AI and Machine Learning for Real-Time Ray Tracing

Chunan Huang, Jurgen Schulze; University of California, San Diego

Abstract

The most used technique for 3D rendering nowadays is Rasterization, which is faster but provides less realistic images than another technique, Ray Tracing. Although Ray Tracing generates amazingly realistic images, it carries a large computational cost and needs a great amount of time to be done. This makes Ray Tracing hard to be used in scenarios that require instant results, like real-time applications. If we can speed up the Ray Tracing process, we can obtain realistic images in real-time.

There have been multiple classic acceleration methods for Ray Tracing, and hardware companies like Nvidia also attempt to accelerate Ray Tracing with specific hardware designs. In this paper, I am about to dive into the new approach: accelerate Ray Tracing with Machine Learning and Artificial Intelligence techniques. With these techniques, I was able to speed up the rendering process of Ray Tracing and use it in a real-time application.

1. Introduction

My research includes two parts. For the first part, I investigated into recovering incomplete images with Neural Networks. The idea is that, during the ray tracing process, we can calculate a smaller number of pixels and only render a noisy image (the uncalculated pixels will remain black as noise). This would be faster than rendering a complete image. Then, we can recover the incomplete image with a pre-trained Neural Network. With this approach, we do not need to deal with the rendering details of Ray Tracing and directly work on a rendered image.

As to the second approach, I dived into the rendering details of Monte Carlo Path Tracing and speeded it up with Reinforcement Learning. The purpose of this is to reduce the noisiness of images rendered by Monte Carlo Path Tracing so that it can use less sampling rays. I focused more on this approach compared with the first one. The details of this approach will be mentioned more later.

2. Related Work

For my first approach, I used a GAN model with contextual attention[1], which is designed at the University of Illinois at Urbana-Champaign and Adobe Research. Among all networks I experimented on, this model provides recovered images with the best quality.

For the second approach, the main idea is proposed in a paper by Nvidia[2]. But it does not provide details of implementation, so how to use this idea in practice was my main focus during the research (in section 4.3). I adapted the idea and used it in a real-time Ray Tracing application I made.

3. Approach One

3.1 Data Preparation

Since my purpose for this approach is to recover noisy images with the Neural Network, I need to prepare training data for the network. An example of a noisy image is shown in Figure1. This image is a screenshot of the scene in the Ray Tracing program I made and was added the “grid” as noisy. To get multiple these screenshots as training data, I created a script that automatically takes screenshots on the scene at random positions and directions (shown in Figure2). With these images, I was able to train the network.

3.2 Results

The well-trained network was able to recover noisy images from the same scene. The result is shown in Figure3 and Figure4. The quality of all recovered images is very good. The resolution of the images I experimented on is 512 x 512.

3.2 Analysis

Although the network can recover the noisy images very well, there are several significant drawbacks to this approach. The first is that since the model was trained by images of a specific scene, when the model is used for another scene, the result is not good. In another word, one trained model can only be used by one application. Another problem is that the GAN model is very heavyweight, which means the training process can take a very long time to converge, especially on high-resolution images. Besides, since the complexness of the network, the recovering process of a noisy image also takes time; in this case, the time we saved by calculating fewer pixels can be wasted on recovering the noisy image.

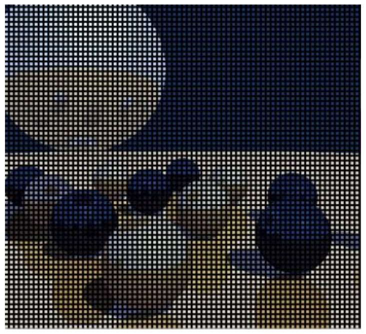
Figure1: Only calculate ¼ of the total pixels

Figure2: Screenshots as training data

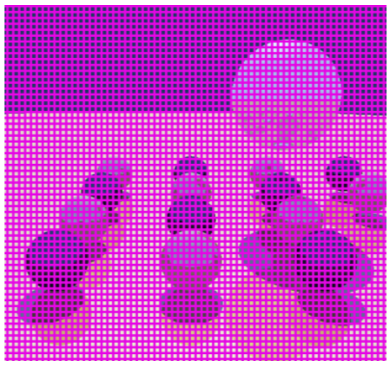
Figure3: Input noisy image

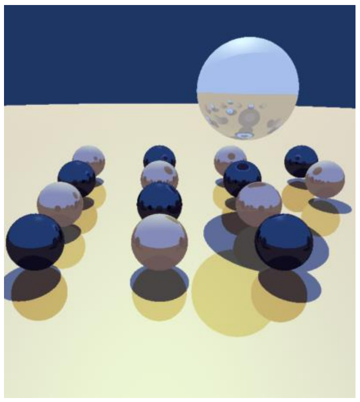

Figure4: The recovered image of Figure 3

4. Approach Two

4.1 Introduction

This approach is to improve the Monte Carlo Path Tracing (MCPT). In MCPT, for each pixel, the camera shots multiple rays (sampling rays), and the color of this pixel is the average of all rays' colors. The problem here is that, when the number of sampling rays is too low, the resulting images can be very noisy. Nonetheless, more sampling rays means a longer rendering time. So, we want to find a way to reduce the noisiness of rendered images without increasing the number of sampling rays.

Figure5 shows an image rendered with 8 sampling rays per pixel, and Figure6 shows an image rendered with 64 sampling rays per pixel. It is noticeable that more sampling rays mean higher image quality. The reason behind this is that, in MCPT, the reflection of rays is random. Therefore, some pixels can shoot rays that result in no color (didn't hit the light). When the number of sampling is low, these non-contributing rays affect the color of a pixel a lot and make the pixel dark. These dark pixels are the noise in the image.

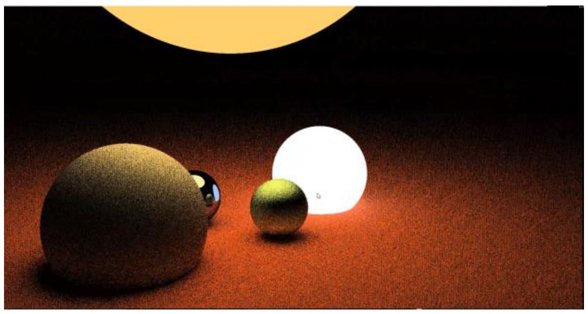

Figure5: 8 samples per pixel

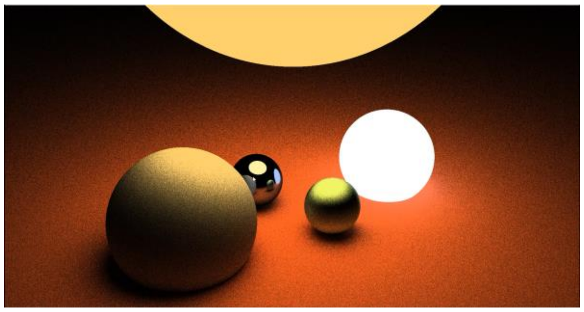

Figure6: 64 samples per pixel

Therefore, we want to reduce the number of rays with zero-contribution to decrease the nosiness of the rendered image. We can achieve this if the rays know where to find the light and try to hit it. Hence, we need to give rays the ability to learn, which can be achieved by reinforcement learning[2].

4.2 Overview of RL

The general idea of Reinforcement Learning (RL) is demonstrated in Figire7: the agent learns through its interactions with the environment. The agent takes an action and then transits to the resulting next state and receiving a reward. To maximize the reward, the agent will learn which action to choose in what state. After multiple attempts, the agent can learn which action is the best under a particular state.

The specific method I used is Q-learning, a model-free reinforcement learning technique. During the learning process, the Q-values for different states and actions, Q(s, a), are updated by the Bellman Equation :

$$Q(s,a) = (1-\alpha)\cdot Q(s,a) + \alpha \cdot \left(r(s,a) + \gamma \cdot \max_{a' \in A} Q(s',a') \right)$$

In general, what it does is: when the agent is at state S and chooses action A, which leads it to the next state S', the value of Q(S, A) will be updated according to the reward it gets at State S and the expectation of state S'.

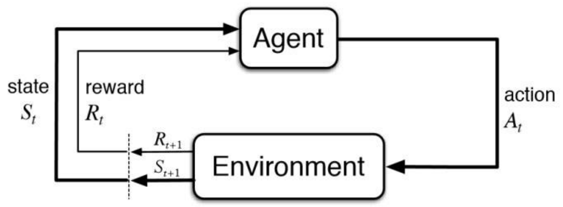

Figure7: Illustration of RL

4.3 Combine RL with MCPT

In the case of MCPT, the agent is a ray. The state is a segment where the ray hits the object and from which the ray reflects. The action is the reflecting direction. And the reward is the luminance of the object. Hence the desired outcome is that, when a ray hits an object (on a state), the reflecting ray will more likely to go in the direction that leads to light sources (choose a good action).

Segmenting the whole scene into states is simple. Figure8 shows the segments of the scene, with a different color representing a state. The difficulty is how to map the reflecting directions as actions because there are infinite possible directions for a ray, but we cannot have infinite actions under a state. To resolve this, I choose 26 "standard directions" as actions for the Q-learning process. As shown in Figure9, from the center of a 3 x 3 cube, there are 26 directions going to the outer 26 cubes. Each of these directions is an action during the learning process.

However, this does not mean that a ray can only choose one of these 26 standard directions when reflects. The reflecting direction is still chosen randomly, but after the reflecting direction is chosen, it will update the Q-value of its nearest standard direction. For example, a ray hit on a state S, and it randomly chooses the reflecting direction, D. The closest standard direction to D is A. So, the reward D gets will be used to update the Q-value of Q(S, A). With this method, a ray can still have infinite reflecting directions, but the actions of Q-learning are finite.

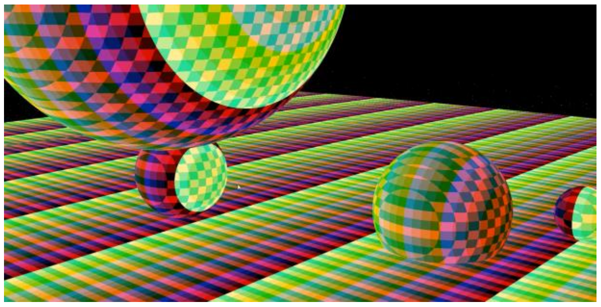

Figure8: segments of the scene

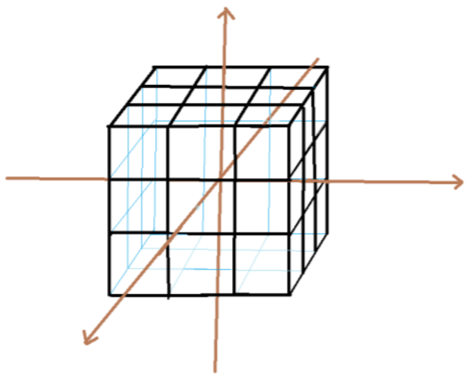

Figure9: 26 standard directions

Then, after the learning process, the learned result will be used to bias the decision of reflecting direction, encouraging the rays to hit the lights. When a ray reflects, the direction is chosen using Alorigthm1.

Algorithm1 Choose reflecting direction

```
1.  Ray hits at state $S$;
2.  $A$ is the set of 26 actions;
3.  Max Q-value at $S$, M_v= $\max_{a \in A} Q(S, a)$;
4.  for i = 7,…,1 do
5.      reflecting direction $d$ = rand_direction();
6.      Find $d$'s nearest standard direction $a$;
7.      if $Q(S, a) > i * 0.125 *$ M_v:
8.          break;
9.  end for
10. return $d$
```

With this algorithm, rays can still reflect in any directions, but more likely to reflect in a direction that will yield high reward (ie. hit the lights).

4.4 Results

Figure10 and Figure11 show the difference between using and not using RL during MCPT. For each image, the left half is rendered without RL and the right half with. All these three images are rendered with 8 sampling rays per pixel. We can notice a great difference in terms of the noisiness. Rendering with RL also makes the scene brighter overall, since the rays are more likely to hit the light sources now.

This approach also helps increase the image quality when the number of sampling rays is high. As shown in Figure12 and Figure13 (check the difference in noisiness of the floor). These images are rendered with 80 rays per pixel.

With this method, we can render images with the same quality using less sampling rays, which provides a great speedup.

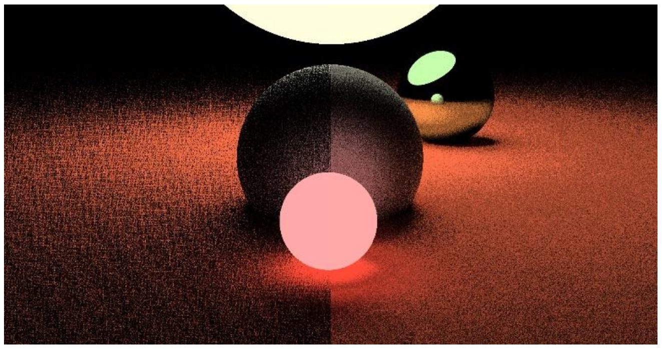

Figure10: 8 rays per pixel comparison

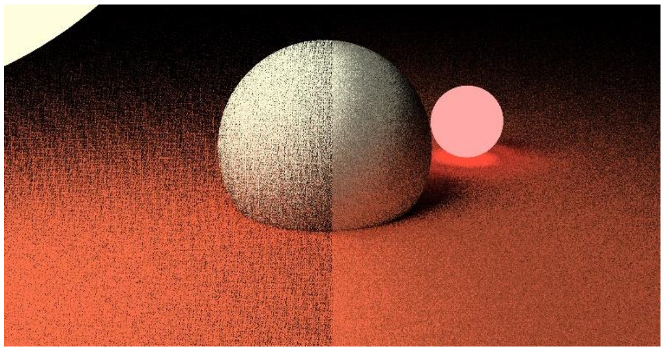

Figure11: 8 rays per pixel comparison

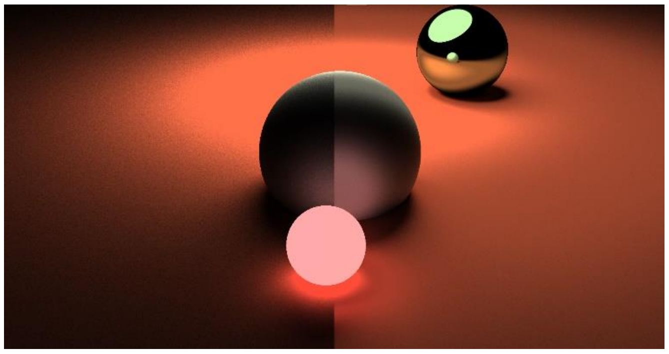

Figure12: 80 rays per pixel comparison

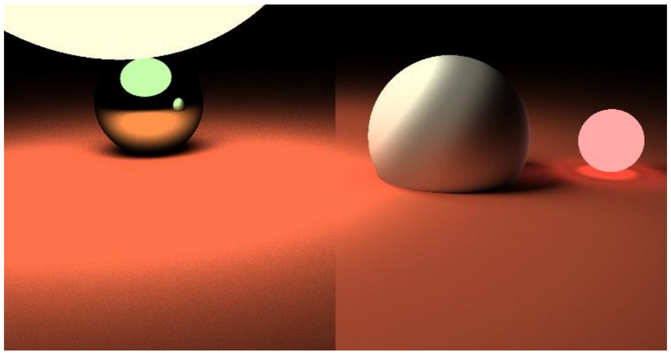

Figure13: 80 rays per pixel comparison

5. Conclusion

I experimented on two approaches to speed up Ray Tracing using AI/ML techniques. The first approach utilizes the current progress in GAN and directly deals with the rendered images; it can be used as a separate part and does not require any modifications of the rendering process. The second approach combines the AI technique, Reinforcement Learning, with Monte Carlo Path Tracing and enables it to render realistic images without using a large number of sampling rays.

6. Acknowledgment

This project was conducted by me, Chunan Huang, as an Honors Thesis Research under the supervision of Professor Jurgen Schulze at the University of California, San Diego. The machines used for the purpose of this research are provided by the Immersive Visualization Lab under Professor Jurgen Schulze at the University of California, San Diego.

7. Reference

[1] Jiahui Yu, Zhe Lin, Jimei Yang, Xiaohui Shen, Xin Lu, Thomas S. Huang. “Generative Image Inpainting with Contextual Attention.” Mar 21, 2018

[2] Ken Dahm, Alexander Keller. “Learning Light Transport the Reinforced Way.” Aug 15, 2017