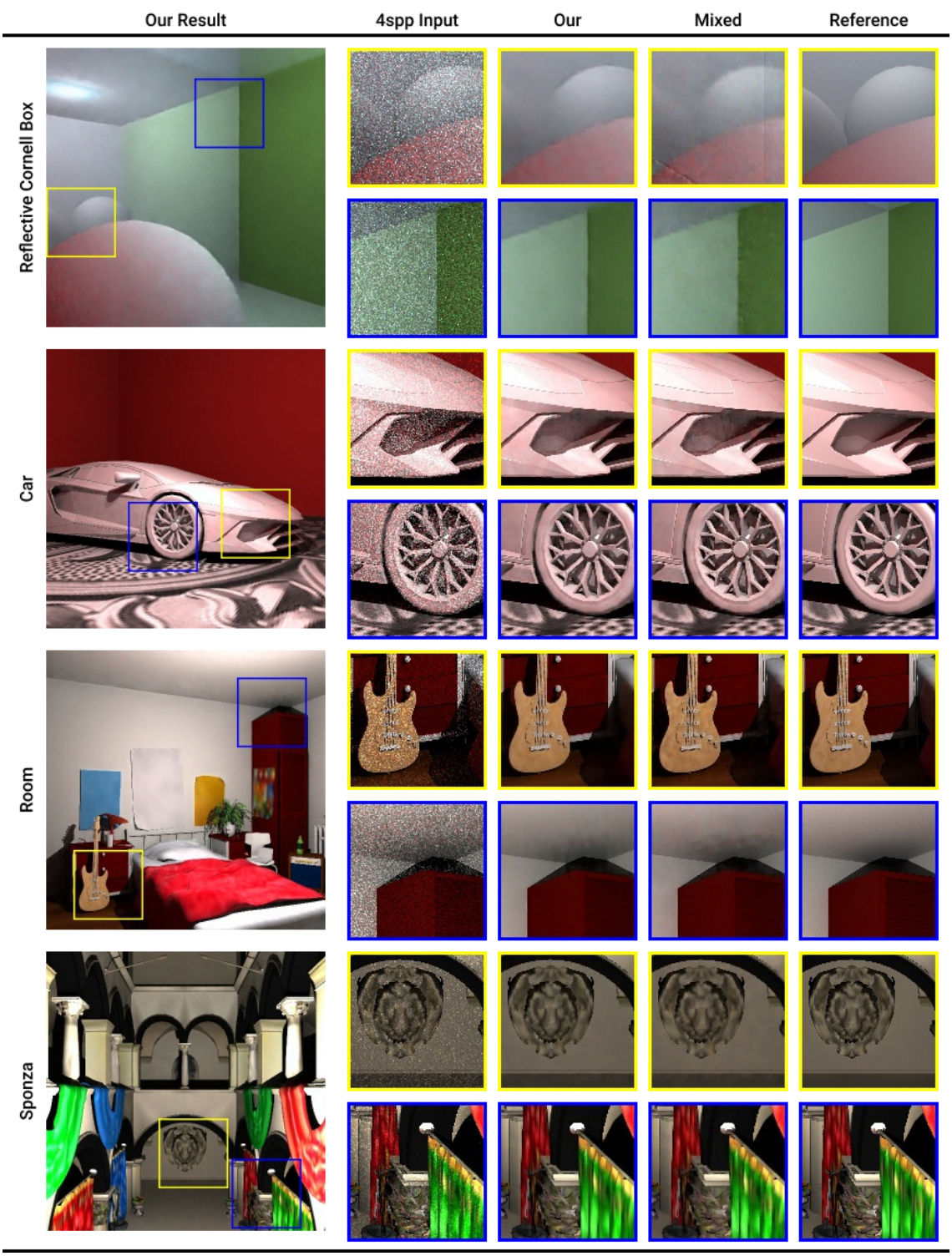

Figure 5: Our Precision Graphics models reconstructs the noise-free rendering very well. Notice how color unevenness and artifacts evolve in the mixed model.

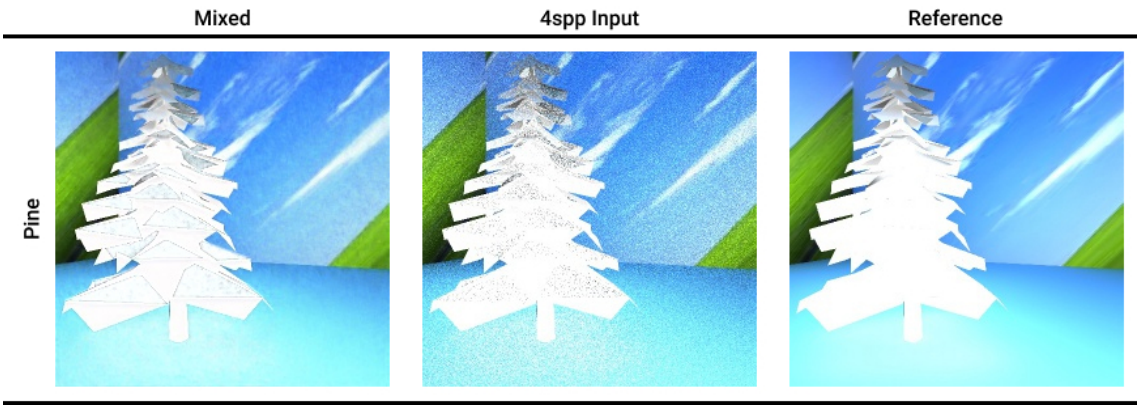

Figure 6: Without prior knowledge of the scene, the model fails to properly remove noise and misinterprets noise distributions (see leaves).

6.3 Superresolution Quality

We first choose not to use the output of the denoising network as input during initial testing but instead downsample the 1024x1024 verification renderings to 512x512 and interpolate back to 1024x1024 to create the input, so our analysis is independent of the denoising component. From table 4, we see our model performs extremely well on computer renderings, exceeding the interpolation PSNR for all but Cornell Box, which may be due to the loss function not handling reflections well. The higher performance on the renderings can be attributed to them being less blurry than actual photos, so the inputs (interpolations) are of higher quality. Qualitatively, we observe that the superresolution provides a nice post-processing to the interpolation by enhancing image features and increasing sharpness (see Figure 7). Note the images in Figure 7 are downsampled from 1024x1024 to 256x256, so finer edges show up as aliasing, and we see that the reference and our results match in aliasing. However, our algorithm has trouble restoring features when most of the information is lost or indistinguishable due to blurring in the interpolation, evident by the poor quality of the blinds in the superresolved Room rendering.

Table 4. Superresolution Algorithm Performance

Standard Dataset	**Training PSNR**	**Validation PSNR**	**Test PSNR**	**Test RMSE**
MSCOC-14	26.55	26.49	26.37	0.048
Scene	**Bicubic PSNR**	**Bicubic RMSE**	**Test PSNR**	**Test RMSE**
Cornell Box	46.50	0.005	42.63	0.007
Car	37.78	0.013	39.70	0.011
Room	33.45	0.022	35.28	0.018
Sponza	36.27	0.016	38.91	0.011

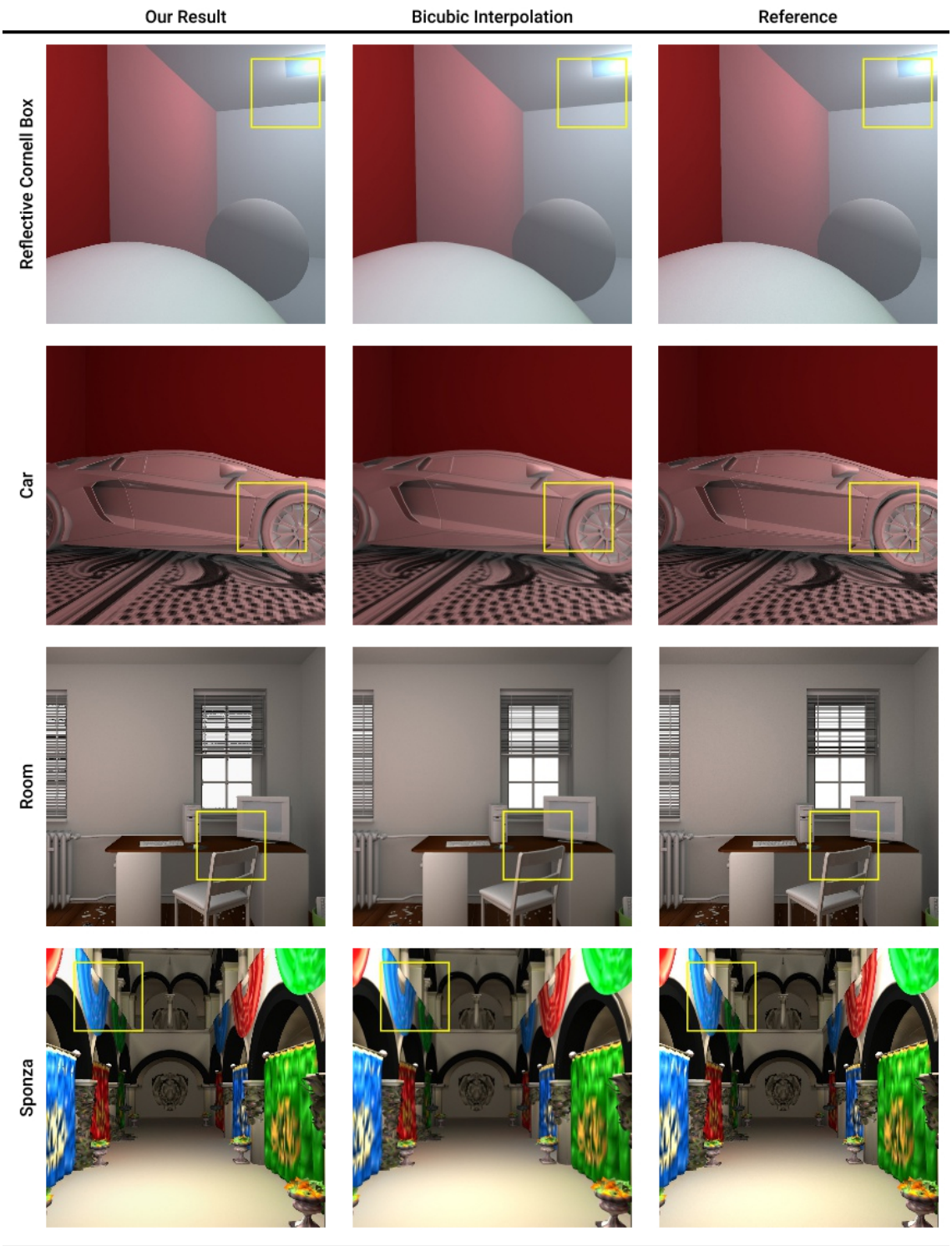

Figure 7: Notice the similarity in aliasing between our result and the reference image. This is because the figures shown were downsampled from 1024x1024, so finer edges appear low in quality while blurry counterparts appear more visually pleasing. In 1024x1024 space however, blurring is evident, destroying sharpness and immersion.

6.4 Interactive Rendering

Our combined post-processing achieves an evaluation time of about 3 seconds per frame, well within the time saved from Local Branch Optimization for medium (Car) and complex scenes (Room, Sponza). The final PSNR of the combined algorithm is lower than that of the denoising output but still is acceptable as each individual component achieves a much higher PSNR than 30 dB. We roughly estimate the final PSNR by adding the RMSE of the denoising and superresolution RMSE's to get the combined RMSE and solving for PSNR. We obtain values for Cornell Box, Car, Room, and Sponza of 34.64, 33.13, 28.92, and 30.32, respectively. In practice the values are higher because interpolation blurs the image, ridding of unevenness in color while the superresolution component enhances the edges. To show this, we test the combined algorithm on four comprehensive Cornell Box renderings, obtaining a PSNR of 37.21 and RMSE of 0.014 (see Figure 8).

A time of 3 seconds would be a bottleneck for simple scenes and consume much of the time allotted for interactive rendering. Since our networks are fully convolutional, we can parallelize the evaluation process by splitting the input into n rectangles where n is the number of cores. In this case, there are no overlapping convolutions between regions, resulting in a negligible error. The different methods (4 cores for parallel) for the Room scene below produced an RMSE of 0.0023. This allows us to achieve highly interactive image reconstruction. From the values in Table 1 and 5, the total rendering times (calculated by adding the corresponding time values from these two tables) on one core for the scenes are 8.69s, 10.66s, 58.91s, and 101.16s. On a 4 core CPU, all times are less than 25s for rendering a photorealistic 1024x1024 image. Thus, we achieved interactive path tracing (highly interactive rendering for small scenes) for the single CPU.

Table 5. Network Evaluation Times

Scene	Avg Time Denoising (s)	Avg Time Superresolution (s)
Cornell Box	1.92	1.13
Car	1.92	1.11
Room	1.85	1.15
Sponza	2.04	1.18

*40 512x512 images avg on one 2.7 GHz core

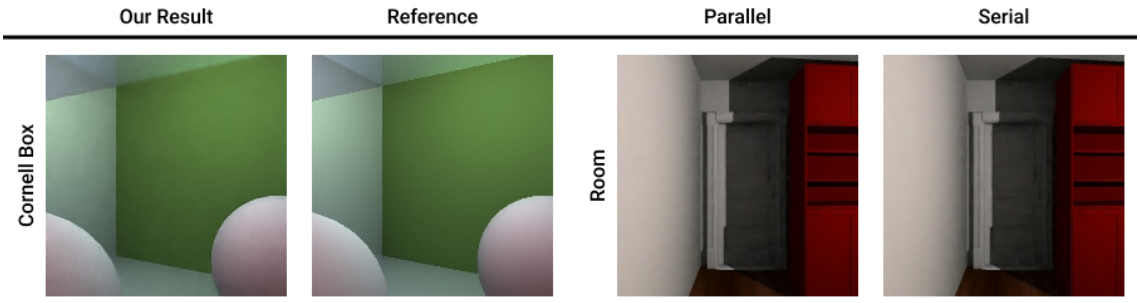

Figure 8: Left and Middle Left: Our post-processing algorithm is very robust, achieving a PSNR of 37.21 and RMSE of 0.014. Visually, the reconstruction is has little noticeable difference compared to the high-quality rendering. Middle Right and Right: Notice parallel evaluation results in no visible change to the network output.

7 Conclusions and Future Work

We described a series of optimizations that brings interactive and highly interactive path tracing to the modern single CPU. We reduced the cost of each ray cast by modifying KD-build and KD-traversal to efficiently handle triangle splitting plane overlaps. We decreased the number of primary and sampling rays needed by using learning algorithms to enhance low-quality renderings rendered using extremely low ray budgets.

In the future, we wish to explore more approaches. First, the final result still contains color unevenness, which we can address by using a bilateral filter to blur the image while preserving edges or iteratively denoising the result, which has been shown to improve quality (Chaitanya et al.). The path tracer is the bottleneck to realtime path tracing, so we want to investigate how to reduce the number of pixels cast, preferably to one pixel without increasing trainable parameters. One solution is to integrate Precision Graphics with some continuous sampling scheme, where the scene and lighting information is collected continuously rather than once per frame, leading to higher-quality results over time. We also are trying to figure out how to increase image upscaling in superresolution from 2x to 4x or even 8x while preserving features and sharpness. Finally, we want to integrate our superresolution component with foveated rendering. In foveated rendering a camera tracks eye movement and renders where the eye is looking at the highest resolution, decreasing resolution as the distance from that location increases. As a result, rendering time improves 5-6x [8].

References

[1] H. A. Aziz. Virtual reality programs applications in healthcare. 2018.

[2] S. Bako, T. Vogels, B. McWilliams, M. Meyer, J. Novak, A. Harvill, P. Sen, T. DeRose, and F. Rousselle. Kernel-predicting convolutional networks for denoising monte carlo renderings. 2017.

[3] P. Baldi. Autoencoders, unsupervised learning, and deep architectures. 2012.

[4] C. Chaitanya, A. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila. Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder. 2017.

[5] P. Christensen, J. Fong, D. Laur, and D. Batali. Ray tracing for the movie 'cars'. 2006.

[6] C. Dong, C. Loy, K. He, and X. Tang. Image super-resolution using deep convolutional networks. 2015.

[7] D. Dorr, D. Schiefele, and D. W. Kubbat'. Virtual cockpit simulation for pilottraining. 2001.

[8] B. Guenter, M Finch, S. Drucker, D. Tan, and J. Snyder. Foveated 3d graphics. 2012.

[9] M. Hapala and V. Havran. Review: Kd-tree traversal algorithms for ray tracing. 2011.

[10] V. Havran. Heuristic ray shooting algorithms. 2000.

[11] N. Kalantari, S. Bako, and P. Sen. A machine learning approach for filtering monte carlo noise. 2015.

[12] S. Kolkur, D. Kalbande, P. Shimpi, C. Bapat, and J. Jatakia. Human skin detection using rgb, hsv and ycbcr color models. 2017.

[13] Y. Lecun, L. Bottou, Y.Bengio, and P. Haffner. Gradient-based learning applied to document recognition. 1998.

[14] C. Ledig, L. Theis, F. Huszar, J. Caballero, A. Cunningham, A. Acosta, A. Aitken, A. Tejani, J. Totz, Z. Wang, and W. Shi. Photo-realistic single image super-resolution using a generative adversarial network. 2017.

[15] J. Long, E. Shelhamer, and T. Darrell. Fully convolutional networks for semantic segmentation. 2015.

[16] X. J. Mao, C. Shen, and Y. B. Yang. Image restoration using very deep convolutional encoder-decoder networks with symmetric skip connections. 2016.

[17] Morgan McGuire. Computer graphics archive, July 2017.

[18] T. Moller and B. Trumbore. Fast minimum storage ray/triangle intersection. 2012.

[19] G. Papaioannou. Computer graphics course: Ray tracing. 2015.

[20] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, K. Morley M. McGuire, A. Robison, and M. Stich. Optix: A general purpose ray tracing engine. 2010.

[21] J. Rupard. Ray tracing and global illumination. 2003.

[22] M. Shiratuddin and A. N. Zulkifli. Virtual reality in manufacturing. 2001.

[23] E. Vasiou, K. Shkurko, I. Mallett, E. Brunvand, and C. Yukse. A detailed study of ray tracing performance: Render time and energy cost. 2018.

[24] T. Veldhuizen. Measures of image quality. 1998.

[25] I. Wald. Realtime ray tracing and interactive global illumination. 2004.

[26] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in o(n log n). 2006.

[27] S. Welstead. Fractal and wavelet image compression techniques. 1999.

[28] A. Williams, S. Barrus, R. K. Morley, and P. Shirley. An efficient and robust ray-box intersection algorithm. 2005.